Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Um conjunto de soluções para a construção de aplicativos de computação ubiqua

Fabricio Nogueira Buzeto

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientador
Prof. Dr. Ricardo Pezzuol Jacobi

Coorientadora
Profª Drª Carla Denise Castanho

Brasília
2010

Universidade de Brasília – UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado em Informática

Coordenador: Prof. Dr. Mauricio Ayala Rincón

Banca examinadora composta por:

Prof. Dr. Ricardo Pezzuol Jacobi (Orientador) – CIC/UnB
Prof.ª Dr.ª Alba Cristina M. de Melo – CIC/UnB
Prof. Dr. Mário Dantas – INE/UFSC

CIP – Catalogação Internacional na Publicação

Fabricio Nogueira Buzeto.

Um conjunto de soluções para a construção de aplicativos de computação ubiqua/ Fabricio Nogueira Buzeto. Brasília : UnB, 2010.
105 p. : il. ; 29,5 cm.

Tese (Mestre) – Universidade de Brasília, Brasília, 2010.

1. Computação Ubíqua, 2. Computação Móvel, 3. Middleware, 4. SOA

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro – Asa Norte
CEP 70910–900
Brasília – DF – Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Um conjunto de soluções para a construção de aplicativos de computação ubiqua

Fabricio Nogueira Buzeto

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Prof. Dr. Ricardo Pezzuol Jacobi (Orientador)
CIC/UnB

Profª. Drª. Alba Cristina M. de Melo
CIC/UnB

Prof. Dr. Mário Dantas
INE/UFSC

Prof. Dr. Mauricio Ayala Rincón
Coordenador do Mestrado em Informática

Brasília, 25 de Junho de 2010

Dedicatória

Dedico este trabalho a todos aqueles que acreditam na computação como uma ferramenta de transformação da sociedade.

Agradecimentos

Um trabalho como este não chega ao ponto de ser aceito por uma banca de mestrado sem ter tido a colaboração de muitas pessoas. Apesar de o autor do trabalho ser apenas um, o esforço de muitas pessoas foi desprendido para que se chegasse ao resultado apresentado, e com sucesso. Agradecer a todos que participaram deste processo é abrir brechas a falha de esquecer alguém muito importante. Apesar da dificuldade da tarefa me arrisco aqui agradecer a aqueles que colaboraram.

Primeiramente gostaria de agradecer a Deus que me deu forças, inspiração e energia para superar três anos, três meses, 18 dias e mais de 1000 horas de empenho neste projeto. Agradeço também por ter colocado pessoas muito especiais em meu caminho e ter me agraciado com bons frutos ao longo do tempo.

Agradeço à minha mãe, meu pai e meus irmãos que por mais que tenham visto um pouco de loucura na computação ubíqua não minimizaram os incentivos nesta minha empreitada.

Deixo meu agradecimento especial a Pah (Paloma Braga) que mais que companheira e revisora foi minha co-autora, assessora de marketing e injetora de energia ao longo de todo este trabalho.

Agradeço aos professores Ricardo Jacobi e Carla Castanho que apostaram em um aluno (na época) desconhecido para orientar nos caminhos nebulosos da computação ubíqua. Ao senhor Alê (Alexandre Gomes) que confiou a mim esta tarefa importante de dar prosseguimento em seu trabalho. Contei também com a colaboração de muitos professores como Jõao Gondim e Pedro Berger que compartilharam conversas, opiniões, e conhecimentos de suma importância ao longo do trabalho. Agradeço em especial a professora Alba Melo que me acompanha desde os tempos de graduação, sempre me auxiliando com dicas importantes nas pesquisas que desenvolvi e ao professor Mário Dantas que não só compartilhou idéias junto ao grupo como aceitou ser um dos validadores deste trabalho.

Muitos foram aqueles que botaram a mão na massa e deixaram a sua marca não só espiritual mais codificada no uOS. Fica então registrado um grande obrigado aos senhores Passarote (Estevão Passarinho), Lucas Lins, Lucas Quintella, Beatriz Marília e Marcelo Bassani que deixaram a marca de seus esforços nos fontes deste trabalho. Fica também a aposta nos senhores Carlos Botelho, Lucio Scartezini, Lucas Pessoa e Bruno Zumba que se dispuseram a continuar nesta nossa linha de pesquisa.

Fica também meu obrigado ao pessoal da família 13 e da Intacto pelo apoio dado ao longo desta empreitada. Em especial ao Carlitos que foi meu incentivador (e desertor) de engajar no mestrado, e me auxiliou nos momentos de aperto. Também fica o obrigado a equipe do Karmonitor que dividiu o laboratório comigo nestes últimos nove meses e se mostraram ótimos companheiros.

Por fim a Rosa e o Daniel da secretaria que salvaram muitas barras e deslizes meus e as "guardetes" dentre muitos outros que deixaram a marca de seus esforços

neste trabalho. Aos citados e muitos outros que por acaso eu venha a ter esquecido o meu muito obrigado.

Conteúdo

Lista de Figuras

Lista de Tabelas

Abstract

The amount of electronic devices around us nowadays increases in an everyday basis. In the same way, how we interact with these devices, and how they interact with each other, change the way we see the world. This scenario was shown by Mark Weiser in the 80's along with the idea of ubiquitous computing. Ubiquitous computing is about creating an intelligent environment that acts with the user providing simple ways of interaction and integration with the resources available.

To make this integration between users and resources in the environment possible, along with the integrations between the resources themselves in a scalable way, it's necessary to build architecture and software artifacts that make this task simpler. This dissertation presents a architecture based on SOA concepts for building applications in smart environments providing integrations between resources, called DSOA. Also is presented a group of communication protocols, uP, with portable characteristics and an implementation of a middleware that validates this architecture, the uOS in the UbiquitOS project.

Keywords: Ubiquitous Computing, Mobile Computing, Middleware, SOA

Resumo

A gama de aparatos eletrônicos presentes a nossa volta aumenta a cada dia, e da mesma maneira, a forma como interagimos com estes dispositivos, e como os mesmos interagem entre si, muda a forma como enxergamos o mundo. É este cenário que Mark Weiser apresentou na década de 80 junto ao conceito de computação ubíqua, caracterizada por um ambiente que interage de maneira inteligente com seus usuários, fornecendo interfaces simples de interação e integração entre os recursos disponíveis.

Para tornar essa integração entre o usuário e os recursos presentes no ambiente possível, bem como a integração entre os próprios recursos entre si de maneira escalável, faz-se necessária a construção de arquiteturas e artefatos de software que simplifiquem esta tarefa. Este trabalho apresenta uma arquitetura baseada em *SOA* adaptada para a disponibilização e acesso a recursos em ambientes inteligentes denominada *DSOA*. Também são propostos um conjunto de protocolos, denominado *uP*, de comunicação portável entre diversas plataformas e a implementação de um *middleware* (o *uOS*) baseado na *DSOA* e que utiliza o *uP*, cujo objetivo é validar a proposta.

Palavras-chave: Computação Ubíqua, Computação Móvel, Middleware, SOA

Capítulo 1

Introdução

O termo "ubíquo" tem origem na palavra latina *ubiquu*, que significa "aquilo que está ao mesmo tempo em toda parte". O conceito da "computação ubíqua" (*ubicomp*) surge quando a computação se encontra pulverizada nos aspectos da vida de cada um. Este termo foi introduzido em 1991 [63] e 1993 [64]. A visão é de uma nova era da computação, na qual esta se apresenta mais amigável e presente na vida do usuário. Porém esta presença não deve ocorrer de maneira intrusiva ou degradar as atividades do usuário. A interação do usuário e o ambiente devem ocorrer da forma mais transparente possível. Por tal razão, a *ubicomp* é também conhecida como "Computação Invisível". Esta "invisibilidade" tem origem quando o ambiente busca utilizar-se do mínimo de atenção do usuário [9], buscando antecipar ou facilitar suas ações. Desta maneira a tecnologia presente no ambiente deve ocupar a periferia da atenção do usuário, permitindo assim que este se concentre em suas atividades.

O que se constata atualmente é uma tendência do surgimento de diversos dispositivos computacionais no dia a dia (celulares, TVs, relógios, etc.). Mesmo itens do cotidiano que tradicionalmente não se encontravam ligados a componentes computacionais começam a ser incrementados com tal capacidade (camisetas [60] [46], sapatos [18], jóias [47], etc.). Sendo estes dispositivos pulverizados no ambiente e interligados por uma rede de comunicação, surge o ecossistema para a construção da computação ubíqua. Tais equipamentos, interagindo entre si são capazes de incrementar o ambiente com a inteligência necessária para concretizar a visão da *ubicomp*. A este ambiente inteligente é dado o nome de *smart space* [1], cujo objetivo é auxiliar o usuário em suas tarefas utilizando os recursos disponíveis.

Um *smart space* pode ser composto por uma grande variedade de dispositivos, que fornecem ao ambiente uma variedade de recursos. Para que estes agreguem as tarefas do usuário, a inteligência presente deve coordená-los de acordo com as informações sobre o ambiente e o usuário. A inteligência é provida através de aplicações que se executam nos dispositivos do *smart space*. Cabe a estas aplicações tratar as distintas plataformas dos dispositivos bem como a comunicação entre eles, para assim tomar ações adequadas.

Para que as aplicações possam tomar tais ações, estas devem estar cientes dos recursos e dispositivos disponíveis no ambiente. Elas devem ser aptas a realizar a troca de informações sobre o ambiente. Além disto, a mobilidade de dispositivos

e usuários no *smart space* deve ser considerada. Dentre os aspectos a serem observados no desenvolvimento dessas aplicações, podemos citar três de particular importância:

- Suporte a *dispositivos com recursos limitados* no *smart space*. Os dispositivos móveis tem limitações de memória, de capacidade de processamento, de largura de banda para comunicação e de consumo de energia. A inclusão destes dispositivos em um *smart space* requer que o sistema computacional subjacente permita o desenvolvimento de aplicações que respeitem essas limitações.

- Suporte à interação entre aplicações e recursos do ambiente. Neste caso, a comunicação pode ocorrer tanto de forma síncrona como de forma assíncrona. Além disto, os dados trafegados podem ser tanto estruturados e em pequena escala como em *streams* de dados.

- Suporte a *plataformas heterogêneas*, tanto em aspectos de *hardware* como de *software*. As diferenças entre plataformas devem analisadas buscando-se um modelo de comunicação que simplifique sua integração ao *smart space*. Com isto, aumentando a gama de dispositivos que possam participar do *smart space*.

Em [17], *Fouial*, *Fadel* e *Demeure* apresentam o conceito de adaptação de serviços, conceito este que serviu como base para a construção do projeto *UbiquitOS* [24]. A adaptabilidade de serviços apresenta uma proposta onde as funcionalidades estão disponíveis ao ambiente na forma de serviços e o acesso a eles é realizado de maneira transparente ao usuário. No projeto *UbiquitOS* é apresentada a implementação de um *middleware* para o desenvolvimento de aplicações de acordo com este conceito. Dentre os três níveis de adaptabilidade possíveis (Possível, Manual e Automático) esta implementação se restringe a possibilitar tal adaptabilidade de maneira manual.

Apresentaremos uma solução decomposta em três partes visando os pontos apresentados e a possibilidade de uma adaptabilidade automatizada dos serviços. A arquitetura *DSOA* se propõe a auxiliar na modelagem do *smart space* endereçando as principais questões dos ambientes de computação ubíqua tirando proveito dos conceitos fornecidos pela *SOA*. Baseda na *DSOA* foi elaborada uma interface de comunicação leve e multi-plataforma chamada de *uP* bem como o *middleware uOS* no auxílio para a construção de soluções neste contexto.

Este trabalho se encontra organizado da seguinte maneira. No capítulo 2 são apresentadas em detalhes as discussões acerca dos ambientes de computação ubíqua, seus desafios e principais soluções encontradas. O trabalho desenvolvido é apresentado no capítulo (cap:proposta) com maiores detalhes sobre cada solução nas sessões que se seguem. A *DSOA* (*Device Service Oriented Architecture*, sessão 4.1) consiste em uma extensão da *SOA* (capítulo 3) apresentando uma organização para ambientes inteligentes endereçando as principais questões encontradas pela *ubicomp*. O *uP* (*Ubiquitous Protocols*, sessão 4.2) consiste em um conjunto de protocolos que fornecem uma interface leve de comunicação entre aplicações e recursos em um *smart space*. O *uOS* (*UbiquitOS Middleware*, sessão 4.3) consiste

em uma evolução do *middleware UbiquitOS* de acordo com os conceitos apresentados pela *DSOA* e utilizando o *uP* como interface de comunicação. No capítulo 5 é apresentada a aplicação *Hydra*, cujo o protótipo construído apresenta as características do uso da *DSOA* em conjunto com o *uP* e o *uOS*. Neste mesmo capítulo são apresentados os resultados dos testes realizados na implementação do *uOS* e comparados a outras soluções encontradas. Por fim no capítulo 6 são apresentadas as considerações finais sobre este trabalho bem como possíveis linhas de trabalho abertas por esta pesquisa.

Capítulo 2

Computação Ubíqua

Para que uma tecnologia possa ser considerada ubíqua, esta deve se encontrar misturada ao cotidiano de maneira que seu uso seja natural e transparente às pessoas. Para o uso da tecnologia, estas não devem ter que se ocupar de detalhes técnicos ou complexos no seu dia a dia. Um bom exemplo de tecnologia ubíqua é a energia elétrica. Ao se ligar um aparelho na tomada, o usuário não se preocupa de onde se origina a energia muito menos em realizar algum ajuste no aparelho para utilizar aquela fonte. Para que a computação possa se apresentar como uma tecnologia ubíqua, esta deve mais que fazer parte do cotidiano, mas também favorecer a construção de um ambiente inteligente, o *smart space* [9]. Tal ambiente deve utilizar a computação de maneira que auxilie o usuário nas suas tarefas. Desta forma a atenção do usuário não deve ser altamente demandada, sendo requerida apenas quando necessário. Com isto a computação deve então ocupar a "periferia da atenção" do usuário, ou seja, estando presente no ambiente, porém se tornando evidente apenas quando necessária. Vejamos o exemplo que se segue:

> Paloma chega ao *shopping* e termina de estacionar seu carro na garagem. Ela está tranqüila, pois sua consulta com o dentista está agendada para as 17h e o relógio marca 16h40. Ao caminhar pelos corredores até chegar ao elevador que leva aos escritórios do prédio ela passa a frente a uma livraria. Neste momento seu celular emite uma mensagem. Acontece que o aniversário de um amigo se aproxima e Paloma tinha registrado a necessidade de comprar um presente para ele. O perfil dele mostra que ele adora livros de aventura, apresentando a livraria como uma conveniente parada no caminho. É curioso observar que nenhum alerta foi emitido à Paloma sobre o salão de beleza no qual ela acabara de cruzar. Mesmo existindo uma tarefa registrada para realizar um corte. Isto ocorreu, pois foi identificado que não existia tempo hábil na agenda para esta tarefa. Comprado o livro, Paloma segue seu curso e chega ao consultório onde é recebida por seu dentista que já a espera. Afinal, ao descer do carro ele já havia sido notificado da chegada de Paloma e sua consulta confirmada.

Este exemplo mostra como o ambiente interagiu a fim de facilitar a execução das tarefas do seu usuário. A inteligência que o ambiente possui possibilitou

que este tomasse ações sem a necessidade de intervenções do usuário (avisar o dentista) bem como a não tomar uma ação (não alertar sobre a necessidade de ir ao salão de beleza). Vemos que o usuário apenas interagiu com o sistema quando isto foi realmente necessário (alerta de presença da livraria). É esta inteligência que permite a tomada de ações que agreguem ao usuário sem afetar negativamente na execução de suas tarefas.

Para que tal inteligência esteja disponível observa-se a necessidade do desenvolvimento de aplicações que a implementem. Tais aplicações devem interagir entre si e junto aos recursos do ambiente, coletando informações e tomando as ações necessárias. É apresentado em [63] três pré-requisitos para a construção deste tipo de ambiente e estes serão discutidos na sessão 2.1. Posteriormente (na seção 2.2) serão apresentados alguns desafios envolvidos na construção destas aplicações e na seção 2.3 apresentaremos projetos que abordam tais desafios. Por fim, na seção 2.4 resumimos os dados apresentados em três requisitos principais que serão objetivados neste trabalho.

2.1 Pré-requisitos

Para que um ambiente inteligente possa ser consolidado é necessário que alguns elementos estejam presentes. De acordo com [63], temos três pré-requisitos para a formação de um ambiente de *ubicomp*, sendo eles relacionados ao *hardware* dos dispositivos presentes no ambiente, as redes de comunicação utilizadas para interação entre dispositivos e os sistemas computacionais responsáveis por coordenar estes componentes.

2.1.1 *Hardware dos dispositivos*

Cada vez mais é comum a miniaturização de dispositivos dotados de poder computacional. Hoje temos à disposição diversas opções de aparelhos que diretamente nos disponibilizam esse tipo de poder (*mp3-player*s, *palm*s, *laptop*s, celulares, computadores de bordo, etc.) bem como aparelhos que nos oferecem esta computabilidade de forma indireta (tênis com sensores [18], camisetas com sensores [60] ou que exibem imagens [46]). Complementar a tudo isso, as tecnologias de armazenamento e transmissão de energia evoluíram e hoje temos em desenvolvimento projetos de transmissão de eletricidade pelo ar, como o *WiTricity* [49] desenvolvido pelo *MIT - Massachusetts Institute of Technology* tornando o uso e surgimento de tais dispositivos mais integrado a objetos do cotidiano.

Para que tais dispositivos integrem de forma mais intrínseca a realidade, eles devem possuir um *baixo custo*. O baixo custo permite o fácil acesso a um número maior de usuários bem como possibilita a criação de ambientes em locais diversos. Agregando mais usuários, maior será a penetração da computação no cotidiano das pessoas. Outro fator a ser observado é que devido a mobilidade dos dispositivos existe a necessidade destes possuírem uma *alta eficiência energética*. Isto aumenta o tempo de vida das baterias permitindo uma mobilidade mais "natural" aos usuários.

2.1.2 Rede de comunicação

Para que os dispositivos no ambiente possam coordenar suas ações é necessário que estes estejam *interconectados por uma rede de comunicação*. Devido à mobilidade que muitos destes equipamentos, possuem além de meios físicos de comunicação como *Ethernet* ou *Gigabit Ethernet*, transmissões sem fio se mostram atraentes, possibilitando assim acesso simplificado aos dispositivos móveis. Neste sentido diversas tecnologias apresentam um grande avanço nesta área, como *Wi-Fi* [32], *Bluetooth* [25], *ZigBee* [4], *LED*s [55]. Além das tradicionais redes para a configuração de *LANs* também temos a evolução das redes formadoras de *WANs*, como 3G, 4G e *WiMax* [7].

Tais redes são necessárias à formação do ambiente inteligente e apesar dos avanços ainda existem desafios a serem destacados neste sentido. A integração entre redes locais (LANs) e redes mais abrangentes (*MANs* e *WANs*) aumenta o alcance das informações no ambiente além de necessitar do tratamento de rotas. Além disto, um mesmo *smart space* pode contar com diversas redes de comunicação, cada uma distinta entre si de acordo com os dispositivos presentes e as interfaces disponíveis neles. Neste caso é necessário tratar o tráfego de informações entre estas redes possibilitando agregar e integrar um maior número de recursos ao ambiente.

2.1.3 Sistema computacionais

Como visto, a *ubicomp* só é alcançada quando o ambiente utiliza-se dos recursos computacionais disponíveis em prol do usuário de maneira inteligente. As aplicações são responsáveis por viabilizar esta inteligência no ambiente, analisando os dados coletados através dos dispositivos e tomando ações adequadas. A fim de simplificar a construção destas aplicações é necessária a construção de *sistemas computacionais que facilitem esta tarefa*. Tais sistemas devem tratar os detalhes da camada física [36] e facilitar o acesso aos recursos por parte das aplicações. Desta maneira estas ficam responsáveis por tratar as interações realizadas junto ao usuário bem como outras informações vindas do ambiente, objetivando a tomada de ações transparentes ao usuário. Na construção destes sistemas é comum a utilização de *middewares* que abstraem as camadas inferiores do ambiente e coordenam as interações entra as aplicações dentro do *smart space*.

2.2 Desafios na construção de Aplicações para *ubicomp*

O *smart space* se apresenta como um complexo ecossistema envolvendo dispositivos, aplicações e usuários. A interação entre usuários e dispositivos demanda análise e coordenação constantes. Tais equipamentos devem ser gerenciados a fim de garantir o devido conhecimento das capacidades do ambiente. Os perfis do usuário devem ser conhecidos para garantir maior acurácia na tomada de decisões. Em [15] encontramos dez desafios enfrentados na construção de soluções para ambientes de computação ubíqua.

1. *Heterogeneidade*: Sendo o poder computacional pulverizado nos mais diversos elementos do ambiente, é de se esperar que existam divergências entre estes. Tais diferenças podem abranger a plataforma (*hardware* e *software*), tipo de rede de comunicação, e devem ser tratadas a fim de agregar o maior número possível de dispositivos ao ambiente.

2. *Escalabilidade*: Com o grande número de dispositivos, usuários e aplicações operando em um *smart space*, a escalabilidade é uma questão de suma importância a fim de se garantir uma interação agradável e consistente junto ao usuário. Além disto, o ambiente deve lidar com a volatilidade dos recursos e usuários que podem entrar e sair do ambiente de maneira dinâmica.

3. *Tolerância a falhas*: Em um ambiente tão volátil e rico em interações, lidar com as alterações no ambiente não pode levar o usuário a uma queda na qualidade dos serviços prestados, caso contrário não teremos a transparência almejada pela *ubicomp*. Adicionalmente, o ambiente deve prover disponibilidade aos recursos e informações garantindo a confiança do usuário no ambiente.

4. *Segurança*: Em um ambiente que conta com uma grande movimentação de usuários e dispositivos, garantir o controle de acesso a recursos e informações é um critério crítico a ser observado. Mesmo assim, esta é uma característica desafiadora a se alcançar de maneira compatível aos princípios de transparência almejados pela *ubicomp*.

5. *Integração espontânea*: A volatilidade do ambiente, tanto em questão de dispositivos quanto de usuários, deve ser tratada de maneira espontânea. Desta maneira novos recursos disponíveis e novos usuários podem interagir junto ao ambiente de maneira não burocrática.

6. *Mobilidade*: Diversos componentes computacionais estão presentes junto ao usuário (celulares, relógios, jóias, etc.), a mobilidade destes dispositivos e seus recursos no ambiente deve ser considerada pelos sistemas presentes. Além disto, o próprio usuário possui mobilidade junto ao ambiente e as aplicações devem tratar esta questão conforme cada caso e tipo de interação.

7. *Sensibilidade ao contexto*: Compreender o estado do ambiente a partir das informações disponíveis bem como deduzir as conseqüências da alteração deste estado é necessário que o sistema atue de forma proativa no *smart space*.

8. *Gestão de contexto*: Conhecendo o contexto no qual o usuário se encontra no ambiente e suas relações, é permitido então a este a tomada de decisões que levem a ações em prol do usuário.

9. *Interação transparente com o usuário*: As formas de interação entre as aplicações e o usuário devem explorar características e habilidades que permitam um contato mais natural e menos intrusivo, permitindo que o usuário concentre seu foco na tarefa e não na interação.

10. *Invisibilidade*: Esta característica remete ao objetivo primordial da computação ubíqua. Todas as ações, interações e decisões tomadas pelas aplicações e mesmo como o ambiente é construído e modelado deve permitir que o usuário mantenha seu foco nas tarefas e não nos sistemas que o rodeiam.

Podemos observar nestes desafios a consistência com relação à invisibilidade das aplicações no ambiente bem como seu foco em facilitar as tarefas do usuário. Por abranger as diversas aplicações que podem compor um *smart space*, tratar cada um destes desafios individualmente por cada aplicação não se mostra adequado. Por isso é comum a utilização de *middlewares* e a definição de protocolos que visem facilitar a interação e coordenação das aplicações e recursos dentro do ambiente.

2.3 Estado da arte

As aplicações são a chave para se obter a inteligência necessária para a construção do *smart space*. E como já apontado nos pré-requisitos da *ubicomp*, tais aplicações devem ser apoiadas por sistemas computacionais que facilitem seu desenvolvimento e coordenação no ambiente.

No intuito de auxiliar a construção destes aplicativos, uma das abordagens mais empregadas é a utilização de *middlewares* [6]. O *middleware* consiste em uma camada de *software* cujo o objetivo é abstrair as complexidades das camadas inferiores, tais como *hardware*, sistema operacional e tecnologia de rede.

Apresentaremos aqui algumas iniciativas da academia e da indústria no intuito de facilitar a construção de ambientes ubíquos.

2.3.1 AURA

O projeto *Aura* [23], desenvolvido pela *Carnegie Mellon University*, consiste em uma plataforma para computação pervasiva de auxílio a atividades do usuário. O *Aura* visa utilizar a plataforma de *software* do ambiente no qual o usuário se encontra, a fim de prover suporte à continuidade de suas atividades conforme este se desloca entre ambientes ou caso o mesmo ambiente sofra alterações em seu estado. O *Aura* determina o contexto do usuário com base nas aplicações que este utiliza. Conforme o usuário troca de ambiente ou dispositivo o ambiente sugere um conjunto novo de aplicações que forneça funcionalidades compatíveis. O objetivo do *Aura* é atuar como um invólucro que adapta o ambiente as tarefas do usuário, provendo continuidade nestas.

2.3.2 GAIA

O projeto *Gaia* [12], desenvolvido pela *University of Illinois*, propõe uma nova visão dos recursos presentes no *smart space*. Neste projeto, o ambiente formado por diversos dispositivos e seus recursos é conhecido como *Active Space*. Para auxiliar a construção de aplicativos no *Active Space*, o projeto *Gaia* propõe a construção de um *middleware* para a criação de uma abstração de sistema operacional (*GaiaOS*) que proporcione uma visão única do ambiente, de forma que

todos os recursos presentes sejam visíveis aos aplicativos como recursos em uma única máquina. Para tal, o *GaiaOS* propõe uma derivação da arquitetura *MVC* [48] (*Model View Controller*) aplicada ao contexto da computação ubíqua. Esta arquitetura é denominada *MPACC* [54] (*Model Presentation Adapter Controller Coordinator*) e consiste no acréscimo de duas novas entidades, o *Adapter* (Adaptador) e o *Coordinator* (Coordenador), que realizam as tarefas de integração das interfaces das aplicações e dos recursos. Esta arquitetura é implementada através de mecanismos centralizados de *RPC*, análise de contexto, gerência e descoberta de recursos.

2.3.3 Gator Tech Smart House

O projeto *Gator Tech Smart House* [34], desenvolvido pela *University of Florida*, é a segunda investida do *Mobile and Pervasive Computing Laboratory* em desenvolver uma casa inteligente (seu primeiro projeto foi o *Matilda Smart House* [39]).

Neste projeto foi construída uma casa dotada com um vasto conjunto de sensores (movimento, localização, pressão e temperatura), câmeras, atuadores e outros dispositivos (telas, controles, aquecedores, entre outros). Seu objetivo é a construção de um ambiente inteligente simulado, contando com a presença de diversos membros pesquisadores interagindo nesta estrutura.

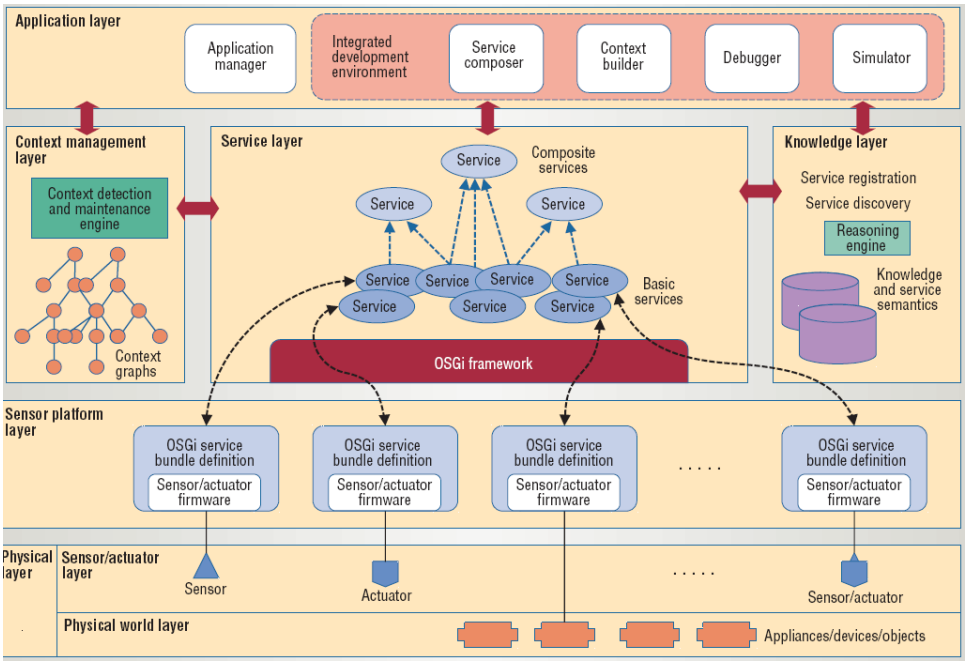


Figura 2.1: Arquitetura do projeto *Gator Tech* em camadas.

Foi utilizado neste projeto o *middleware OSGi* para a definição dos *drivers* responsáveis por integrar os sensores e atuadores junto camada de serviços (figura 2.1). Este projeto consiste em uma arquitetura em camadas, onde as aplicações interagem junto ao ambiente através de serviços disponibilizados pela arquitetura.

Fica sob a responsabilidade da mesma o controle sobre as relações de contexto e as decisões a serem tomadas.

2.3.4 UbiquitOS

O projeto *UbiquitOS* [24] [44], da Universidade de Brasília, possui seu foco no conceito de adaptabilidade de serviços. Seu objetivo é a decomposição dos dispositivos e recursos presentes no ambiente em serviços de tal maneira que as aplicações (ou o usuário) possam escolher qual melhor atenda melhor e de maneira transparente as suas necessidades.

Neste projeto foi desenvolvido um *middleware* homônimo que permite aos dispositivos a definição de *drivers* para seus recursos. Tais *drivers* são registrados em um servidor central que é acessado pelas aplicações a fim de encontrar os provedores adequados para os recursos necessários. O desenvolvimento foi realizado na plataforma *Java* [59], possuindo suporte tanto a plataforma mais robustas (utilizando *JSE*, *Java Standart Edition*) quanto para a plataforma limitadas (utilizando *JME*, *Java Micro Edition*). O *middleware* foi desenvolvido utilizando-se uma arquitetura em camadas, com destaque a camada de comunicação que utiliza o padrão *microkernel*. Isto permite ao *middleware* operar em redes de tecnologias distintas, dando suporte a comunicação utilizando *Bluetooth* [25] e *Ethernet* [30].

2.3.5 MediaBroker

O *MediaBroker* [51] é um projeto desenvolvido pelo *Georgia Institute of Technology* com foco no transporte eficiente de *streams* de dados. Neste projeto foi desenvolvido um *middleware* que permite a comunicação entre as aplicações através de "canais de dados". Estes canais de dados são "tipados" permitindo que aplicações que necessitem (*sinks*) de um tipo de dado encontrem fontes (*sources*) para o tipo desejado. Outra característica implementada é que podem ser associados aos tipos de dados disponíveis, algoritmos de conversão que permitem ao *middleware* disponibilizar fontes de dados "virtuais" que realizam a conversão dos dados a quem necessitar. Para acessar estas funcionalidades foi desenvolvida uma evolução do *framework D-Stampede* [52], permitindo o fácil acesso a estas características.

2.3.6 WSAMI

O *middleware WSAMI* [41] [33], desenvolvido pelo grupo de pesquisa *ARLES* (*Architecture Logicielles Et Systèmes distibués*) do *INRIA-France*, busca o estabelecimento de uma infra-estrutura de *web services* para plataformas móveis. Sua proposta utiliza a plataforma *SOA* (*Service Oriented Architecture* [27]) e se concentra em prover mecanismos de composição de serviços e a disponibilidade destes em redes de conectividade instável. Neste sentido foi realizada a implementação de um protocolo de descoberta distribuída de serviços, onde estes se encontram disponíveis utilizando *SOAP* [62] com *HTTP* [21]. Este *middleware* possui duas implementações disponíveis uma para *PCs* e outra para *Palm*. A utilização de *SOA* tem por objetivo facilitar o reuso, desenvolvimento e integração de serviços.

2.3.7 Mundo

O projeto *Mundo* [2] [3] [40], desenvolvido pela *Darmstadt University of Technology*, tem por objetivo o estudo no desenvolvimento de aplicativos para computação ubíqua. Os temas abordados pelo projeto abrangem temas como arquitetura de *software*, *middlewares*, *frameworks*, serviços comuns e processos de desenvolvimento [5]. Dentre estas pesquisas destaca-se o *middleware MundoCore*. Este *middleware* opera junto a aplicações permitindo que estas troquem informações entre si utilizando o conceito de objetos de serviços. Este se encontram disponíveis ao ambiente de maneira distribuída e são acessados através de rótulos conhecidos pelas aplicações. O *MundoCore* apresenta implementações em três plataformas distintas (*Java*, *C++* e *Phyton*) e os objetos podem ser disponibilizados ao ambiente utilizando duas formas de representação, *XML* e formato binário.

2.3.8 MoCA

O projeto *MoCA* (*Mobile Collaboration Architecture*) [10], desenvolvido pela Pontifícia Universidade Católica do Rio de Janeiro, possui seu foco na construção de aplicações sensíveis a contexto. Neste projeto foi desenvolvido um *middleware* homônimo que captura informações acerca da qualidade de sinal, localização, conectividade e carga de bateria dos dispositivos e os disponibiliza às aplicações para a tomada de decisões. Para a interação às aplicações utilizam uma comunicação orientada a eventos, de tal maneira que permite o controle das alterações do contexto em tempo real. O *middleware MoCA* notifica os eventos no ambiente em mensagem *XML* e provê implementações para a comunicação utilizando *Short Message Service (SMS), User Datagram Protocol (UDP), Transmission Control Protocol (TCP), Java Messaging Service (JMS),* ou *Wireless Application Protocol (WAP).*

2.3.9 EasyLiving: InConcert

O projeto *EasyLiving* [57], desenvolvido pelo *Vision Group* da *Microsoft Research*, visa fornecer ao usuário uma experiência mais transparente e menos invasiva da tecnologia. Neste contexto o projeto apresenta o *InConcert*, um *middleware* de suporte ao desenvolvimento de aplicativos e serviços. De maneira similar ao projeto *Gator Tech*, no *InConcert*, os serviços são interfaces de acesso aos dispositivos presentes no ambiente, fornecendo aos aplicativos um meio simples de acesso a estes recursos. Para acessar estas funcionalidades, os dispositivos devem possuir poder computacional e conectividade ao ambiente ao qual estão inseridos. Foi desenvolvido ao longo do projeto um conjunto de aplicações que permitem ao usuário ajustar sua interação com o computador de maneira mais fluída, permitindo (por exemplo) a troca de computador sem impacto nas tarefas em execução.

2.3.10 Home SOA

O projeto *Home SOA* apresenta uma plataforma de serviços para ambientes residenciais. Sua arquitetura propõe o uso de *drivers* de serviços a fim de isolar as

diferenças entre as diversas plataformas de rede e protocolos disponíveis no ambiente. Sua implementação utiliza o *middleware OSGi* para a definição de *drivers*. O uso do *OSGi* permite a integração dinâmica de novos *drivers* ao ambiente de maneira transparente às aplicações em execução.

2.3.11 Comparativo

Observando os projetos apresentados podemos listar um conjunto de características que valem ser observadas neste trabalho. Cada projeto aborda a questão do desenvolvimento na *ubicomp* com objetivos distintos, porém todos com o foco em facilitar o acesso a estas aplicações aos recursos do ambiente. Dentre estas abordagens vemos a aplicação de *middlewares* já existentes em novas arquiteturas bem como o desenvolvimento de novos e a elaboração de protocolos para a comunicação no *smart space*. Dentre as diversas características de cada projeto vamos analisar apenas quatro aspectos distintos: a forma como o projeto modela seu ambiente, quais plataformas são almejadas pelo projeto, qual o arquitetura seguida pelos dispositivos e como estes se comunicam entre si.

2.3.11.1 Visão do Ambiente

A visão do ambiente corresponde a como são modeladas as funcionalidades que estarão disponíveis as aplicações. Esta característica reflete diretamente em como cada iniciativa aborda a questão de desenvolver aplicativos para *ubicomp*. Dentre os projetos estudados podemos enxergar alguns tipos básicos de modelagem seguida. Estas abordagens podem ser vistas na tabela 2.1 e explicitadas a seguir:

Projeto	Visão	
Aura	Aplicativos	
Gaia		Recursos
Gator Tech		Drivers
Home SOA		Drivers
Media Broker		Streams
UbiquitOS	Serviços	Serviços Síncronos
WSAMI		Serviços Síncronos
Mundo		Objetos
MoCA		Eventos
InConcert		Serviços Síncronos

Tabela 2.1: Visão do ambiente nos *middlewares*.

- A visão do ambiente como um conjunto de *aplicativos* é vista no projeto *Aura*. Neste caso o ambiente é visto como um conjunto de aplicativos que possuem um grau de equivalência entre si de tal forma que permite ao ambiente a tomada de decisão de como e quando migrar uma tarefa de um aplicativo a outro.

- A utilização de *serviços* como representação das funcionalidades do ambiente é a abordagem mais comum entre as soluções estudadas. Dentro desta visão podemos destacar algumas especificidades.
  - Projetos como *Gaia*, *Gator Tech* e *Home SOA* permitem o acesso aos serviços através de agrupamentos destas funcionalidades denominados como recursos (*GaiaOS*) ou *drivers* (*Gator Tech*, *Home SOA*). Este tipo de abstração permite o acesso às funcionalidades de maneira coesa.
  - O projeto *MediaBroker* apresenta uma abordagem onde os serviços são acessados através de *streams de dados*. Esta abordagem permite o acesso a dados contínuos, adequados para o tráfego de grandes volumes de dados (arquivos) ou fluxos de áudio e vídeo.
  - O acesso aos serviços de maneira *síncrona* (*UbiquitOS*, *WSAMI*, *InConcert*) permite as aplicações um acesso mais próximo do modelo funcional de programação. Fortemente adequado para solicitações de dados e comandos de ação.
  - As *notificações de eventos* (*MoCA*) permitem às aplicações observar as alterações no ambiente em tempo real.
  - A abordagem do *middleware MundoCore* é disponibilizar ao ambiente serviços através de *objetos* rotulados. Esta abordagem permite a troca de informações através da alteração no estado destes objetos.

2.3.11.2 Plataformas

Dependendo da aplicação e do tipo de ambiente almejado por cada projeto, as plataformas dos dispositivos considerados são distintas. Alguns projetos possuem foco em dispositivos móveis, outros em ambientes ricos em atuadores e sensores. Este tipo de característica afeta quais dispositivos podem ser diretamente integrados ao ambiente ou que tipo de acesso estes terão. Com relação a esta característica foram considerados apenas aqueles que são o foco direto dos projetos tendo acesso direto aos recursos das plataformas desenvolvidas. As plataformas foram analisadas de acordo com os seguintes critérios:

- Plataforma de hardware: Aqui foram consideradas as características do porte dos dispositivos suportados pelo projeto. Dentro desta característica foram consideradas três gradações distintas:
  - Plataformas robustas: Aqui temos dispositivos que possuem alto poder de processamento e gasto energético como *PCs* e *laptops*.
  - Plataformas intermediárias: Consideramos nesta gradação dispositivos que se encontram entre o nível robusto e limitado como *palms* e *smartfones*.
  - Plataformas limitadas: Dispositivos dotados de limitações diversas (processamento, bateria, memória, conectividade). Inclui-se aqui grande parte dos dispositivos móveis (*soft-fones*, relógios, jóias) além de sensores e atuadores.

- Plataforma de desenvolvimento: Aqui analisamos as plataformas de desenvolvimento suportadas por cada projeto.

Projeto	Plataformas robustas	Plataformas intermediárias	Plataformas limitadas	Plataformas de desenvolvimento
Aura	Sim	Sim	Não	*
Gaia	Sim	Sim	Não	Lua
Gator Tech	Sim	Não	Não	Java
Home SOA	Sim	Não	Não	Java
Media Broker	Sim	Não	Não	C++
UbiquitOS	Sim	Sim	Sim	Java
WSAMI	Sim	Sim	Não	Java, C++
Mundo	Sim	Sim	Sim	Java, C++, Phyton
MoCA	Sim	Não	Não	Java
InConcert	Sim	Não	Não	*
* Não foram encontradas informações sobre as plataformas utilizadas.				

Tabela 2.2: Plataformas nos *middlewares*.

A tabela 2.2 apresenta os dados dos projetos analisados com relação as plataformas consideradas. Notamos duas características interessantes neste quadro. Primeiramente vemos que poucos projetos possuem foco em disponibilizar acesso direto ao ambiente para dispositivos limitados. O mesmo é visto por muitos projetos como uma questão a ser tratada através de dispositivos (mais robustos) que realizem o intermédio destas interações. Outra característica que observamos é a predominância da linguagem *Java* no desenvolvimento dos projetos em conjunto com a presença de projetos com suporte a múltiplas plataformas de desenvolvimento.

2.3.11.3 Arquitetura

A definição de um *smart space* leva em consideração o tipo de aplicações e usuários que freqüentarão aquele ambiente. Na definição de como os dispositivos estarão agrupados podemos seguir uma abordagem *centralizada*, onde todas as informações estão concentradas em um nó central (ou um grupo destes) ou de maneira *distribuída*, onde cada dispositivo possui parte das informações acerca do ambiente. A tabela 2.3 mostra como estas abordagem foram realizadas em cada projeto. Podemos notar que temos uma presença bem dividida de ambas as alternativas nos projetos estudados. Isto está diretamente ligado a como cada projeto trata os pontos fortes e fracos de cada abordagem de acordo com as aplicações alvo.

2.3.11.4 Comunicação

As redes de comunicação desempenham um papel crucial nos ambientes inteligentes. Os dispositivos necessitam destes meios para coordenar suas ações e compartilhar

Projeto	Arquitetura
Aura	Centralizado
Gaia	Centralizado
Gator Tech	Centralizado
Home SOA	Centralizado
Media Broker	Distribuído
UbiquitOS	Centralizado
WSAMI	Distribuído
Mundo	Distribuído
MoCA	Distribuído
InConcert	Centralizado

Tabela 2.3: Arquiteturas nos *middlewares*.

suas informações. Cada projeto utiliza-se de uma ou mais tecnologias de comunicação para atingir esta tarefa, porém independente do meio utilizado cada abordagem utiliza-se de um padrão distinto para representar suas mensagens.

Projeto	Comunicação
Aura	Prórpio
Gaia	RPC/Corba
Gator Tech	OSGi
Home SOA	OSGi
Media Broker	Binário
UbiquitOS	JINI/RMI
WSAMI	SOAP/HTTP
Mundo	XML, Binário
MoCA	XML
InConcert	XML

Tabela 2.4: Comunicação nos *middlewares*.

Na tabela 2.4 encontramos a referência das representações utilizadas por cada projeto. Notamos aqui forte presença de interfaces de comunicação providas pelas plataformas de desenvolvimento (*Gaia*, *Gator Tech*, *Home SOA* e *UbiquitOS*). Esta abordagem simplifica o esforço de desenvolvimento nestas plataformas, porém limita a evolução dentro destes modelos. Outra presença forte é da utilização de *XML* para representar os dados comunicados. O uso de *XML* simplifica abordagens de checagem de interfaces e de formatos de dados além de possibilitar a fácil adaptação a mudanças.

2.4 Desenvolvendo aplicativos para computação ubíqua

Observados os requisitos apresentados em [64], os desafios listados por [15] e as características encontradas nos projetos analisados, chegamos nos três pontos que

serão observados neste trabalho. Estes pontos foram previamente apresentados no capítulo 1 e serão melhor detalhados a seguir:

2.4.1 Suporte a dispositivos limitados

Poucos dos projetos analisados apresentam foco na integração direta de dispositivos limitados. Porém a presença deste tipo de equipamento nos ambientes é uma tendência crescente. A integração destes de maneira direta aos sistemas de apoio é um passo necessário para que características como mobilidade sejam tratadas de forma espontânea no *smart space*. Desta forma o sistema de suporte deve considerar limitações como:

- Processamento: Esta limitação leva os sistemas de apoio a demandarem pouco esforço computacional deste tipo de dispositivo para se manterem no ambiente. Para tal, deve-se optar por protocolos de comunicação leves, algoritmos de segurança e confiabilidade de baixo custo computacional.
- Bateria: Um dispositivo com baixa carga energética deve poupá-la para suas funcionalidades principais não podendo desprender sua energia com o tráfego desnecessário de informações ou estando alerta por longos períodos de tempo.
- Memória: Baixa quantidade de memória demanda protocolos que onerem pouco os dados trafegados e algoritmos com baixa complexidade em memória.
- Largura de banda: A utilização de redes de comunicação intermitentes ou de baixa potência fornece aos dispositivos uma conectividade igualmente limitada. Para tal, redundâncias e mensagens pesadas devem ser descartadas ou utilizadas com cautela.

2.4.2 Tratamento da comunicação

Vimos que cada projeto aborda as funcionalidades do ambiente de maneiras diversas. Dentro do tratamento das funcionalidades podemos observar que a forma como é realizada as interações no *smart space* são bastante variadas. Dentre o que foi observado podemos listar dois aspectos distintos destas comunicações:

- Notamos que o acesso aos recursos do ambiente pode ser realizado de duas maneiras. Seja de forma *síncrona*, utilizado para consultas e comandos, ou de maneira *assíncrona*, utilizada na notificação de eventos e alterações no ambiente.
- O tráfego de informações nestas funcionalidades se apresenta tanto de maneira *discreta* e estruturada como em fluxos de dados *contínuos*.

Estas questões devem ser abordadas por um sistema de apoio, pois aplicações podem operar nas diversas combinações que estas características possibilitam.

2.4.3 Heterogeneidade

A integração de uma vasta quantidade de dispositivos implica em lidar com as diferenças que estes trazem consigo. Tais diferenças abordam desde as suas diferenças de *hardware* e tecnologias de comunicação, até suas plataformas de *software*. Com relação às plataformas de *hardware* e *software*, esta questão pode ser tratada utilizando-se padrões abertos e facilmente suscetíveis a mudanças e evoluções, permitindo a sua adaptação a novos dispositivos. Com relação às tecnologias de comunicação, um mesmo ambiente pode possuir diversos equipamentos operando em redes distintas e ainda assim fazer parte do mesmo *smart space*. Sendo assim, a comunicação entre estes deve ser levada em consideração.

2.5 Resumo do capítulo

Neste capítulo foi exposta a nossa visão com relação à computação ubíqua cujo principal objetivo é a construção de ambientes que auxiliem o usuário em suas tarefas de maneira invisível. Nesta tarefa é de crucial importância o papel desempenhado pelas aplicações bem como a capacidade de interagir e conhecer os recursos disponíveis no ambiente. A construção destas apresenta diversos desafios dos quais três foram escolhidos como foco deste trabalho: heterogeneidade suporte a dispositivos limitados e tratamento dos detalhes de comunicação. Apresentou-se um conjunto de soluções para o auxílio na construção destas aplicações onde ficou exposto que nenhuma endereça de maneira completa estes três requisitos simultaneamente.

Capítulo 3

SOA - Service Oriented Architecture

Conforme definido em [28], a Arquitetura Orientada a Serviços (*SOA*), proposta por *Roy W. Schulte* em 1996 [42], é um "paradigma para organização e utilização de capacidades distribuídas que possam se encontrar sob o controle de diferentes domínios e propriedades" .

A proposta desenvolvida na *SOA* está na componentização e reuso de funcionalidade de softwares. Para compreender a proposta tomemos por base o seguinte cenário:

> *Uma organização possui um parque tecnológico dotado de diversas soluções de software distintas que abrangem as diversas áreas que a compõem. Surge então a necessidade de criação de uma nova solução de software por parte de uma destas áreas. Esta nova solução pode ser fragmentada em diversas capacidades (requisitos) específicas. Parte destas funcionalidades podem já ser atendidas parcial ou completamente por uma ou mais funcionalidades presentes nas demais soluções em produção nesta organização. Como evitar a replicação destas funcionalidades?*

Neste cenário temos um problema tradicional de reuso de *software*. Uma possível solução seria a divisão do *software* em módulos. Desta forma os módulos responsáveis por implementar as funcionalidades comuns a outras soluções podem ser empacotados, distribuídos e integrados a estas. Porém esta solução introduz algumas restrições que podem apresentar problemas para a integração e manutenção desta abordagem. Em primeiro lugar, com relação a distribuição de atualizações nos módulos reutilizados. A cada alteração ou atualização destes módulos, estes devem ser novamente empacotados, distribuídos e integrados às soluções clientes. Caso as interfaces ou premissas destes módulos sejam alteradas, um esforço de adaptação das soluções clientes será necessário. Outra questão diz respeito à limitação imposta por esta solução com relação à plataforma tecnológica a ser utilizada. Caso tenhamos soluções em plataformas distintas não será possível a estas usufruir das soluções distribuídas.

3.1 Serviços

A proposta da *SOA* é de que cada *software* atenda apenas a seus requisitos específicos, de forma que os requisitos que venham a ser necessários por outras aplicações sejam compartilhados com estas na forma de serviços. Os serviços servem como um mecanismo pelo qual as necessidades e capacidades das aplicações são associadas de maneira colaborativa. O conceito de *serviço* definido na *SOA* se encontra a seguir:

> *Serviço* é um mecanismo que disponibiliza uma ou mais funcionalidades, onde o acesso é provido através de uma interface definida entre as partes (quem utiliza e quem provê o serviço).

Desta forma cada serviço está acessível às aplicações clientes através de um canal de comunicação, evitando a necessidade de se controlar a distribuição e manutenção de alterações nas funcionalidades disponibilizadas pelos serviços. Apesar de não ser um conceito definido pela *SOA*, o baixo acoplamento entre os serviços é tido como um objetivo, porém é de difícil definição devido a suas características subjetivas [28]. Este baixo acoplamento se dará através da definição das interfaces e meios de acesso aos serviços definidos.

A forma como um serviço implementa suas funcionalidades é transparente à aplicação cliente. Porém a execução de um serviço deve provocar efeitos que possam ser observados externamente. Estes efeitos podem ser:

1. Uma informação é retornada como forma de resposta da solicitação do serviço.
2. Ocorre uma mudança de estado em alguma entidade visível externamente (compartilhada).
3. Alguma combinação dos itens 1 e 2.

3.2 Os Papéis

No ecossistema de entidades que compõem o *SOA*, onde temos entidades diversas interagindo através de seus serviços, podemos definir três papéis distintos de acordo com a forma como cada qual contribui com o processo de interação. De acordo com este ponto de vista são definidos os seguintes papéis.

- O *Consumidor (Consumer)* é a parte interessada em acessar as funcionalidades providas por um determinado serviço (ou conjunto destes) para a conclusão de uma determinada tarefa.
- O *Provedor (Provider)* é responsável por prover uma implementação das funcionalidades desejadas pelos *Consumidores* na forma de serviços e disponibilizá-los através de interfaces públicas comuns entre as partes.

- O *Registro (Broker)* agrega as informações sobre os *Provedores*, e seus serviços, presentes no ambiente e fornece estas informações aos *Consumidores*.

Não existem restrições a uma entidade desempenhar diversos papéis simultaneamente. Assim uma mesma entidade pode ser consumidora de determinados serviços e disponibilizar outros serviços as demais entidades. Inclusive uma mesma entidade pode acessar um conjunto de serviços e utilizá-los para prover outro conjunto de serviços, o que é conhecido como *composição de serviços*.

Estes três papéis podem interagir de duas formas distintas, sendo estas diferenciadas pela presença ou ausência de um *registro* no contexto. Estas duas situações estão expostas nas figuras 3.1 e 3.2.

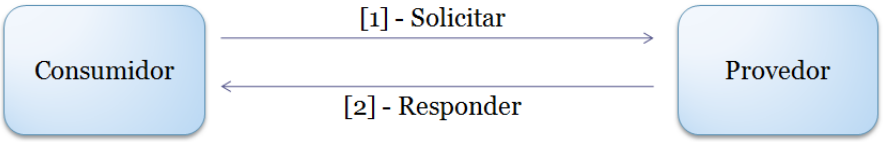


Figura 3.1: Interação *SOA* entre um Provedor e um Consumidor

A existência de um *registro* não é obrigatória na arquitetura *SOA*. Porém, sem a presença de uma entidade que cumpra este papel no ambiente, faz-se necessário que o consumidor conheça os provedores dos serviços que necessita. Este tipo de interação é a mais simples e é constituída pelos passos descritos abaixo (conforme ilustrado na figura 3.1).

1. O *consumidor* solicita ao *provedor* a execução de um determinado serviço. Neste passo é necessário que o *consumidor* informe ao *provedor* os parâmetros de execução para o serviço de acordo com a interface definida por este.

2. O *provedor* executa as ações associadas ao serviço solicitado e retorna ao *consumidor*, caso necessário, com as informações de acordo com a interface do serviço.

No caso de existir uma entidade que cumpra o papel de *registro*, não se faz necessário o conhecimento prévio dos *provedores* por parte do *consumidor*. Desta maneira, para o *consumidor* tomar conhecimento dos *provedores* dos serviços necessários deve-se executar uma consulta ao *registro* antes da solicitação da execução do serviço conforme descrito anteriormente (conforme ilustrado na figura 3.2).

1. O *provedor* cadastra os seus serviços junto ao *registro*.

2. O *consumidor*, ao necessitar de um serviço, e desconhecendo quem provê tal serviço, solicita as informações de quem possui este serviço ao *registro*.

3. O *registro* retorna ao *consumidor* com a lista dos *provedores* do serviço em questão.

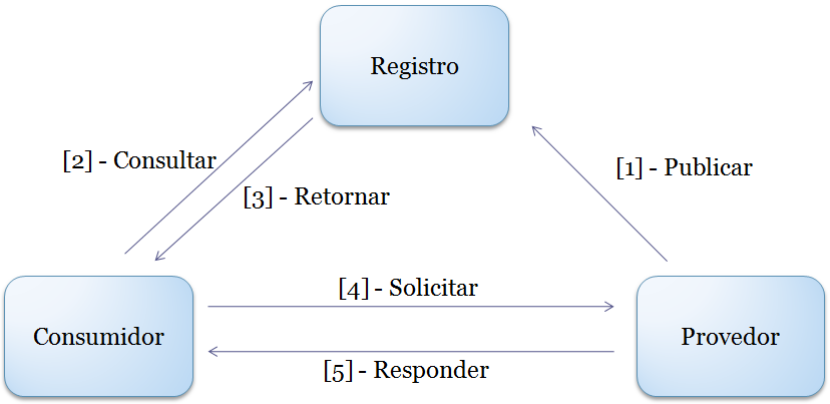


Figura 3.2: Interação *SOA* entre um Provedor e um Consumidor com a mediação de um Registro

4. De posse das informações de quem provê o serviço, o *consumidor* solicita a execução do serviço ao *provedor* escolhido. Neste passo é necessário que o *consumidor* informe ao *provedor* os parâmetros do serviço de acordo com a interface definida para este.

5. Em seguida, o *provedor* executa as ações associadas ao serviço solicitado e retorna ao *consumidor*, caso necessário, com as informações estabelecidas na interface do serviço.

3.3 Dinâmica dos Serviços

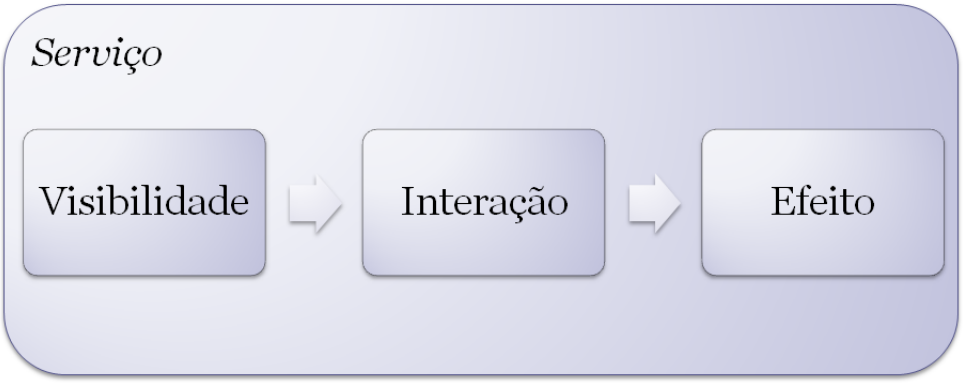


Figura 3.3: Conceitos envolvidos na interação dos serviços *SOA*.

Para compreender como ocorre a interação entre as entidades desempenhando os papéis definidos na *SOA* através dos serviços é necessário compreender três conceitos básicos (representados na figura 3.3). O conceito de *visibilidade* diz respeito a como estas entidades estabelecem uma noção dos serviços disponibilizados. A *interação* está ligada a como estas realizam a comunicação e execução

destes serviços. Por fim, o conceito de *efeito* indica como estes serviços afetam o ambiente no qual estão inseridos.

3.3.1 Visibilidade

Para que as entidades envolvidas em um ambiente *SOA* possam interagir faz-se necessário que exista uma relação de visibilidade entre elas. Sem este requisito mínimo não é possível que ocorra a interação entre estas partes, e assim a relação de consumo de um serviço não pode ser concretizada. Para se estabelecer o conceito de visibilidade entre duas entidades são necessários três requisitos:

- Uma entidade deve estar *ciente* da existência da outra parte. O desconhecimento das partes é impeditivo para que estas iniciem uma comunicação, e portanto este conhecimento deve existir antes que quaisquer interação ocorra. Como visto anteriormente, um consumidor pode tanto conhecer seus provedores *a priori* ou pode buscar esta informação no ambiente através de outra entidade (cumprindo o papel de registro).

- As entidades devem possuir a *intenção* de participar da comunicação. Caso uma das partes não a possua, a execução do serviço será abortada antes de seu início. Observe que em muitos casos a falta de intenção de participar da interação em um serviço pode ser esperada, como no caso de sobrecarga do provedor deste serviço.

- Uma entidade deve ser capaz de *alcançar* a outra entidade afim de estabelecer um canal de comunicação. Sem um meio por onde trafegar os dados acerca do serviço a ser prestado, não existe como tomar conhecimento da necessidade de execução deste.

3.3.2 Interação

Não bastam que as entidades possuam uma visibilidade entre si, estas devem ser capazes de interagir a fim de comunicar a intenção de execução de um serviço e receber o retorno desta (caso necessário). Para que isto seja alcançado, as partes envolvidas devem concordar acerca dos meios que serão utilizados para estabelecer esta comunicação. Deve-se então estabelecer um modelo de comunicação entre as entidades de maneira que estas concordem com relação ao formato (*sintaxe*) e significado (*semântica*) das mensagens [1] por elas trocadas. Neste modelo devem ser estabelecidos como devem se *comportar* as partes de acordo com a ordem e o tipo de mensagem trocada entre elas. Por fim, faz-se necessário que as partes concordem em como os serviços devem ser *representados*, formando assim um entendimento comum do significado destes. A soma destes fatores (sintaxe, semântica, comportamento e representação de serviços) estabelece uma interface

[1] A *SOA* não estabelece um modelo de comunicação entre as entidades. Podendo assim serem utilizados diversos modelos de comunicação, como eventos, memória compartilhada, dentre outros. Este modelo é utilizado aqui apenas de forma ilustrativa e não remove a possibilidade dos exemplos apresentados serem implementados de outra forma.

comum de comunicação entre as entidades. Esta interface então permite a interação não ambígua entre as partes.

3.3.3 Efeito

Espera-se que a interação entre duas entidades através de um serviço leve a um efeito externamente observável. Um serviço que não possui um efeito que possa ser experimentado por outras entidades presentes no ambiente, não possui um propósito claro. Neste caso sua execução não afeta os demais e portanto sua existência no ambiente não agrega valor ao mesmo. A execução de um serviço deve proporcionar algum fluxo de dados ou alteração de estado no ambiente, seja pela troca de informações entre o provedor e o consumidor, ou através da alteração do estado em algum espaço compartilhado entre as entidades. Esta alteração de estado, ou troca de informação (efeito) provocada pelo serviço deve ser esperada pelas entidades participantes do processo de execução do serviço (de acordo com a interface do mesmo). O entendimento do efeito de um serviço não deve levar a parte consumidora ao conhecimento das ações executadas pela parte provedora. Os detalhes para se prover um serviço devem ser transparentes ao consumidor, deixando este apenas ciente do resultado final apresentado.

3.4 Mapeamento de Domínio

Da maneira como foi apresentada, a *SOA* é apenas uma arquitetura para o desenvolvimento e integração de aplicações e soluções de *software*. E como uma arquitetura por si só, não é uma solução de *software*. Para que esta possa ser concretizada o primeiro passo é mapear os conceitos da SOA de acordo com a realidade de cada contexto em que se pretende implantá-la. Cada conceito deve ser analisado e instanciado de acordo com o negócio envolto no problema a ser resolvido. A partir de então, é possível mapear esta instância da SOA em uma solução de *software* a ser implementada e implantada.

3.4.1 Web Services

Um exemplo de instância da *SOA* é a arquitetura de *WebServices* [43]. Seu objetivo é prover a integração de aplicações através da *web* baseando-se no protocolo *HTTP* e no uso do formato *XML*. Atualmente a quantidade de aplicações desenvolvida na *web* dentro das organizações vem crescendo a cada ano. Por muitas vezes estas aplicações necessitam de compartilhar parte de suas funcionalidades entre si e o custo de se desenvolver protocolos de comunicação à parte das aplicações não se mostra interessante. A arquitetura de *WebServices* propõe o compartilhamento destas funcionalidades através da rede utilizando o protocolo *HTTP*, nativo a estas aplicações, para realizar a comunicação entre as partes. Para representar estes serviços é utilizado como protocolo de transporte das mensagens o *XML*.

Um dos objetivos a serem alcançados pela arquitetura de *WebServices* na utilização do *XML* como forma de representação de seus serviços e suas mensagens

é o desacoplamento da implementação dos serviços e as aplicações clientes. A arquitetura define os mesmos três papéis da *SOA* para as aplicações que participam da arquitetura, bem como que estas podem acumular e desempenhar vários papéis concomitantemente.

Os serviços são representados utilizando a linguagem *XML* sendo a representação mais comum a denominada *WSDL* (*Web Services Description Language*) responsável por estabelecer um comum entendimento das aplicações com relação as interfaces dos serviços. Esta representação define características como:

- **Identificador do serviço**
- **Parâmetros de chamada do serviço**
- **Parâmetros de retorno do serviço**
- **Formato dos parâmetros a serem trafegados**

Nesta arquitetura os serviços são acessados pelo uso de mensagens discretas através de uma interação síncrona do tipo requisição/resposta (*request/response*). Desta maneira uma aplicação que necessita de um serviço deve enviar uma mensagem de requisição deste serviço à aplicação provedora do mesmo. Em seguida a aplicação provedora deve tratar a requisição e posteriormente retornar uma mensagem de resposta da solicitação realizada. Para o acesso e solicitação de serviços dentro da arquitetura de *WebServices* temos dois protocolos mais proeminentes: o *SOAP* e o *REST*.

O **SOAP**, acrônimo para *Simple Object Access Protocol* [62], é um protocolo completamente baseado em mensagens *XML*. Neste protocolo temos a definição das "*tags*" *XML*, responsáveis por representar as mensagens de requisição e resposta dos serviços. Também estão definidas "*tags*" que representam os parâmetros de chamada e retorno dos serviços no protocolo.

O **REST** (*REpresentational State Transfer*) é um protocolo que se baseia no mapeamento dos serviços em *URL*'s da aplicação bem como nos métodos nativos do protocolo *HTTP*. Este mapeamento direto da chamada e retorno dos serviços, torna a implementação e execução de seus serviços mais simples e eficiente que o uso do *SOAP*. Em contrapartida torna o entendimento dos serviços mais complexas devido à necessidade de se compreender os mapeamentos realizados. Assim como no *SOAP*, o *REST* define "*tags*" *XML* para a representação dos parâmetros de chamada e retorno dos serviços no protocolo.

3.4.2 Aplicações em ambientes de computação ubíqua

A utilização de *SOA* parece natural, visto que o contexto de compartilhamento de serviços é comum a ambas. Um exemplo de aplicação que utiliza os conceitos desta arquitetura é o *middleware WSAMI*, desenvolvido pelo grupo de pesquisa francês *ARLES*. O *WSAMI* disponibiliza os serviços das aplicações presentes no *smart space* através de especificações *WSDL*. A comunicação destes serviços é feita através do protocolo *SOAP* e a descoberta de serviços é feita de maneira distribuída e colaborativa entre as aplicações (e dispositivos) presentes no ambiente.

3.5 Resumo do capítulo

Neste capítulo apresentamos em maiores detalhes a arquitetura orientada a serviços. Suas definições, papéis e princípios estabelecem uma base para orquestrar a interação entre componentes de software em uma realidade distribuída. Como uma arquitetura, a *SOA* apresenta uma maneira de se enxergar este problema, necessitando assim de ser instanciada a cada realidade que se apresenta. Neste sentido foram expressos os *Web Services* como uma instância da *SOA*. Por fim, foram discutidas a utilização da SOA em ambientes de computação ubíqua.

Capítulo 4

Proposta

Observando a realidade do ambiente inteligente fica claro que as aplicações são responsáveis pela tarefa de coordenar os dispositivos e fornecer auxílio aos usuários. O desenvolvimento destas soluções envolve diversos requisitos dentre os quais três foram destacados. Tratar cada desafio de maneira individual pelas aplicações não é a melhor abordagem e por isso a utilização de *middlewares* se mostra a mais utilizada.

As arquiteturas orientadas a serviços apresentam diversas características que auxiliam em ambientes inteligentes. Sua abordagem proporciona o reuso de capacidades em ambientes distribuídos de *software*, bastante característico dos *smart spaces*. Porém, a *SOA* não endereça as características específicas de modelagem de um ambiente inteligente, limitando-se a descrever serviços de alto nível e componentes genéricos. Além disto, a arquitetura não propõe um modelo de comunicação a ser seguido, deixando de lado os detalhes de interação característicos da *ubicomp*.

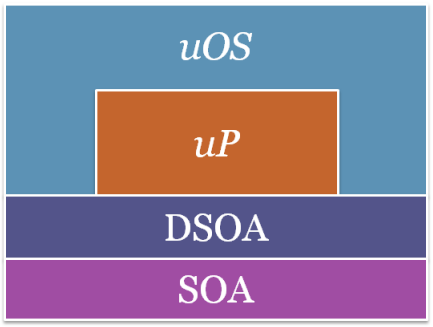


Figura 4.1: Representação da solução apresentada neste trabalho.

Observando estas questões, este trabalho propõe um conjunto de soluções elaborado em três partes (figura 4.1). Primeiramente temos a *DSOA* (*Device Service Oriented Architecture*), um extensão da SOA com o intuito de complementar duas características importantes de ambientes inteligentes:(i) a forma de modelar o ambiente deve favorecer um acesso coeso as capacidades disponíveis; (ii) as formas de interações possíveis no *smart space* são definidas de acordo com os requisitos da *ubicomp*. Com base na *DSOA* foi definido o conjunto de protocolos *uP* (*ubiquitous*

Protocols) com o intuito de fornecer uma interface padrão de comunicação entre os dispositivos no ambiente. Por fim é apresentado o *middleware uOS* (*ubiquitous OS*) que provê mecanismos computacionais para se desenvolverem os principais componentes de software de um ambiente, as aplicações e os *drivers* de recursos.

A arquitetura *DSOA*, seus conceitos e estratégias são apresentados na sessão 4.1. Os protocolos que compõem o *uP* e seus formatos de mensagens são definidos na sessão 4.2. A estrutura de componentes fornecida pelo *middleware uOS* e como são integradas novas instâncias a este é apresentada na sessão 4.3.

4.1 DSOA - Device Service Oriented Architecture

A realidade de um ambiente inteligente é constituída pela presença de uma grande variedade de dispositivos e aplicações, cuja interaçãogira em torno de um objetivo comum. Conforme exposto em [64], este objetivo consiste em auxiliar o usuário em suas tarefas da maneira menos intrusiva possível. Para alcançar este objetivo, as aplicações devem conhecer os recursos presentes no ambiente. Suas decisões serão baseadas nas informações providas por estes, e suas ações poderão fazer uso dos serviços disponibilizados pelos recursos. Além disto, as aplicações podem trocar informações entre si na composição de serviços a serem realizadas em prol do usuário. Para ilustrar esta situação, consideremos o seguinte cenário:

> Um *smart space* é constituído por uma sala de estar. Esta sala conta com recursos de câmeras, sensores de movimento e um televisor dotado de um bom sistema de áudio e vídeo. Todos estes recursos estão integrados através de uma rede de comunicação. Uma aplicação de personalização do ambiente está ligada aos recursos presentes no ambiente. Seu objetivo é adaptar o ambiente de acordo com as características do usuário, tomando ações que visem antecipar suas necessidades. Considere a seguinte seqüência de acontecimentos:
>
> 1. Um usuário, ao entrar no ambiente, é detectado pelo sensor de movimento.
> 2. O sensor notifica uma aplicação a cerca da presença do usuário no ambiente.
> 3. Ao receber a notificação, a aplicação solicita à câmera que capture a imagem do usuário.
> 4. De posse da imagem do usuário, a aplicação busca em sua base de dados as informações a seu respeito e o identifica como sendo "Lucas".
> 5. O perfil de Lucas diz que ele é um grande apreciador de música e que ele tem o hábito de ouvi-las em seu celular quando fora de casa. A aplicação então solicita ao celular de Lucas as informações sobre as músicas que ele estava ouvindo antes de entrar na sala.

6. De posse destas informações, o ambiente define que a televisão é o equipamento mais adequado para a apreciação destas músicas.
7. A aplicação transfere então as músicas do celular de Lucas para a televisão e continua a execução da lista de reprodução neste aparelho.

Para Lucas estas ações ocorreram de maneira transparente, de forma que suas atividades prosseguiram sem interrupções. O ambiente foi responsável por realizar a aquisição de informações e com base nelas e nas características conhecidas sobre Lucas tomou ações a fim de antever suas necessidades. Porém, para que tudo ocorresse foram necessárias diversas interações entre os dispositivos presentes no ambiente e a aplicação, incluindo dispositivos móveis que foram introduzidos no ambiente, no caso o celular de Lucas.

Para se orquestrar interações em um *smart space*, vemos que as aplicações devem possuir a capacidade de conhecer o ambiente e os recursos disponíveis de maneira dinâmica. Os dispositivos móveis introduzem novos recursos no ambiente, e as aplicações ubíquas devem ter ciência destas alterações e recofigurar-se de acordo com elas.

É neste sentido que as idéias apresentadas pela arquitetura *SOA* no capítulo 3 vem de encontro a este cenário apresentado. O que se busca no contexto de computação ubíqua são formas de integrar estes recursos (e os serviços relacionados a eles) e as aplicações presentes no ambiente de maneira dinâmica e colaborativa. A *DSOA* (*Device Service Oriented Architecture*) [11] tem por objetivo estabelecer uma arquitetura que una os conceitos apresentados na *SOA* e o cenário distribuído das aplicações e recursos da computação ubíqua. O papel desta arquitetura é definir, expandir e instanciar estes conceitos tendo em vista as necessidades e detalhes envolvidos na colaboração entre dispositivos. Observando assim critérios como as distinções entre as diversas plataformas e os detalhes nas interações entre os dispositivos.

4.1.1 Conceitos

4.1.1.1 O ambiente inteligente

O conceito de ambiente inteligente remete à aplicação de algum tipo de inteligência artificial. Porém, aqui este conceito visa restringir o escopo do ambiente a ser analisado pela arquitetura. Na *DSOA*, o ambiente inteligente é definido conforme segue:

> O ambiente inteligente (ou *smart space*) é composto por um conjunto de dois ou mais dispositivos dotados de poder computacional e conectados através de uma rede de comunicação de maneira colaborativa junto aos usuários do ambiente.

Nesta definição primeiramente devemos observar a cardinalidade envolvida. Um ambiente composto por apenas um único dispositivo não apresenta a diversidade que caracteriza um ambiente inteligente. É da composição de vários

dispositivos e usuários que a realidade vista pela *ubicomp* toma forma. É da interação entre estas diversas entidades que surgem as condições de provimento de serviços aos usuários, de forma a apoiar a realização de suas ações. São destas mesmas interações que ocorrem os diversos desafios envolvidos neste tipo de ambiente. Por tal razão se necessita agregar e coordenar os diversos dispositivos pulverizados no ambiente. Além disto, a volatilidade característica dos ambientes ubíquos torna a interação entre aplicativos, recursos e usuários mais complexa de se gerenciar.

As entidades que podem integrar um ambiente ubíquo são quaisquer dispositivos com poder de processamento e comunicação. Com relação ao poder computacional, não existe uma determinação ou restrição às características do *hardware*, ficando esta a cargo das aplicações a serem executadas e o contexto do ambiente em questão. O mesmo se aplica a capacidade de comunicação, que não é limitada pela arquitetura *DSOA* com relação às formas e tecnologias de comunicação a serem utilizadas.

Figura 4.2: Um exemplo de ambiente inteligente como uma sala estar.

Por fim temos que estes dispositivos devem se integrar de maneira colaborativa. Por colaboração entende-se que estes dispositivos devem ser capazes de disponibilizar recursos (e seus serviços) ao ambiente, bem como hospedar as aplicações que serão responsáveis por interagir junto ao usuário. A figura 4.2 mostra a sala inteligente apresentada no exemplo utilizado neste capítulo. Podemos identificar os seguintes dispositivos neste ambiente:

1. **Ar condicionado**: O ar condicionado pode ser utilizado para funções como alterar a temperatura, ajustar a ventilação ou consultar a temperatura do ambiente.

2. **TV**: A televisão pode ser vista como um dispositivo que oferece recursos de saída de áudio bem como exibição de imagens na sua tela.

3. **Câmera de vídeo**: A câmera pode ser utilizada tanto para a gravação como envio de imagens estáticas bem como vídeos do ambiente.

4. **Sensor de Movimento**: O sensor de movimento informa sobre a presença de pessoas dentro do ambiente.

5. **PC**: O computador pessoal oferece recursos de armazenamento além de executar aplicações, como o aplicativo de personalização do ambiente apresentado anteriormente.

6. **Celular**: O celular (assim como a televisão) pode oferecer diversos recursos como entrada e saída de áudio, exibição de imagens em seu visor, armazenamento de dados (como músicas) e permitir a execução de aplicativos.

Neste ambiente representado podemos ver a multiplicidade de atributos que um dispositivo pode agregar. No caso da televisão, temos um dispositivo que abriga dois recursos distintos, um de saída de áudio e um de saída de vídeo. O computador pessoal tanto abriga um recurso de armazenamento de dados, como um aplicativo de personalização do ambiente. E por fim temos o ar condicionado que apesar de abrigar apenas um recurso este disponibiliza três funcionalidades distintas. Na *DSOA* esta multiplicidade de atributos ocorre de acordo com características dos dispositivos e do ambiente no qual se encontram.

4.1.1.2 Dispositivo

> O dispositivo é um equipamento computacional com a capacidade de se comunicar, abrigar aplicações ou tornar recursos disponíveis ao *smart space*.

Com base nesta definição vemos que os dispositivos são responsáveis por abrigar as capacidades que o ambiente possui. É através dos dispositivos e seus recursos que serão possíveis as interações junto ao usuário bem como a troca de informações necessária para tal. Dentro desta visão podemos ver os dispositivos como um agrupamento de recursos disponíveis no ambiente, bem como a entidade responsável por hospedar as aplicações em execução.

No exemplo da nossa sala inteligente podemos identificar dispositivos que abrigam apenas um recurso assim como dispositivos dotados de diversos recursos. O ar condicionado pode ser visto como um único recurso responsável pelo controle da temperatura no ambiente. Por outro lado o celular pode ser visto como o agrupamento de diversos recursos (saída e entrada de áudio, saída de vídeo, armazenamento de dados) além de hospedar aplicações.

4.1.1.3 Recurso

Um dos conceitos chave envolvidos na *DSOA* é o recurso. É através deles que os dispositivos possuem acesso aos serviços e estabelecem um conhecimento das

funcionalidades disponíveis no ambiente. A definição de recurso na *DSOA* se encontra a seguir:

> Um recurso é um grupo de funcionalidades logicamente relacionadas. Estas funcionalidades devem ser acessíveis através de interfaces pré-definidas.

Para exemplificar consideremos o recurso de "câmera" disponibilizado pela "*webcam*" da nossa sala inteligente (figura 4.3). Podemos inicialmente identificar as seguintes funcionalidades básicas a serem providas pelo recurso de câmera:

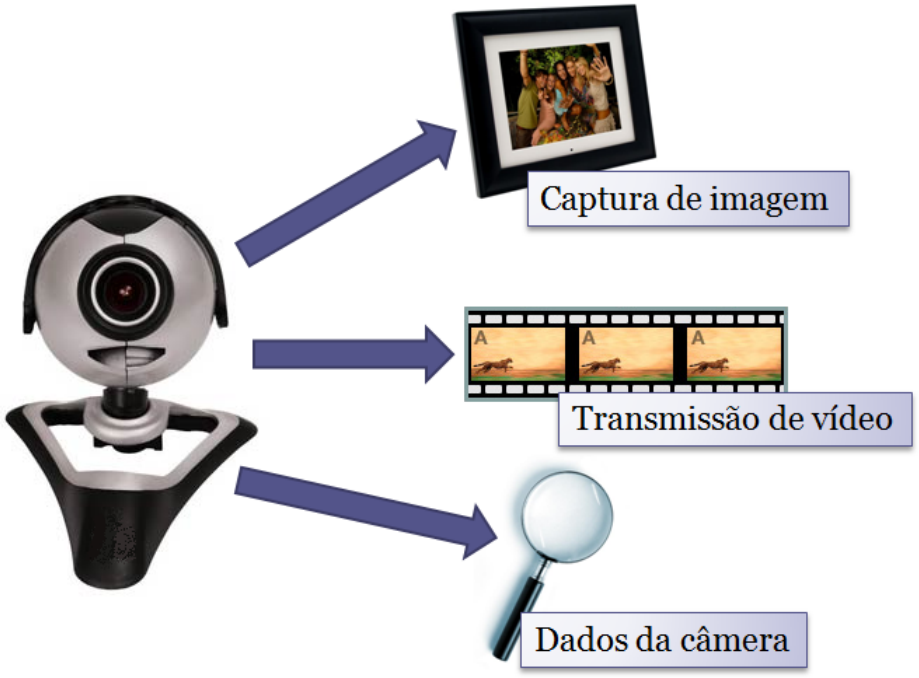


Figura 4.3: Decomposição de um recurso em funcionalidades de acordo com a *DSOA*.

- Captura de imagem: permite aquisição de fotos capturadas pela câmera.
- Transmissão de vídeo: disponibiliza a visualização em tempo real das imagens obtidas pela câmera de acordo com parâmetros de codificação adequados.
- Dados da câmera: informa usuários e outros recursos das características da câmera (resolução máxima, formatos de envio de vídeo, etc.).

Cada uma destas funcionalidades pode ser representada através de um ou mais serviços que devem ser disponibilizados de acordo com interfaces adequadas. Note que as funcionalidades providas pelo recurso não possuem uma relação direta entre si. A execução da funcionalidade de captura de imagem não afeta a transmissão de vídeo. Desta maneira os serviços de um recurso podem apresentar efeitos

relacionados, porém não é isto que determina o relacionamento lógico que os define como um recurso na *DSOA*. Neste caso o que relaciona estes serviços entre si se encontra no fato destes estarem ligados à captura de informações visuais do ambiente de acordo com um ponto de vista ("lente") do ambiente.

Apesar da denominação de recurso poder se confundir com os próprios recursos físicos dos dispositivos (câmera, teclado, mouse, etc.) a definição apresentada na *DSOA* permite a existência de algo mais amplo. Como os recursos são agrupamentos lógicos de serviços, podemos ter recursos que não estão diretamente relacionados a elementos físicos do ambiente. Sendo assim um recurso pode ser disponibilizado através da composição de outros recursos e seus serviços. Um exemplo é um recurso de conversão de vídeo que possibilita o acesso aos recursos de vídeo presentes no ambiente em um formato desejado. Este recurso não está relacionado a um elemento físico específico do ambiente, de fato ele atua como um consumidor de outros recursos de vídeo e se utiliza dos dados recebidos para realizar a conversão adequada aos seus usuários.

4.1.1.3.1 Interface do recurso :

Na *DSOA* os recursos são as unidades primárias de colaboração entre os dispositivos. É através deles que serão identificadas as funcionalidades presentes no ambiente e quais serviços deverão ser utilizados para acessar estas. Para que as entidades presentes no *smart space* possuam um entendimento comum de cada recurso disponível, e quais destes são significativos para suas funcionalidades, é necessário definir uma interface de comunicação. A interface de um recurso é composta por apenas dois atributos:

- Um *identificador*, ou *nome* do recurso, responsável por identificar unicamente um recurso em meio a outros presentes no ambiente.
- O *conjunto de serviços* que compõem o recurso e são disponibilizados através deste.

Duas instâncias de recurso serão equivalentes caso estes dois atributos sejam equivalentes em ambas simultaneamente. Esta equivalência é utilizada para se determinar a compatibilidade quando uma aplicação cliente busca um recurso que deseja acessar ou na identificação de várias instâncias de um mesmo recurso no ambiente. Uma instância de um recurso corresponde a cada "objeto" disponível no ambiente que disponibiliza esta interface. No nosso ambiente da sala inteligente temos duas instâncias do recurso de saída de áudio, uma no celular e outra na televisão. Observe que um mesmo dispositivo pode possuir mais de uma instância do mesmo recurso (como um celular com diversas câmeras).

Caso duas instâncias presentes no ambiente possuam o mesmo identificador de recurso, porém com conjuntos de serviços distintos, teremos uma *colisão de serviços* e neste caso estas instâncias não serão consideradas compatíveis, ou seja, não representarão o mesmo recurso. Por outro lado, duas instâncias podem possuir o mesmo conjunto de serviços, porém identificadores distintos. Neste caso, ambas as instâncias também serão consideradas incompatíveis, mas não representarão conflito ao ambiente.

4.1.1.4 Serviço

Como visto anteriormente, os dispositivos encontram as funcionalidades que necessitam no ambiente através dos recursos disponíveis. Estas funcionalidades, por sua vez são representadas no ambiente através de serviços relacionados. A definição de serviço de acordo com a *DSOA* se encontra a seguir.

> O serviço é a implementação de uma funcionalidade disponibilizada no *smart space* através de um recurso e uma interface pública conhecida.

Como uma implementação de uma funcionalidade, um serviço apenas possui valor ao ambiente caso produza um efeito observável (sessão 3.3.3) às outras entidades presentes no ambiente. Estes efeitos normalmente estão ligados à troca de informações entre as entidades produtora e consumidora do serviço ou mesmo a alteração do estado do ambiente. É o efeito que caracteriza um serviço como uma funcionalidade que agrega valor ao *smart space*, justificando sua existência.

Vemos também que os serviços estão sempre relacionados a um recurso. Este pode ser composto por um único serviço, mas serviços correlatos sempre farão parte do mesmo recurso. Desta maneira garantimos critérios de coesão com relação à modelagem das funcionalidades no ambiente.

4.1.1.4.1 Interface do serviço :

Para que as aplicações clientes possam discernir entre os serviços presentes, é necessário que se estabeleça uma interface para que cada serviço seja unicamente identificado de acordo com o recurso em que está inserido. De acordo com a *DSOA* temos três informações que estabelecem a interface de um serviço, e através delas que se determina a equivalência entre serviços:

- O *recurso* do qual o serviço faz parte. Sem um recurso não há como um serviço ser encontrado por entidades clientes.
- O *identificador*, ou *nome* do serviço, responsável por identificá-lo unicamente dentro de um recurso.
- Os *parâmetros* necessários para a execução da funcionalidade relacionada ao serviço.

De acordo com esta definição de interface, temos alguns casos que valem ser ressaltados. Dois serviços podem possuir o mesmo identificador, porém pertencer a recursos distintos. Caso dois serviços com mesmo identificador existam no mesmo recurso, teremos um *conflito de definição*, o que é inválido de acordo com a especificação acima. Dois serviços presentes em instâncias distintas apenas serão equivalentes caso todos os elementos de sua interface sejam equivalentes, ou seja, estejam em recursos equivalentes, possuam os mesmos identificadores e possuam parâmetros compatíveis.

4.1.1.5 Aplicações

Uma aplicação é a implementação de um conjunto de comportamentos e regras relacionadas ao *smart space*, cujo o objetivo é a tomada de ação ou a interação junto ao usuário. As aplicações são hospedadas nos dispositivos do ambiente e se utilizam dos recursos e serviços do ambiente durante sua execução.

São as aplicações as responsáveis por implementar a inteligência do ambiente. Tomam por base as informações providas pelos recursos, e de acordo com o caso, levam à ações em prol do usuário. As ações são realizadas através do acesso aos serviços providos por recursos adequados no *smart space*.

Também é responsabilidade das aplicações intermediarem as interações do usuário junto ao ambiente. Para tal as aplicações devem se relacionar com os recursos para notificar o usuário ou receber os comandos por ele informados.

4.1.2 Estratégias de comunicação

Os serviços representam as ações que os dispositivos podem executar no ambiente. Tais ações envolvem uma troca de informação e possuem características que devem ser observadas. Tomemos como exemplo dois serviços distintos: uma solicitação de vídeo em tempo real em um recurso de câmera e um serviço de alteração de temperatura do ambiente em um recurso de ar condicionado. Cada um destes possui características que os diferenciam com relação à forma como cada serviço é prestado. Neste sentido a *DSOA* estabelece quatro estratégias de comunicação dividas em dois grupos distintos:

- Tipo de tráfego, direto ou contínuo.
- Tipo de interação, síncrona ou assíncrona.

4.1.2.1 Tipo de tráfego de dados

Na comunicação entre dois dispositivos podemos considerar, simplificadamente, dois casos distintos a serem tratados com relação à forma como se realiza a troca de informações. Vale ressaltar que, apesar de se referir a comunicação realizada entre as entidades como "mensagens", a *DSOA* (assim como a *SOA*) não limita qual a forma como a comunicação entre as partes deve ser implementada. Esta comunicação pode ocorrer tanto por troca de mensagens quanto por memória compartilhada, entre outras.

- *Mensagens Discretas* são mensagens de tamanho finito trafegadas entre as partes envolvidas em uma comunicação. Neste caso, as partes conhecem onde se inicia e onde terminam os dados trafegados dentro do meio de comunicação. Este tipo de comunicação é comum em solicitações de informações em pequena escala, dados em domínios discretos (como dados de *QoS* ou localização) ou na solicitação de tomadas de ações (como a solicitação de abertura de uma porta).

- *Dados Contínuos* são formatos compostos por *streams* de dados, onde não se tem conhecimento prévio da extensão destes dados. Este tipo de comunicação é ideal para a transmissão de grandes massas de dados (como a solicitação de uma transferência de arquivo) ou na transmissão de fluxos dados (como na transmissão de áudio e vídeo).

Este tipo de distinção entre a forma de tráfego de dados pode ser observada no seguinte exemplo. Consideremos o recurso de câmera em um ambiente inteligente. Este recurso disponibiliza, dentre outros, os serviços de solicitação de informações de configuração da câmera e de solicitação de transmissão de vídeo. Ao solicitarmos as informações de configuração da câmera (resolução, formatos suportados, etc.), a quantidade de informações trafegadas é pequena, junto a isto seu formato e tamanho são conhecidos e delimitados. Por outro lado, ao solicitarmos o acesso à transmissão de vídeo a quantidade de informação é bastante superior, além de que o início e o fim da transmissão não são conhecidos *a priori*.

4.1.2.1.1 Canais de Dados :

Tanto a forma em que é implementada a comunicação entre os dispositivos (fila de mensagens, memória compartilhada, etc.) quanto a tecnologia que se utiliza para esta comunicação (*Bluetooth*, *Ethernet*, etc.) não influenciam nas definições vistas anteriormente. Com base nas diferenças encontradas entre as alternativas de tráfego dos dados na *DSOA* faz-se necessário diferenciar de maneira lógica onde serão trafegados os dados entre as entidades dentro do meio físico escolhido. Desta forma são definidos dois tipos de canais lógicos onde serão trafegados os dados na *DSOA*. Estes canais encontram-se representados na figura 4.4 e descritos a seguir.

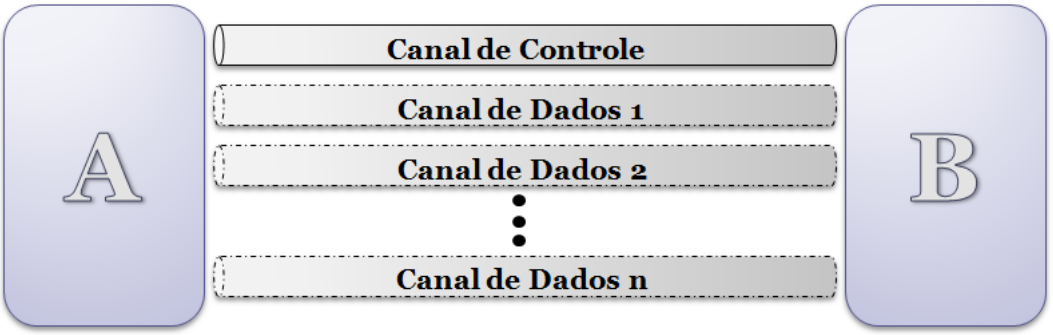


Figura 4.4: Representação dos canais lógicos de dados entre dois dispositivos A e B de acordo com a arquitetura *DSOA*

- Existe apenas um *canal de controle* onde são trafegadas *mensagens discretas*. Este canal sempre estará presente na comunicação entre os dispositivos, pois é nele que são trafegadas as mensagens de solicitação de serviços (mensagens de controle) e os retornos de dados correlatos.

- Podem existir diversos *canais de dados*. Estes são utilizados conforme necessário, e estabelecem a comunicações de tráfego contínuo de dados entre provedores e clientes. Estes canais podem até mesmo não existir no ambiente, caso as aplicações e serviços prestados não necessitem deste tipo de tráfego de informação.

4.1.2.2 Forma de interação entre dispositivos

Tradicionalmente a interação de serviços é vista como sendo uma extensão da chamada de funções em linguagem de programação. Esta interação onde se realiza uma requisição e aguarda o retorno da mesma não atende por completo as interações que ocorrem em um ambiente inteligente. Por muitas vezes uma aplicação não deseja tomar nenhuma ação, mas sim aguardar a ocorrência de determinado evento para então poder definir como deverá agir. Desta forma diferenciamos duas formas distintas dos dispositivos interagirem em um ambiente inteligente.

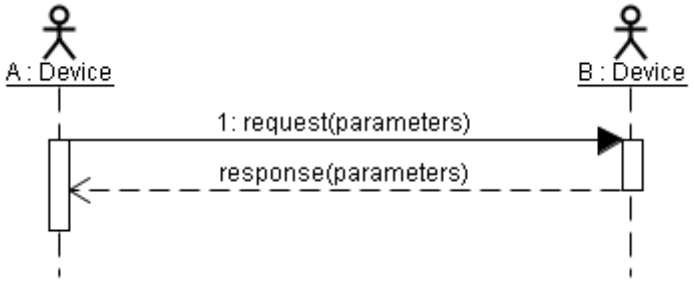


Figura 4.5: Representação de uma interação síncrona de acordo com a arquitetura *DSOA*

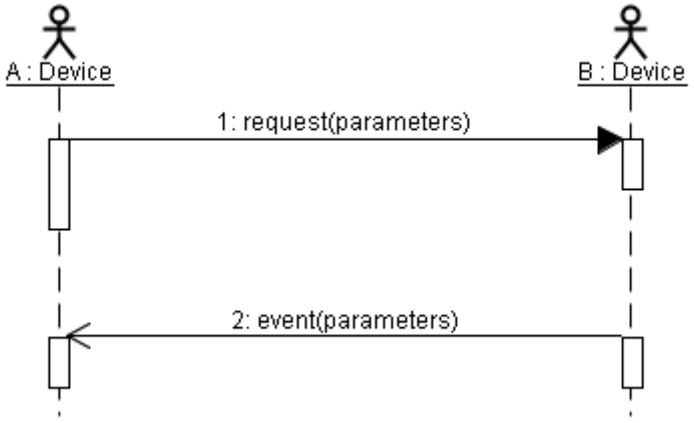


Figura 4.6: Representação de uma interação assíncrona de acordo com a arquitetura *DSOA*

- A interação *síncrona* (representada na figura 4.5) é adequada a solicitações de tomada de ações (como a solicitação de diminuição de temperatura em um ar-condicionado) ou solicitações de informações (como a consulta de temperatura do ambiente a um sensor). Neste tipo de interação, uma requisição de execução de serviço é seguida de uma resposta relativa a esta para que a aplicação cliente prossiga com suas tarefas.

- Uma comunicação *assíncrona* (representada na figura 4.6) atende a casos em que um dispositivo deseja aguardar a ocorrência de determinado evento para a execução de uma ação. Um exemplo é uma aplicação que aguarda a presença (*evento*) de uma pessoa no ambiente, capturada por um sensor, para então tomar a ação de acender uma lâmpada. Neste caso, a aplicação deve em um primeiro instante notificar o provedor de sua intenção de receber informações de ocorrência de determinado evento. Em um momento posterior arbitrário, que pode inclusive não ocorrer, o provedor ao identificar que tal evento solicitado ocorre notifica então à aplicação cliente. De posse das informações do evento, a aplicação cliente pode prosseguir com sua execução.

Para exemplificar a distinção destas duas formas de interação tomemos por base um recurso de sensor de temperatura do ambiente. Este recurso provê dois serviços distintos. Um serviço de consulta de temperatura do ambiente responsável por retornar a temperatura do ambiente no momento que foi solicitado. Este serviço é claramente síncrono, visto que necessitamos da informação no momento da solicitação. O segundo serviço é de notificação de mudança de temperatura de acordo com um limite estabelecido. Este serviço recebe como parâmetro um limite de temperatura a ser observado e, caso este limite seja alcançado, uma notificação é enviada à aplicação solicitante. Neste caso temos uma comunicação assíncrona, pois a aplicação cliente não possui conhecimento de quando o evento ocorrerá, ou mesmo se o mesmo ocorrerá em algum momento. E esta aplicação só iniciará sua execução quando este evento ocorrer.

4.2 uP - Ubiquitous Protocols

A fim de estabelecer uma interface adequada à visibilidade dos recursos e a interação entre os serviços de maneira compatível ao proposto pela *DSOA*, foi construído neste trabalho um conjunto de protocolos denominado *uP* (*Ubiquitous Protocols*). O papel do *uP* junto à *DSOA* é similar ao *SOAP* [62] e a arquitetura de *webservices*, junto à *SOA*. Este conjunto de protocolos segue uma linha similar ao proposto pelo *SLP* (seção 4.2.1), definindo como as entidades no ambiente devem interagir de forma a estabelecer um canal de comunicação e a descoberta dos serviços existentes. No *uP* é utilizado como formato de mensagens o *JSON* (seção 4.2.2), que provê um mecanismo leve e estruturado de representar a informação. Adicionalmente, para se construir esta interface de comunicação junto ao *smart space* são definidas as representações dos conceitos da *DSOA* (seção 4.2.3), as mensagens utilizadas (seção 4.2.4) e os protocolos estabelecidos (seção 4.2.5).

4.2.1 SLP - Service Location Protocol

O *Service Location Protocol* (*SLP*) [45] consiste em um padrão leve de comunicação com a finalidade de simplificar a descoberta de recursos (serviços) disponibilizados através de uma rede de comunicação entre diversos dispositivos. Ele não pressupõe nenhuma plataforma de *hardware*, protocolo de transporte ou linguagem específica, sendo simples a sua implementação para diversas plataformas.

O *SLP* define três papéis bastante similares aos definidos na *SOA*:

- *User Agents* : Responsáveis por acessar os serviços e repassá-los as aplicações clientes.
- *Service Agents* : Responsáveis por prover e anunciar os serviços disponibilizados.
- *Directory Agents* : Responsáveis por coletar e disponibilizar os dados acerca dos serviços disponíveis no ambiente.

O *SLP* visa automatizar os seguintes procedimentos:

- *Compatibilizar* as necessidades dos *User Agents* aos serviços oferecidos pelos *Service Agents*
- *A publicação* dos serviços providos pelos *Service Agents*
- *A organização* dos serviços em diretórios geridos pelos *Directory Agents*
- *A obtenção* de informações de serviços pelos *User Agents*
- *Prover meios* para os serviços informarem as aplicações clientes de suas capacidades e requisitos de configuração.

Conforme exposto na Figura 4.7, um exemplo de funcionamento do *SLP* consiste nos seguintes passos:

1. Descoberta de Serviços:
   (a) Um *User Agent* (*UA*) necessita de um determinado tipo de recurso, este requisita a um *Directory Agent* (*DA*) os *Service Agents* (*SA*) que possuem recursos compatíveis ao desejado.
   (b) O *DA* consulta sua base de recursos e retorna ao *UA* uma lista contendo os dados dos *SA*s e seus recursos compatíveis ao desejado pelo *UA*.
2. Seleção do Serviço:
   (a) O *UA* seleciona o *SA* que melhor lhe atende e requisita a este o acesso ao recurso desejado.

O *SLP* pode ser decomposto em duas funções principais e três auxiliares:

- Principais:

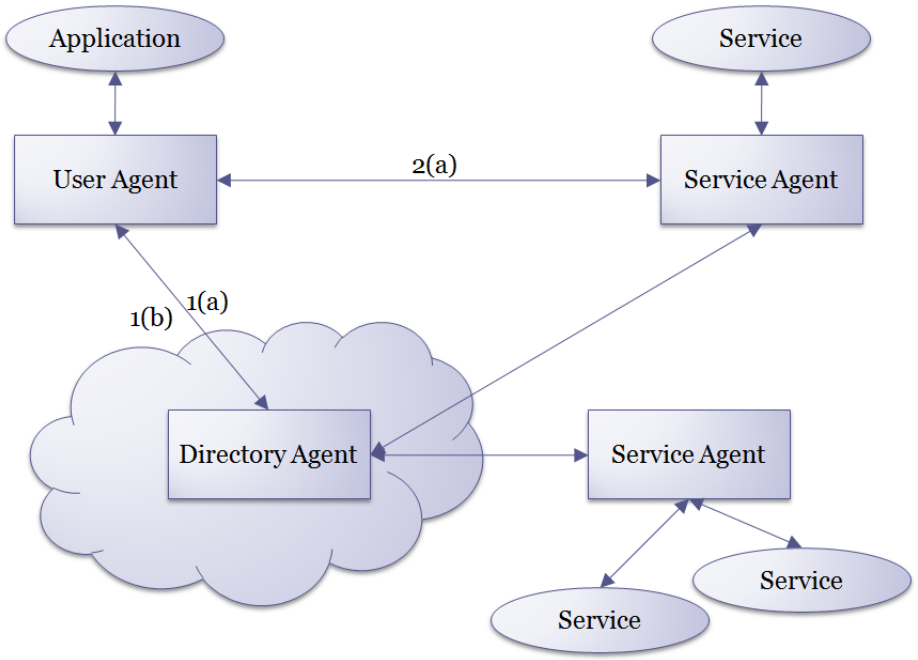


Figura 4.7: Exemplo de funcionamento do SLP em uma rede de dispositivos.

  - Obter os dados de acesso aos serviços para os *User Agents*.
  - Manter o diretório de serviços publicados.

- Auxiliares:
  - Descobrir os atributos dos serviços disponíveis.
  - Descobrir os *Discovery Agents* disponíveis.
  - Descobrir os tipos de *Service Agents* disponíveis.

Para atender a estas funções o *SLP* define nove tipos de mensagens:

- *Service Request* (Requisição de Serviço): Utilizada pelos *User Agents* para obter os dados de acesso a um serviço (como o endereço do serviço)
- *Service Reply* (Resposta de Serviço): Enviada aos *User Agents* juntamente com os dados de acesso a um serviço.
- *Service Deregistration* (Cancelamento de Serviço): Enviado por um *Service Agent* quando deseja descontinuar um serviço ou alterar determinados atributos.
- *Service Acknowledgment* (Confirmação de Serviço): Confirmação de recebimento de um pedido de registro ou cancelamento de registro.
- *Attribute Request* (Requisição de Atributos): Utilizada pelos *User Agents* para obter informações acerca das características de um serviço em particular.

- *Attribute Reply* (Resposta de Atributos): Provê os dados solicitados em uma requisição de atributos.
- *Directory Agents Advertisement* (Anúncio de Directory Agent): Uma das maneiras de um *Directory Agent* anunciar aos *Service Agents* e *User Agents* de sua presença no ambiente.
- *Service Type Request* (Requisição de Tipo de Serviço): Lista os serviços disponíveis.
- *Service Type Reply* (Resposta de Tipo de Serviço): Resposta da requisição de tipo de serviço solicitada.

O *SLP* determina que os *Directory Agents* podem ser definidos no ambiente de três maneiras distintas:

- Configuração Estática: Onde cada *Service Agent* e *User Agent* tem conhecimento prévio do *Directory Agent* ao qual ele deve se reportar, sendo este configurado estaticamente em cada *Agent*.
- Anúncio de *Directory Agent*: Onde o *Directory Agent* se anuncia no ambiente através da utilização de mensagens de Anúncio.
- Descoberta de *Directory Agent*: Onde cada *User Agent* e *Service Agent* busca no ambiente por um *Directory Agent* responsável.

O *SLP*, conforme apresentado, estabelece uma forma simples de descoberta de serviços em uma rede de dispositivos. Sua estrutura e definição de papéis se mostra compatível com a *SOA*, motivo pelo qual este modelo foi utilizado como base na definição do *uP*.

4.2.2 Formato de mensagens

Como formato para as mensagens do *uP* foi escolhido o *JSON* (*JavaScript Object Notation*) [35] [14]. O *JSON* é um formato de troca de informações de baixo custo computacional, conforme é exibido nos estudos realizados por *Bassani* e *Enoki* [16], além de ser um formato estruturado e auto-descritivo, assim como o *XML* [13]. Outra característica análoga entre o *JSON* e o *XML* está no fato de suas representações serem independentes de plataforma. Isto ocorre devido a utilização em suas mensagens a codificação *UTF-8* [65], suportada pela maioria das plataformas de *hardware* e *software* disponíveis. Esta interoperabilidade é uma característica importante para a comunicação entre dispositivos, dada a natureza heterogênea dos dispositivos presentes em um *smart space*.

Outra característica apresentada em [16] e ilustrada nas figuras 4.8 e 4.9 é o baixo custo computacional quando comparado ao *XML*. Na figura 4.8 conseguimos observar a conexão entre o tamanho da mensagem com relação ao tamanho da estrutura de dados representada. Os resultados demonstraram queo formato *JSON* leva a mensagens de menor tamanho que as representadas em *XML*. Por outro lado no gráfico 4.9, observamos a quantidade de tempo despendido no tratamento

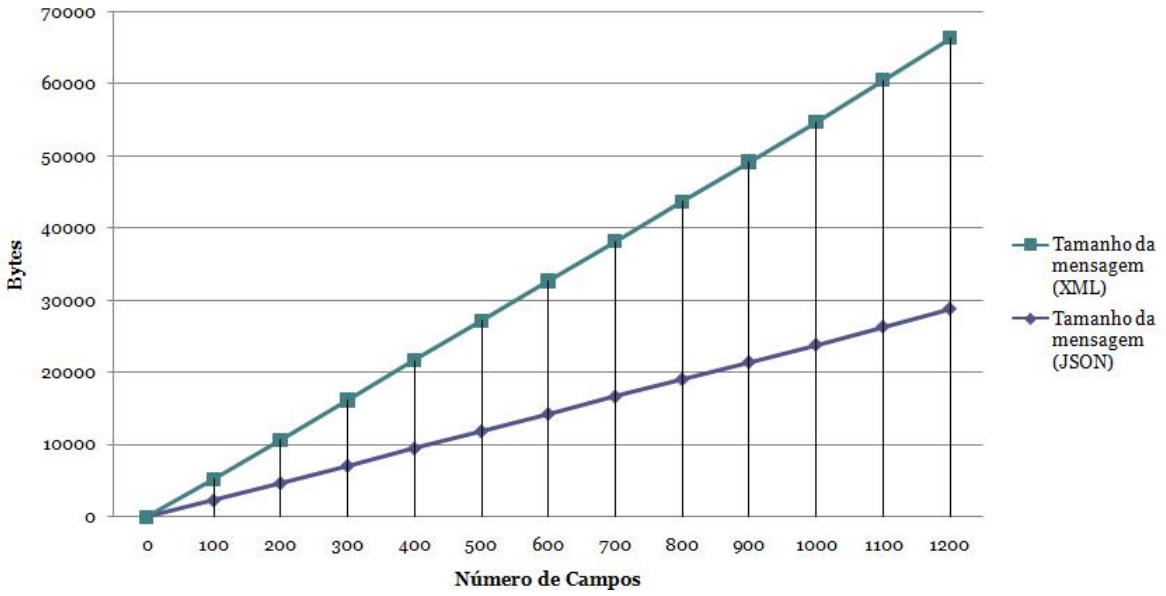


Figura 4.8: Relação entre o tamanho de uma mensagem e a quantidade de campos representados em JSON e XML

de uma mensagem com relação ao tamanho da estrutura de dados representada. Novamente, os resultados evidenciam que mensagens representadas em *JSON* demandam menor esforço com relação aquelas representadas em *XML*. Conforme visto na seção 2.4.1, é de intersse da *ubicomp* considerar e prover meios de que as limitações dos dispositivos posam ser minimizadas. Mensagens menores reduzem a necessidade de utilização da rede e de armazenamento para seu tratamento, mostrando-se como um fator positivo na inclusão de dispositivos com limitações de largura de banda e memória. Por outro lado, a redução no tempo de tratamento de tais mensagens leva a um menor consumo do poder computacional do dispositivo, economizando recursos de processamento e bateria. Tendo em vista que um dos focos deste trabalho é viabilizar a integração de dispositivos limitados na construção de um ambiente ubíquo, estas características se mostraram chave para a escolha do *JSON* como formato de mensagem para o *uP*.

4.2.3 Representações

A fim de prover um entendimento do *smart space* em conformidade com os conceitos apresentados pela *DSOA*, são definidos no *uP* representações que determinam como tais conceitos podem ser expressados.

A principal representação do ambiente está nos dispositivos que o compõem. Seguindo a linha da *DSOA*, no *uP* cada dispositivo (*Device*) possui duas propriedades:

- *"name"*: Nome responsável por identificar o dispositivo no *smart space*.
- *"networks"*: Lista de interfaces de rede ao qual o dispositivo está conectado. É composto por pares do tipo de rede e do endereço do dispositivo.

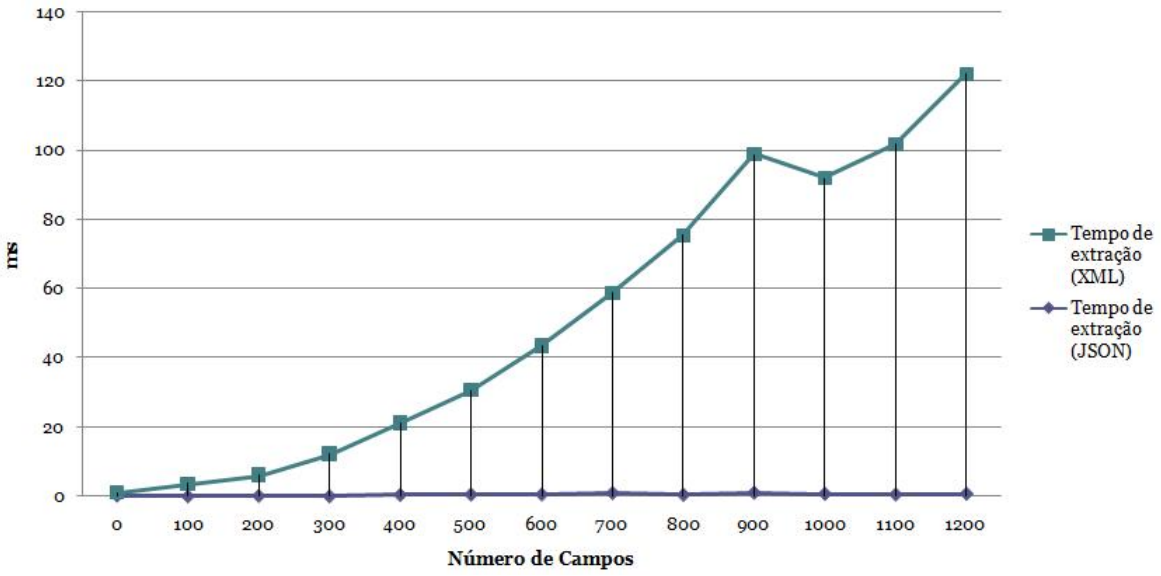


Figura 4.9: Relação entre o tempo para se tratar uma mensagem e a quantidade de campos representados em JSON e XML

Através destas informações é possível identificar cada dispositivo pertencente ao ambiente de maneira única, além de obter conhecimento sobre as interfaces disponíveis para comunicação junto ao mesmo. Na listagem A.1 encontramos a representação de um *Device* no *uP*.

O *driver* é a representação do conceito de recurso apresentado pela *DSOA*. Cada instância de recurso é endereçada no *uP* através de um identificador único ("*instanceId*") dentro do dispositivo ao qual este faz parte. Um *driver* apresenta a interface de um recurso sendo composto por:

- *"name"*: Nome responsável por identificar o recurso disponível no *smart space*.
- *"services"*: Lista de serviços síncronos disponibilizados pelo recurso.
- *"events"*: Lista de serviços assíncronos disponibilizados pelo recurso.

A listagem A.2 apresenta um exemplo da representação de um *driver*.

O *Serviço* é a representação do conceito homônimo na *DSOA*. A interface de um *Serviço* é definida pelos seguintes dados:

- *"name"* : Nome responsável por identificar o serviço disponível no recurso.
- *"parameters"* : Lista de parâmetros necessários para a execução do serviço. Estes parâmetros podem ser definidos de acordo com duas categorias:
  - "*OPTIONAL*" : Indica que a presença do parâmetro na requisição do serviço é opcional.

– "*MANDATORY*" : Aponta que o parâmetro possui presença obrigatória na requisição do serviço.

Na listagem A.3 temos um exemplo de definição de um serviço isolado e na listagem A.4 um exemplo da definição de um *driver* completo.

4.2.4 Mensagens

O *uP* define quatro tipos de mensagens responsáveis por definir as interações possíveis no *smart space*. Elas dão suporte aos protocolos definidos pelo *uP*, bem como podem ser personalizadas e incrementadas para ampliarem seus comportamentos e características.

As mensagens no *uP* seguem uma estrutura conforme representado na figura 4.10, onde todas apresentam um atributo "*type*" responsável por identificar qual tipo de mensagem que será representada e (opcionalmente) um identificador de erro. O atributo "*error*" traz consigo a identificação de um erro que possa ter ocorrido durante a execução relacionada a mensagem em questão, informando a outra parte da interação que aquela operação não ocorreu conforme esperado. A listagem A.5 apresenta uma representação básica do formato de uma mensagem no *uP*.

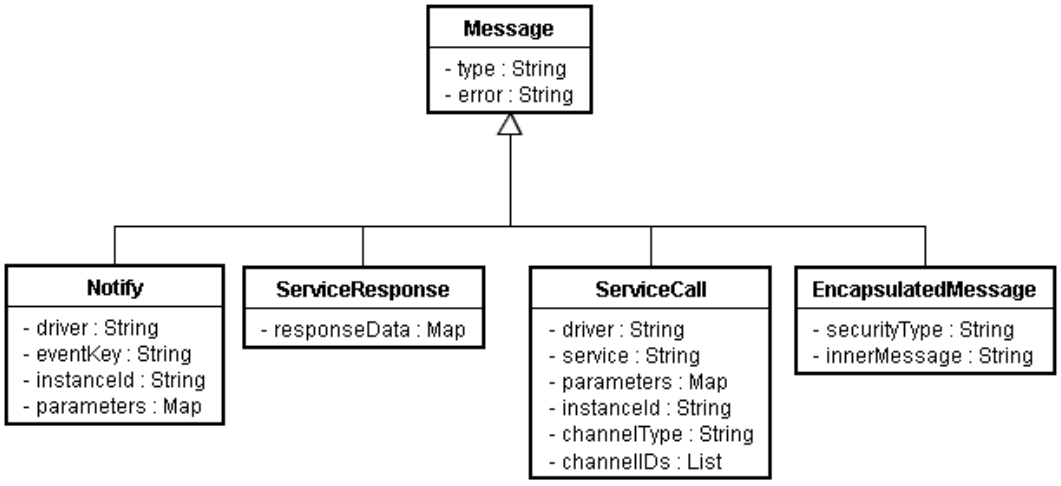


Figura 4.10: Representação de mensagem no *uP*.

4.2.4.1 Service Call

A *Service Call* é uma mensagem que representa a solicitação de um serviço síncrono uitilizando do *uP*. Através dela trafegam parâmetros que permitem a definição de qual serviço deve ser executado no dispositivo provedor. Os parâmetros trafegados em uma *Service Call* são definidos através de suas propriedades, sendo um conjunto destas propriedades obrigatórias e as demais opcionais (possuindo valores padrão).

Propriedades obrigatórias

- **type**: Esta propriedade indica o tipo de mensagem sendo trafegada. No caso do *Service Call* seu valor é "*SERVICE_CALL_REQUEST*".
- **driver**: Identificador do recurso (*driver*) a ser invocado o serviço em questão.
- **service**: Nome do serviço a ser chamado.
- **parameters**: Objeto contendo os parâmetros a serem repassados ao serviço para a sua execução.

Propriedades opcionais

- **instanceId**: Identificador único da instância do *driver* no dispositivo. Como cada dispositivo pode possuir diversas instâncias de um mesmo *driver* (recurso), como um celular com duas câmeras, cada instância é identificada através de um identificador de instância. Caso não seja informado, fica a critério da implementação em questão em selecionar arbitrariamente uma das instâncias disponíveis para execução.
- **serviceType**: Esta propriedade indica a forma de tráfego dos dados no serviço (*vide seção 4.1.2.1*) e pode assumir os valores '*DISCRETE*' para tráfego de dados discreto, ou "*STREAM*" para o tráfego de dados contínuo. Caso não informado, assume-se que o serviço possui tráfego de dados discreto. No caso do tráfego de dados contínuos devem ser informados os parâmetros "*channelIDs*" e "*channelType*".
- **channelIDs**: Os identificados dos canais de dados criados para esta comunicação.
- **channelType**: Tipo de rede utilizado para o canal de dados.

Um exemplo de uma mensagem de *Service Call* é apresentada na listagem A.6.

4.2.4.2 Service Response

A *Service Response* é a mensagem responsável por trafegar os dados de resposta de uma chamada de serviço (iniciado no *Service Call*). Suas propriedades são:

- **type**: Esta propriedade indica o tipo de mensagem sendo trafegada. No caso do *Service Response* seu valor é "*SERVICE_CALL_RESPONSE*".
- **responseData**: Objeto (opcional) contendo as informações retornadas na execução do serviço.

Na listagem A.7 temos um exemplo de um *Service Response*.

4.2.4.3 Notify

O *Notify* é responsável pelo tráfego de notificações de eventos correspondentes a serviços assíncronos. Esta mensagem opera de maneira similar ao *Service Response* trafegando os dados sobre a ocorrência de determinado evento. Suas propriedades são:

- **type**: Esta propriedade indica o tipo de mensagem sendo trafegada. No caso do *Notify* seu valor é "*NOTIFY*".
- **eventKey**: Nome do serviço (evento) que ocorreu.
- **driver**: Identificador do recurso (*driver*) onde ocorreu o evento.
- **instanceId**: Identificador da instância onde ocorreu o evento.
- **parameters**: Objeto (opcional) contendo as informações sobre a ocorrência do evento.

A listagem A.8 apresenta um exemplo de uma mensagem *Notify*.

4.2.4.4 Encapsulated Message

O *uP* define um tipo de mensagem especial denominada de *Encapsulated Message* cujo o objetivo é permitir o tráfego de mensagens codificadas através do protocolo. Tais mensagens funcionam como "envelopes" responsáveis por encapsular as demais mensagens do protocolo em um formato específico. Este tipo de mensagem pode ser utilizada tanto na compactação de mensagens como na representação de canais seguros. Os campos que a compõem estão definidos a seguir:

- **type**: Esta propriedade indica o tipo de mensagem sendo trafegada. No caso do *Encapsulated Message* seu valor é "*ENCAPSULATED_MESSAGE*".
- **securityType**: Identificador do tipo de codificação no encapsulamento da mensagem.
- **innerMessage**: Mensagem encapsulada.

4.2.5 Os Protocolos

O *uP* é composto por uma série de protocolos com o objetivo de atender as funcionalidades estabelecidas pela *DSOA*. Para prover as características de visibilidade, interação e efeito é necessário prover mecanismos de acesso aos recursos (*drivers*), publicação e descoberta destes no *smart space*. Para este fim, o *uP* divide seus protocolos em duas categorias: os protocolos básicos, utilizados para prover a invocação de serviços, e os protocolos complementares que, baseados nos protocolos básicos, complementam as funcionalidades necessárias ao *uP*.

4.2.5.1 Protocolos básicos

Existem dois protocolos básicos no *uP*, o *SCP* (*Service Call Protocol*), responsável por operacionalizar as chamadas de serviços síncronos, e o *EVP* (*Event Protocol*), responsável pelo acesso a serviços assíncronos (vide seção 4.1.2.2).

A figura 4.11 apresenta como o *uP* opera através do uso das mensagens *Service Call* e *Service Response*.

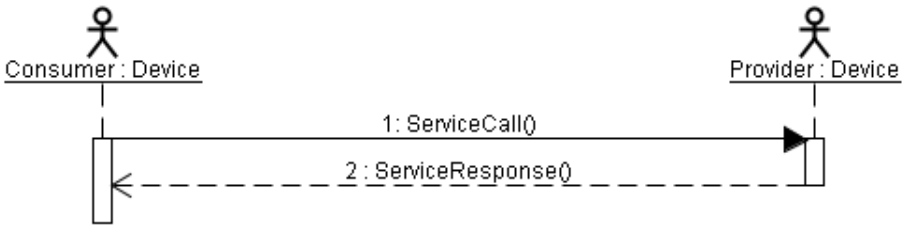


Figura 4.11: Representação da troca de mensagens no protocolo básico *SCP*.

1. O *consumidor* envia ao *provedor* uma mensagem de *Service Call* contendo as informações necessárias para a execução do serviço solicitado.

2. O *provedor* retorna ao *consumidor* uma mensagem de *Service Response* contendo as informações acerca da execução do serviço.

A interação ocorrida no *EVP* é representada na figura 4.12. Vale ressaltar que para que um *driver* de recurso possa suportar eventos este deve possuir os serviços "*registerListener*" e "*unregisterListener*" responsáveis por realizar o registro e remoção de um dispositivo para a notificação de determinados eventos.

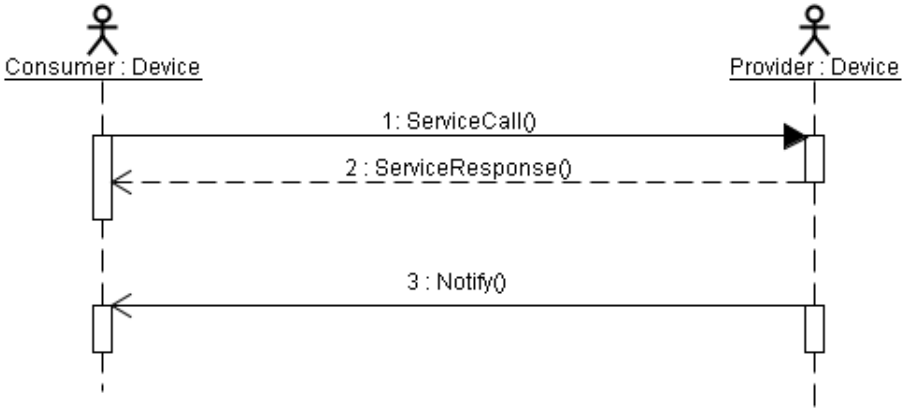


Figura 4.12: Representação da troca de mensagens no protocolo básico *EVP*.

1. O *consumidor* envia ao *provedor* uma mensagem de *Service Call* contendo as informações acerca do evento (serviço assíncrono) que deseja ser notificado. A especificação do evento é informada no parâmetro "*eventKey*" em uma chamada ao serviço "*registerListener*".

2. O *provedor* retorna ao *consumidor* uma mensagem de *Service Response* informando que o cadastro ocorreu com sucesso.

3. No caso da ocorrência do evento, o *provedor* envia ao *consumidor* uma mensagem de *Notify* contendo as informações acerca o ocorrido.

Em qualquer um destes protocolos suas mensagens podem ser trafegadas de maneira encapsulada. Neste caso, não existe diferenciação na forma como ocorre a interação, pois cada mensagem, após ser decodificada, é tratada conforme especificado.

4.2.5.2 Protocolos complementares

Os protocolos básicos fornecem os mecanismos necessários a interação dos serviços dentro do *smart space* de acordo com as estratégias definidas pela *DSOA*. Podemos ver no *SLP* que a interação de serviços corresponde a apenas dois de seus nove protocolos. Portanto, além da interação direta dos serviços, o protocolo deve fornecer mecanismos que permitam as aplicações no ambiente conhecer quais serviços estão disponíveis e as informações acerca dos mesmos. É com esta finalidade que o *uP* define os *protocolos complementares* que se encontram divididos em dois grupos, os relativos ao dispositivo e os relativos ao smart space. Cada um destes grupos é mapeado através de um *driver* de recurso responsável por definir a interface de comunicação destes "serviços" de forma coesa.

4.2.5.2.1 Device Driver :

O grupo de protocolos responsável por disponibilizar acesso a funcionalidades intrínsecas do dispositivo ou fornecer informações sobre o mesmo é mapeado no *DeviceDriver* cujo o identificador é "*br.unb.unbiquitous.ubiquitos.driver .DeviceDriver*". Seus serviços são:

- *"ListDrivers"*: Este protocolo retorna uma lista com as instâncias dos *drivers* disponíveis no dispositivo. Ele recebe (opcionalmente) os seguintes parâmetros que atuam como filtros nesta listagem. Caso nenhum filtro seja informado todas as instâncias serão retornadas.
  - "*serviceName*": Nome do serviço a ser buscado nas instâncias disponíveis.
  - "*driverName*": Identificador do recurso a ser buscado nas instâncias disponíveis.
- *"Handshake"*: Através deste protocolo dois dispositivos trocam informações entre si. Este protocolo recebe como parâmetro obrigatório um objeto do

tipo *device* sob a chave "*device*" com as informações do dispositivo que o invocou e retorna um objeto do mesmo tipo com as informações do dispositivo que recebeu a chamada.

- *"Goodbye"* : Este protocolo tem por objetivo informar que as informações sobre o dispositivo client do serviço não são mais válidas e devem ser removidas. É utilizado quando um dispositivo deixa o *smart space*.

- *"Authenticate"* : Responsável por estabelecer um contexto de segurança entre dois dispositivos, este protocolo depende do esquema de segurança a ser utilizado. O esquema de segurança é informado através do parâmetro obrigatório "*securityType*". Um exemplo de implementação deste protocolo pode ser encontrado em [53].

4.2.5.2.2 Register Driver :

O *Register Driver* disponibiliza as informações que o dispositivo possui acerca *smart space* a sua volta. Estes serviços são utilizados por outros dispositivos a fim de obter informações sobre o ambiente e seus recursos. Este *driver* é definido sobre o identificador "*br.unb.unbiquitous.ubiquitos.uos.driver.RegisterDriver*" e é compostos pelos seguintes serviços:

- *"ListDrivers"*: Este protocolo retorna uma lista com as instâncias de *drivers* de recursos disponíveis no *smart space*. Tais recursos estão presentes nos dispositivos conhecidos pelo *register* consultado, e são pertencentes aos seus dispositivos vizinhos. Ele recebe (opcionalmente) os seguintes parâmetros que atuam como filtros nesta lista. Caso nenhum filtro seja informado todas as instâncias serão retornadas.
  - "*serviceName*": Nome do serviço a ser buscado nas instâncias disponíveis.
  - "*driverName*": Identificador do recurso a ser buscado nas instâncias disponíveis.
  - "*device*": Nome do dispositivo onde deseja-se restringir a busca de informações.

- *"Publish"*: Permite a um dispositivo publicar uma instância de um *driver* de recurso a ser disponibilizado no *smart space*.

- *"UnPublish"*: Remove as informações sobre uma instância de um *driver* de recurso, tornando-a indisponível ao *smart space*.

4.2.5.3 Modos de operação

Assim como o *SLP*, o *uP* permite que os dispositivos no ambiente tomem conhecimento dos recursos disponíveis no *smart space* de diversas maneiras. Uma *configuração estática* ocorre no *uP* do mesmo modo que estabelecido no *SLP*, onde o dispositivo consumidor deve conhecer os dados do dispositivo provedor a fim de estabelecer a interação do serviço. Um *anúncio do registro* ocorre no

uP quando o mesmo realiza um *handshake* com cada dispositivo no ambiente, permitindo assim que estes tomem conhecimento de sua interface. Por fim uma *descoberta dos registros* ocorre quando cada dispositivo encontra um *register* e realiza o *handshake* junto a este. Adicionalmente a estas duas últimas opções operam os protocolos complementares de *ListDrivers*, *publish* e *unpublish*.

4.2.6 Segurança

Como foi visto, o protocolo "*Authenticate*" do "*DeviceDriver*" permite o estabelecimento de um contexto de segurança entre dois dispositivos. Após o estabelecimento deste contexto, os mesmos podem se comunicar utilizando as mensagens do tipo "*EncapsulatedMessage*", permitindo a utilização de um canal cifrado. Um exemplo de implementação neste sentido se encontra no protocolo de autenticação proposto em [53] e melhor detalhado a seguir (figura 4.13).

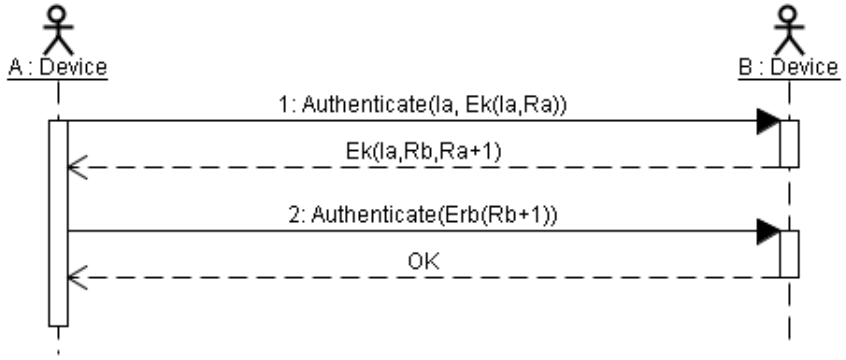


Figura 4.13: Representação do estabelecimento de um contexto de segurança.

1. O dispositivo "A" possui junto ao dispositivo "B" uma chave compartilhada ("k"). "A" produz um valor aleatório "Ra". A realiza uma chamada a "B" no serviço "*Authenticate*" com os seguintes parâmetros:
   - "Ia": A identificação (nome) de "A".
   - "Ek(Ia,Ra)": Valor encriptado da identificação de "A" ("Ia") concatenado com o valor aleatório produzido ("Ra") utilizando a chave compartilhada ("k").
2. O dispositivo "B" recebe esta mensagem e conhecendo a chave compartilhada ("k") decodifica os parâmetros. O dispositivo então calcula o incremento do valor informado ("Ra+1") e produz um novo valor aleatório ("Rb"). Por fim retorna a resposta (Ek(Ia,Rb,Ra+1)) com o valor codificado da identificação de "A" ("Ia"), o incremento do valor gerado pro "A" ("Ra+1") e o novo valor gerado por "B" ("Rb") utilizando a chave compartilhada ("k").
3. O dispositivo "A" recebe esta mensagem e decodifica-a, encontrando o valor do número gerado por "B" ("Rb") e realiza uma nova chamada ao serviço

"*Authenticate*" informando como parâmetro o valor do incremento deste valor ("Rb+1") codificado com a chave "Rb".

4. Recebendo o valor informado por "A", o dispositivo "B" retorna informando que "Rb+1" será utilizada como chave para esta sessão segura.

Desta maneira, com apenas duas chamadas ao serviço "*Authenticate*" dois dispositivos conseguem estabelecer um canal seguro de comunicação.

4.3 uOS - Ubiquitous Middleware

Conforme apresentado, o desenvolvimento de aplicativos para computação ubíqua é comumente apoiado pela utilização de *middlewares*. Estes têm por objetivo auxiliar na construção destas aplicações, empregando conceitos de reuso e simplificando as interações dentro dos padrões estabelecidos. Conforme visto na seção 2.3.4, o grupo de pesquisa *UnBiquitous* da Universidade de Brasília possui o projeto *UbiquitOS* [1]. Tal projeto envolveu o desenvolvimento do *middleware UbiquitOS* [24] [44] cujo o foco está na adaptabilidade de serviços. A fim de adequar esta implementação aos conceitos apresentados pela *DSOA* e a utilização do *uP* foi desenvolvido o *uOS* (Ubiquitous OS), um *middleware* para adaptabilidade de recursos no *smart space* que será apresentado neste capítulo.

4.3.1 UbiquitOS

Conforme discutido na seção 2.3.4, o *middleware UbiquitOS* foi concebido a fim de promover a adaptabilidade de serviços em um ambiente de computação ubíqua. Sua arquitetura está dividida em camadas distintas com o objetivo de simplificar seu desenvolvimento e promover a coesão entre as responsabilidades.

- Camada de rede: Esta é a camada responsável por gerenciar as interfaces de rede acessíveis ao dispositivo.
- Camada de conectividade: Esta camada opera como interface de comunicação dentro do *middleware*. Todas as informações recebidas pela camada de rede são coordenadas e direcionadas adequadamente a camada superior. De maneira similar toda chamada da camada de adaptabilidade é tratada e repassada à interface de rede responsável.
- Camada de adaptabilidade: Nesta camada ocorre o gerenciamento dos serviços disponíveis ao ambiente bem como o tratamento do acesso aos mesmos.

4.3.1.1 Interação

A figura 4.14 apresenta a arquitetura do *middleware UbiquitOS* em detalhe. Este *middleware* segue um modelo centralizado de dispositivos onde cada um pode

[1] Este trabalho é parte integrante do projeto *UbiquitOS*

assumir três papéis distintos: *ubiquitos-provider*, *ubiquitos-client* ou *ubiquitos-smartspace*. Cabe ao *ubiquitos-smartspace* gerenciar as informações e interações dentro do ambiente. Para que um serviço seja disponibilizado e acessado no ambiente deve-se seguir os seguintes passos:

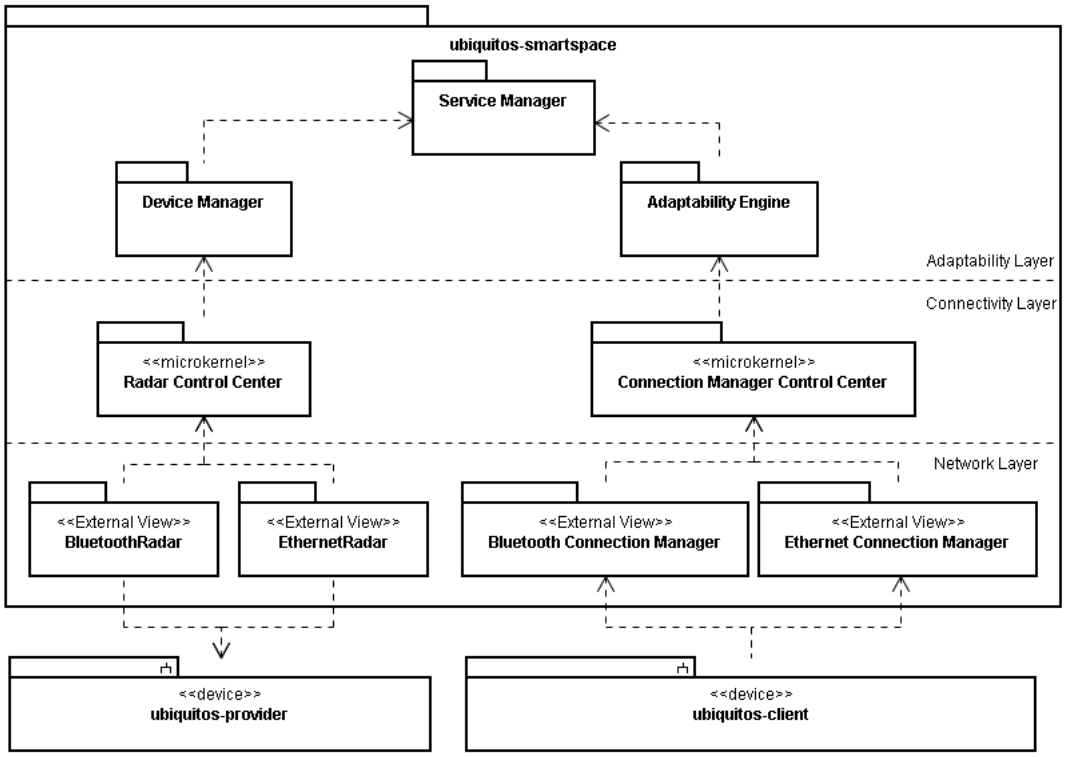


Figura 4.14: *Middleware UbiquitOS* - Diagrama de interação entre as camadas

- *Provider*:
  - O *ubiquitos-provider* responsável por disponibilizar o serviços deve defini-lo através de uma interface *Java* que deve ser implementada por um adaptador.

  O *ubiquitos-provider*, ao ser questionado pelo *ubiquitos-smartspace*, deve enviar a implementação do adaptador para a publicação do serviço.

- *Client*:
  - O *ubiquitos-client* que deseja acessar um serviço deve definir uma interface *Java* para o acesso a este. Tal interface deve ser implementada por um adaptador.
  - Ao necessitar acessar um serviço, o *ubiquitos-client* deve enviar ao *ubiquitos-smartspace* seu adaptador a fim de estabelecer o acesso ao serviço.

- *SmartSpace*:

– O *ubiquitous-smartspace* deve possuir uma implementação de um conector compatível com as interfaces dos adaptadores providos tanto pelo *ubiquitos-provider* como pelo *ubiquitos-client*.

– O iniciar uma chamada de serviço por parte do *ubiquitos-client*, cabe ao *ubiquitos-smartspace* realizar em seu conector a comunicação entre os dois adaptadores de serviço.

A utilização destes adaptadores assegura que tanto as partes (*client* e *provider*) concordam com a interface de serviço estabelecida bem como o próprio *middleware* está ciente de como intermediar esta comunicação. O gerenciamento dos adaptadores disponíveis no ambiente bem como a validação de interfaces é realizada utilizando a plataforma *JINI* [22]. Para que o *ubiquitos-smartspace* tome conhecimento dos serviços disponíveis em um *ubiquitos-provider*, assim como um *ubiquitos-client* tomar ciência dos serviços disponíveis no ambiente, são utilizados protocolos próprios definidos pelo *middleware*.

4.3.1.2 Problemas

A implementação do *middleware UbiquitOS* demonstra com sucesso a operacionalização dos conceitos de adaptabilidade de serviços no contexto da computação ubíqua. Porém, alguns pontos de sua implementação apresentam questões com relação a sua implantação efetiva em um *smart-space*.

- Organização da camada de rede: Mesmo suportando o acesso a interfaces de rede distintas, o *middleware* as trata como parte integrante do mesmo. Isto dificulta o desenvolvimento, integração e configuração de implementações para novas tecnologias de comunicação.

- Dependência de plataforma: A utilização do *JINI* para garantir a consistência entre as interfaces dos serviços causa uma dependência entre as aplicações desenvolvidas e a plataforma *Java*, impossibilitando o acesso por dispositivos sob outras plataformas.

- Centralização de acesso: Arquiteturas centralizadas são utilizadas em diversos *middlewares*, porém no *UbiquitOS*, devido a utilização dos adaptadores, a entidade central é responsável não só pelas informações sobre o ambiente como por intermediar todas as comunicações. Criando assim um "gargalo" nas interações do *smart space*.

- Complexidade de acesso aos serviços: A utilização de adaptadores visa assegurar a compatibilidade de interfaces no acesso aos serviços. Porém isto implica no desenvolvimento de três entidades distintas para cada serviço disponibilizado, onerando em excesso sua implementação.

4.3.2 uOS - Ubiquitous OS

O *uOS* é uma implementação de um *middleware* para auxílio no desenvolvimento de aplicações para ambientes de computação ubíqua. Sua proposta é compatível

com os conceitos apresentados pela *DSOA* e utiliza como interface de comunicação o conjunto de protocolos *uP*. A figura 4.15 apresenta o ecossistema onde se encontra o *middleware* junto ao projeto *UbiquitOS*. Neste contexto vemos o *uOS* como um componente de auxílio para o desenvolvimento de *drivers* de recurso, aplicações e *plugins* de rede a serem utilizados em ambientes ubíquos.

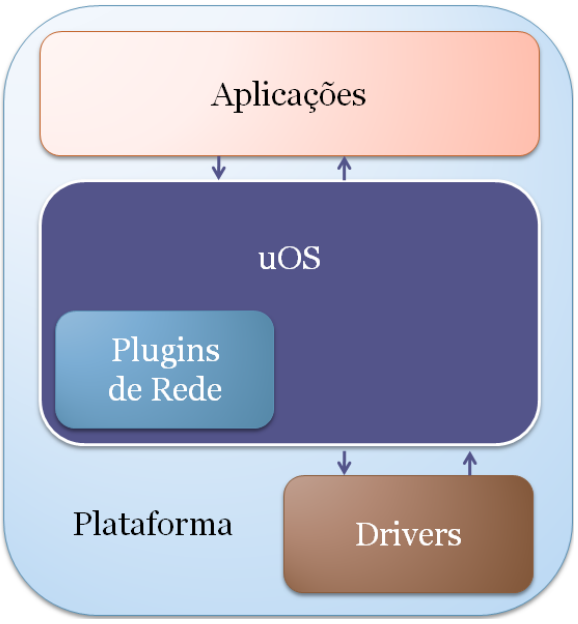


Figura 4.15: *uOS* - Ecossistema do projeto *UbiquitOS*

4.3.2.1 Arquitetura

O *uOS* segue uma arquitetura interna em camadas, assim como as presentes no *middleware UbiquitOS*. Apesar de estas camadas possuírem as mesmas responsabilidades, foram necessários ajustes para embarcar os conceitos definidos pela *DSOA* bem como comportar a utilização do protocolo *uP* em sua comunicação. A figura 4.16 apresenta o *uOS* e uma visão geral de suas camadas. Pode-se visualizar que a camada de adaptabilidade sofreu alterações, pois além de gerir o serviços, agora esta fornece interfaces para a gerência dos *drivers* e aplicações no dispositivo. A camada de conectividade foi ajustada para a utilização do *uP* como interface de comunicação e por fim a camada de rede para gerenciar e se integrar com os *plugins* de rede.

Como um *middleware* de sistema operacional, o objetivo do *uOS* é fornecer uma visão homogênea do *smart space* provendo um acesso transparente aos recursos disponíveis. Para tal, cada dispositivo deve possui o *middleware* bem como seus *drivers* e aplicações em execução. Desta maneira cada aparelho, de acordo com o que foi apresentado pela *DSOA*, será um componente colaborativo dentro do *smart space*.

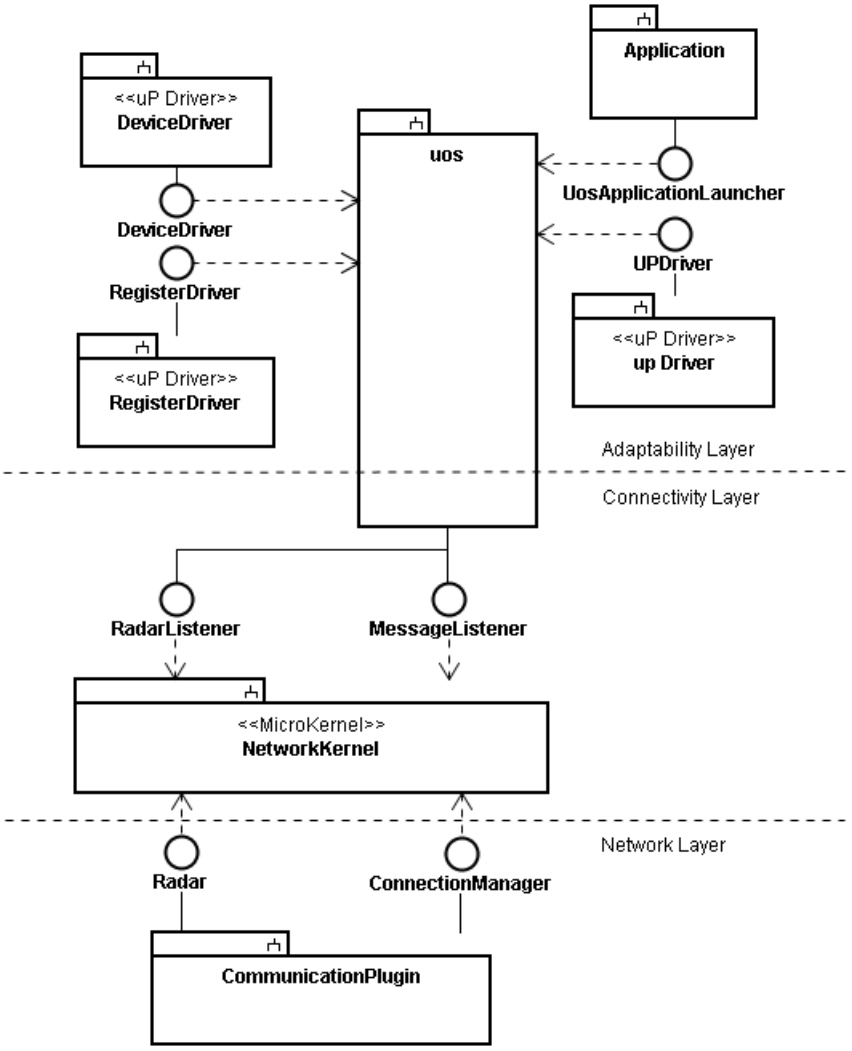


Figura 4.16: *Middleware uOS* - Diagrama de interação entre as camadas.

4.3.2.1.1 Camada de rede (*Network Layer*) :

A camada de rede é responsável por tratar as questões específicas de cada meio acessível ao dispositivo através de suas interfaces de rede. O acesso a estas é realizado através de *plugins* de rede específicos para cada tecnologia. No *uOS* cada *plugin* de rede é externo ao *middleware* e a comunicação deste junto a eles é realizada através de interfaces definidas. A figura 4.17 mostra um exemplo de *plugin* para a comunicação para interfaces de rede *Bluetooth*.

Os *plugins* e o *middleware* se comunicam através das interfaces "*ConnectionManager*" e "*Radar*". O *ConnectionManager* determina o responsável por gerenciar a criação e o recebimento de conexões na tecnologia de comunicação sob a responsabilidade do *plugin*. Fica a cargo do *Radar* a descoberta de novos dispositivos no meio, permitindo o controle de quem se encontra no ambiente. A presença de um radar não é obrigatória ao *plugin*. Em casos de *plugins* que não

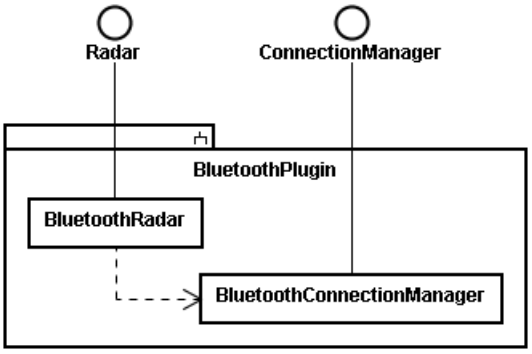


Figura 4.17: *Middleware uOS* - Bluetooth plugin.

possuam radar, ou em que este esteja desabilitado, não existe a descoberta de dispositivos. Nestes casos o dispositivo fica restrito as informações obtidas de outros dispositivos no ambiente, de informações registradas diretamente pelo usuário ou dados estáticos. Mais detalhes sobre estas classes pode ser encontrada em [44].

Durante o desenvolvimento do *uOS*, o suporte a *TCP*, *UDP* e *Bluetooth* disponível pelo *middleware UbiquitOS* foi encapsulado em *plugins* adequados a cada uma destas tecnologias. Juntamente foram desenvolvidos *plugins* específicos para *RTP* [38] e *LoopBack* [50]. Esta estrutura de utilização de *plugins* permite a adaptação do *middleware* para novas formas de comunicação tanto para plataformas de enlace (como *zigbee* [4] ou *PLC* [20]) ou de formatação dos dados (como *HTTP* [21], *RTSP* [26] ou *DCCP* [19]). Agregando assim um maior número de dispositivos e simplificando a construção de aplicações utilizando estes tipos de comunicação.

4.3.2.1.2 Camada de Conectividade (*Connectivity Layer*) :

A camada de conectividade é responsável por gerenciar a comunicação dentro do *middleware*. Encarrega-se de gerenciar os *plugins* da camada de rede bem como realizar a tradução dos dados trafegados em formato adequado. A figura 4.18 apresenta como a camada de conectividade se encontra dentro do *middleware*.

O núcleo de rede (*Network Kernel*) é responsável por gerenciar os *plugins* de rede para cada tecnologia de comunicação disponibilizado pela camada de rede. A coordenação dos *plugins* é feita pelo *Connection Manager* e pelo *Radar Manager*. O *Connection Manager* é responsável pelo controle das conexões criadas e recebidas em cada um dos *plugins* de rede. Esta entidade centraliza as solicitações de conexão das camadas superiores para os *plugins* devidos. Além disto, também é responsável por centralizar o controle das conexões recebidas para que recebam o tratamento devido. O *Radar Manager* gerencia o processo de gestão dos dispositivos presentes do ambiente. A entrada e saída destes são tratadas e notificadas às camadas acima, para que as ações devidas sejam realizadas nestes casos.

A *Message Engine* é responsável por traduzir os dados transmitidos na camada de rede em objetos compreendidos pelas demais camadas. As chamadas a

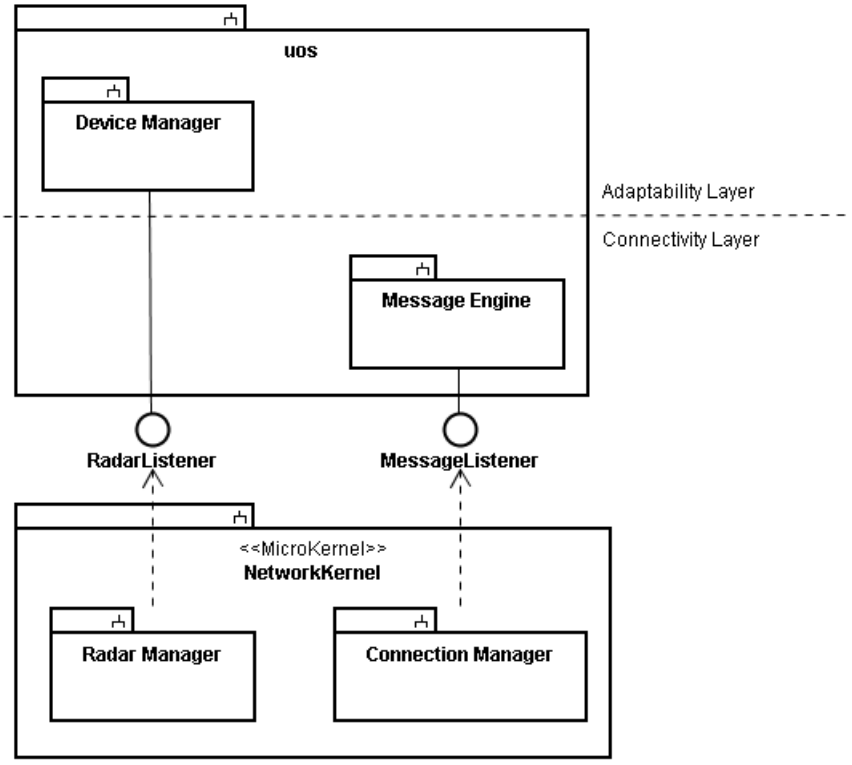


Figura 4.18: *Middleware uOS* - Connectivity Layer.

serviços (*SCP* e *EVP*, sessão 4.2.5.1) são recebidas pelas camadas superiores e as mensagens são traduzidas em formato *uP* para a transmissão na camada de rede. Na *Message Engine* são recebidos os dados advindos de conexões recebidas na camada de rede. Estes são então traduzidos em objetos de acordo com o formato *uP* de mensagens. Depois de traduzidas, estas mensagens são repassadas a camada de adaptabilidade para receber o tratamento adequado.

O *Device Manager* pertence da camada de adaptabilidade, sua responsabilidade consiste na gestão dos dispositivos no ambiente, portanto sendo notificado sobre a entrada e saída de dispositivos pelo *Radar Manager*. No caso do surgimento de novos dispositivos, o *Device Manager* realiza o processo de descoberta de informações acerca do dispositivo utilizando os protocolos *handShake* e *listDrivers* (sessão 4.2.5.2) através da camada de adaptabilidade.

4.3.2.1.3 Camada de Adaptabilidade (*Adaptability Layer*) :

A camada de adaptabilidade é responsável por coordenar as interações realizadas através do *middleware*. Para tal, esta camada fornece interfaces que se comunicam com os *drivers* e recursos e as aplicações em execução. A figura 4.19 apresenta uma visão geral da camada de adaptabilidade.

A *Adaptability Engine* é a entidade central desta camada sendo responsável por intermediar as comunicações entre aplicações e *drivers* com o *smart space*. A interação com os serviços providos pelos *drivers* são delegadas a *Message En-*

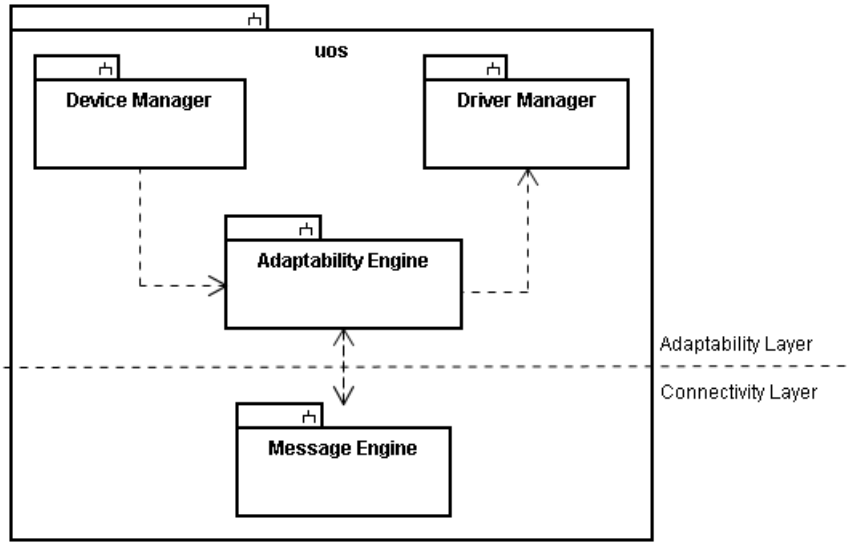


Figura 4.19: *Middleware uOS* - Adaptability Layer.

gine, entidade responsável por tratá-las e transmiti-las aos responsáveis. De maneira similar, chamadas a serviços recebidos pela *Message Engine* são delegadas a *Adaptability Engine* que em conjunto com o *Driver Manager* determina o *driver* de recurso responsável por tratar a chamada. O relacionamento entre a *Adaptability Engine* e o *Driver Manager* é detalhado na figura 4.20

A gestão das aplicações no *middleware* é realizada através da classe *Application Deployer* (conforme exibido na figura 4.21). Esta entidade é responsável pelo controle do ciclo de vida destes componentes no dispositivo. Assim como nos *drivers*, a interface das aplicações com o *smart space* é a *Adaptability Engine*.

4.3.3 Modos de operação

O *uOS* permite que os dispositivos atuem de duas maneiras distintas no *smart space*. Estes modos de operação permitem que o ambiente funcione de maneira centralizada, distribuída ou de maneira híbrida, de acordo com as necessidades e características de cada dispositivo, usuário e ambiente.

4.3.3.1 Modo Ativo

No modo ativo, o dispositivo se utiliza dos radares disponíveis para se atentar a cada alteração nas configurações do ambiente. A cada novo dispositivo descoberto, o protocolo de *handshake* em conjunto com o *listDrivers* são utilizados para descobrir quais serviços novos estão disponíveis para serem utilizados. Por outro lado, a cada dispositivo que abandona o *smart space*, os serviços providos são removidos da base de dados, tornando-os inacessíveis.

O modo ativo é utilizado em dispositivos que necessitem manter os dados dos recursos presentes no ambiente sempre atualizados e de rápido acesso antes da tomada de suas decisões. Este modo de operação se mostra adequado para

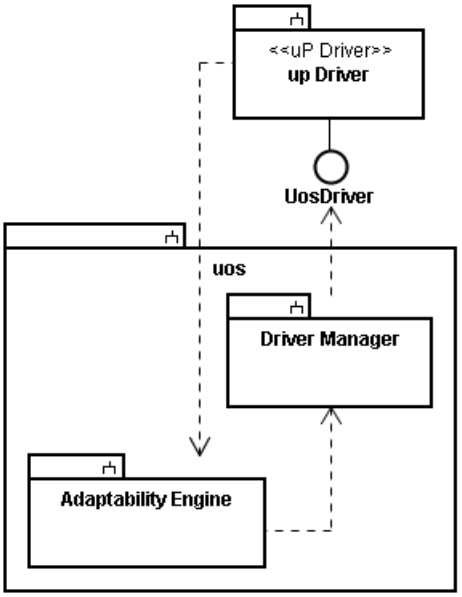


Figura 4.20: *Middleware uOS* - Adaptability Layer com destaque no gerenciamento dos *drivers*.

esta finalidade, mas não se restringe aos dispositivos que atuam como *register* no *smart space*. Além disto, a utilização constante do radar apresenta um ônus ao dispositivo devido à intensidade de acesso a rede e da tecnologia de rede utilizada. Este modo de operação, portanto, não se mostra adequado para dispositivos com limitada capacidade de processamento e comunicação.

4.3.3.2 Modo Passivo

Dispositivos operando em modo passivo obtêm as informações acerca do ambiente através de descobertas realizadas por outros dispositivos, de informações previamente conhecidas ou fornecidas pelo usuário durante sua execução. Neste modo, o dispositivo não atua no *smart space* em busca das informações. Assim o dispositivo busca a informação acerca dos recursos disponíveis consultando os dispositivos conhecidos (*registers*) através de interações realizadas *a priori*. O tratamento de recursos não presentes no ambiente é realizado através do tratamento de erros no acesso a estes. Dispositivos com limitações de energia ou de comunicação encontram no modo passivo uma maneira mais adequada de integrar o *smart space*.

4.3.3.3 Proxying e Filtering

Um *register* opera como um recurso de páginas amarelas dentro do ambiente. Outros dispositivos buscam informações ao *register* a fim de conhecer quais recursos se encontram disponíveis. Um problema presente nesta abordagem para o ambiente de computação ubíqua reside na questão da visibilidade e conectividade [50] entre os dispositivos.

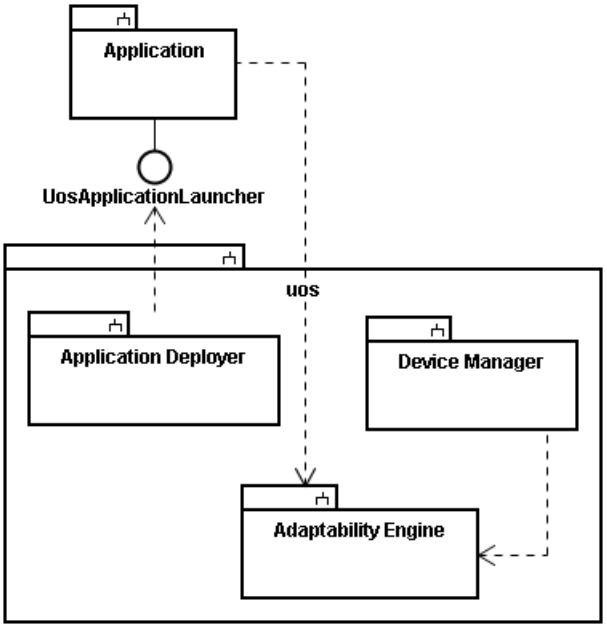


Figura 4.21: *Middleware uOS* - Adaptability Layer com destaque no gerenciamento das aplicações.

Na figura 4.22 temos um conjunto de dispositivos em um *smart space* para exemplificar este caso. As linhas representam as possibilidades de comunicação direta neste ambiente. O dispositivo em destaque ao centro ("R") representa um *register* que possui acesso a todos os dispositivos neste cenário. Porém não existe conectividade direta entre os dois dispositivos a esquerda ("A" e "B") e os três dispositivos a direita ("C","D" e "E").

Sendo "R" o *register* do ambiente, ele será consultado pelos demais caso necessitem de informações sobre o ambiente. Supondo que este não realize nenhum tratamento sobre o estas informações, de forma que a cada consulta retorne todo o conhecimento que possui. Caso "A" consulte os recursos disponíveis terá como retorno todos os recursos do ambiente. Incluindo as informações sobre seus próprios recursos (informação redundante) e sobre recursos não acessíveis a ele (presentes em "C","D" e "E"). Para tratar esta questão o *uOS* se utiliza de duas estratégias que podem ser utilizadas isoladamente ou de maneira conjunta.

4.3.3.3.1 Proxying :

Na estratégia de *proxying*, o dispositivo consultado busca viabilizar a comunicação entre dispositivos que não a possuem de maneira direta no ambiente. Neste caso um dispositivo (normalmente o *register*) ao ser consultado acerca dos recursos que este conhece no ambiente, verifica a conectividade entre o dispositivo que realiza a consulta e os dispositivos que hospedam os recursos. Nos casos em que é verificada a não conectividade entre os dispositivos, o dispositivo consultado informa ao solicitante que ele mesmo é o provedor deste recurso. No futuro, em caso

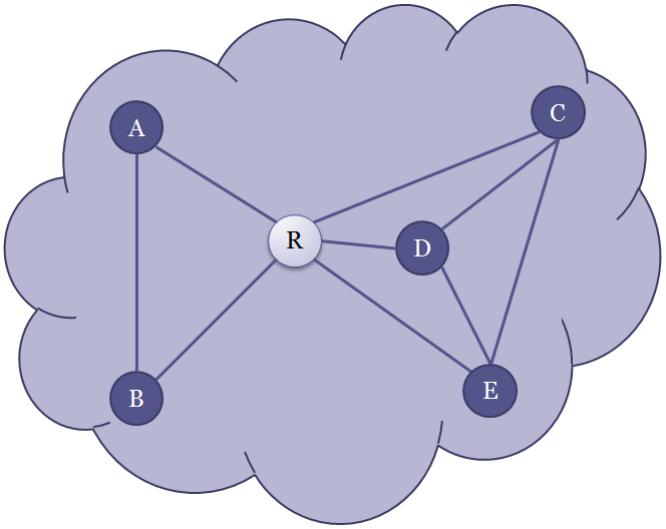


Figura 4.22: Distribuição de dispositivos com visibilidade limitada.

deste recurso ser acessado, as chamadas aos serviços do recurso serão repassadas ao detentor real deste. Por questões de consistência, um dispositivo nunca realiza *proxying* de seus próprios recursos ou dos recursos do dispositivo que solicita as informações.

No exemplo apresentado acima, utilizando-se a técnica de *proxying*, caso "A" solicite a "R" as informações de recursos disponíveis, "A" teria como retorno: Os recursos de "B" e "R" de maneira direta (pois estes são diretamente acessíveis por "A") juntamente de recursos *proxies* de "C","D" e "E" providos de maneira indireta através de "R" (visto que estes não são diretamente acessíveis por "A").

4.3.3.3.2 Filtering :

Caso um dispositivo sendo consultado sobre os recursos do ambiente retorne informações sobre dispositivos fora de alcance, teremos, além do tráfego de informações desnecessárias, a possibilidade de iniciar um processo de tentativa e erro. Em muitos casos pode ser aceitável, mas normalmente isto viola a transparência proposta pela *ubicomp*. Por isso a estratégia de *filtering* propõe que o dispositivo sendo consultado retorne apenas as informações acerca de dispositivos que estejam ao alcance do dispositivo solicitante. Evitando assim o tráfego desnecessário de informações no ambiente.

Utilizando-se a técnica de *filtering* no exemplo anteriormente citado, caso "A" solicite a "R" as informações de recursos disponíveis, "A" teria como retorno: Os recursos de "B" e "R" de maneira direta (pois estes são diretamente acessíveis por "A"), porém nenhum recurso provido por "C","D" ou "E" será retornado (visto que estes não são diretamente acessíveis por "A").

4.3.4 Construindo soluções

O *uOS* implementa uma infra-estrutura para lidar com a gestão dos recursos de um dispositivo e permitir que este acesse os recursos presentes em outros dispositivos no ambiente. Para que estas funcionalidades encontrem propósito, *drivers* de recursos e aplicativos devem estar em execução nestes dispositivos, assim como é estabelecido pela *DSOA*. O *uOS* provê meios para que estas entidades sejam integradas ao *middleware* e possuam acesso aos recursos do ambiente. Estes meios foram apresentado inicialmente na sessão 4.3.2.1.3 e agora sua utilização será melhor detalhada.

4.3.4.1 Drivers

Como foi visto na figura 4.20, todo *driver* deve implementar a interface *UosDriver* (Listagem B.1). Esta interface define três métodos que devem ser implementados pelos *drivers* de recurso:

- *UpDriver getDriver()* : Método responsável por retornar um descritor da interface do recurso em formato *uP*. Este descritor conterá as informações do nome do *driver*, seus serviços e eventos de acordo com a representação definida pelo *uP*.
- *void init(UOSApplicationContext applicationContext)* : Responsável por realizar a inicialização do *driver*. Neste método é informado o objeto *UOSApplicationContext* responsável por fornecer acesso aos recursos do *middleware* ao *driver*.
- *void tearDown()* : Durante o processo de finalização do *middleware* dentro do dispositivo, este método será invocado afim de o *driver* liberar os recursos por ele alocados.

Além destes métodos, todo serviço declarado pelo *driver* deve estar na forma apresentada na listagem B.2. O nome do método ("serviceName") deve coincidir com o nome do serviço declarado na interface do recurso. Isto deve ser realizado, pois o acesso ao serviço é realizado utilizando a *Reflections API* fornecida pelo *Java*. Os objetos *ServiceCall* e *ServiceResponse* são responsáveis por trafegar as informações de execução do serviço de acordo com o protocolo *SCP* do *uP* (sessão 4.2.5.1). O objeto *UOSMessageContext* carrega informações acerca do dispositivo que invocou a comunicação e dos canais de dados abertos junto ao mesmo.

Na implementação *JME*, devido a ausência da *Reflections API* a interface do *UosDriver* sofre a adição de um método (listagem B.3). O método "*handleServiceCall*" é responsável por tratar todas as execuções de serviço do *driver* invocado baseando-se das informações providas no *ServiceCall*.

Por fim os *drivers* disponíveis em um dispositivo podem ser definidos de duas maneiras. Através da propriedade "*ubiquitos.driver.deploylist*" do arquivo de configuração do *middleware* podem ser definidas as classes dos *drivers* a serem disponibilizados. A listagem B.4 apresenta uma destas declarações. Podemos observar que os *drivers* podem ser declarados informando-se o identificador de

instância a ser utilizado para o *driver*. No caso da declaração do "*EchoDriver*" temos como seu identificador "*pingDriver*". Porém, para *drivers* cujo identificador não for informado, o *middleware* gerará e lhe atribuirá automaticamente um identificador de instância.

Outra forma de se informar *drivers* ao *middleware* é utilizando diretamente o *DriverManager*. Como visto na sessão 4.3.2.1.3, esta classe é responsável por gerenciar os *drivers* disponíveis no dispositivo. Esta classe possui o método "*deployDriver*" responsável por tratar da adição de novos *drivers* junto ao dispositivo. Juntamente temos o método "*undeployDriver*" cuja ação corresponde a remoção de *drivers* do dispositivo.

4.3.4.2 Aplicações

Assim como os *drivers*, aplicações devem seguir uma interface para serem integradas no *middleware*. Conforme foi apresentado figura 4.21 as aplicações devem obedecer a interface *UosApplicationLauncher* (listagem B.5). Esta interface define apenas métodos responsáveis por controlar o ciclo de vida da aplicação:

- *void start(UOSApplicationContext applicationContext)* : Método invocado quando se inicia a execução da aplicação no dispositivo. Cada aplicação é executada em uma *Thread* distinta.
- *void stop()* : Durante o processo de finalização do *middleware* no dispositivo, este método será invocado afim de possibilitar a aplicação a liberação dos recursos por ela alocados.

As aplicações podem ser declaradas ao *middleware* através do arquivo de configurações, utilizando a propriedade "*ubiquitos.application.deploylist*". A listagem B.6 apresenta um exemplo desta abordagem. Podemos declarar aplicações com identificadores de instância específicos (como o "*UosChat*") ou sem especificá-lo (como o "*UVNCApp*"). Neste último caso o *middleware* se encarrega de atribuir um identificador automático a instância.

4.3.4.3 Interface de acesso ao *smart space*

Como foi apresentado, o *UOSApplicationContext* é o objeto responsável por fornecer as aplicações e *drivers* o acesso aos recursos do *middleware*. Uma das principais informações fornecidas por esta entidade é o acesso a *AdaptabilityEngine*. Conforme visto na sessão 4.3.2.1.3 este objeto é responsável por fornecer o acesso aos principais recursos do *smart space*. Isto é feito através dos seguintes métodos presentes nesta classe:

- *callService* : Este método é responsável por realizar a chamada de um serviço em um recurso presente em um dispositivo do *smart space*. São aceitos todos os parâmetros (*ServiceCall*) determinados pelo protocolo *SCP* definido no *uP* e como retorno a resposta (*ServiceResponse*) da execução deste serviço. Ao invocar o método, a instância do *driver* de recurso a ser utilizado pode não ser especificada, neste caso fica a cargo do *middleware*

determinar qual das instâncias disponíveis será utilizada. Caso não seja informado nenhum dispositivo na chamada ao método, a escolha deste fica a cargo do próprio *middleware*.

- *registerForEvent* : Este método permite que uma aplicação (ou *driver*) se registre para o recebimento de notificações de eventos (serviços assíncronos), especificando o identificador do evento, *driver*, instância (opcional) e dispositivo que se deseja registrar.

- *unregisterForEvent* : Este método retira o *driver* ou aplicação do cadastro de recebimento de eventos seguindo os mesmos parâmetros especificados pelo método *registerForEvent*.

- *listDrivers* : Este método retorna todos os *drivers* conhecidos pelo dispositivo. Nesta consulta são considerados tanto os *drivers* de recursos locais ao *middleware*, como *drivers* de recursos de outros dispositivos conhecidos na vizinhança.

- *sendEventNotify* : Este método permite que os *drivers* notifiquem a ocorrência de eventos ao *smart space* de acordo com os registros de aplicações e *drivers* que solicitaram o recebimento dos mesmos.

Através destes métodos, o *middleware* permite que tanto aplicações como *drivers* tenham acesso simplificado aos *drivers* de recursos e seus serviços presentes no *smart space*. Além disto, fornece meios de que estes tomem conhecimento da disponibilidade destes no ambiente.

4.4 Resumo do capítulo

Neste capítulo apresentamos o trabalho desenvolvido em busca de satisfazer os três requisitos destacados no desenvolvimento de aplicações de *ubicomp*. A solução apresentada foi decomposta em três partes. A *DSOA* foi apresentada como uma extensão da arquitetura orientada a serviços provendo um ferramental para se modelar ambientes inteligentes e endereçar as características de comunicação necessárias. O *uP* define um conjunto de protocolos leve e multi plataforma provendo mecanismos de comunicação ao *smart space*. Por fim, o *middleware uOS* apresenta recursos que facilitam a construção dos componentes de *software* necessários ao ambiente, como *drivers* de recurso e aplicações. Estes componentes recebem interfaces de acesso aos ambiente de acordo com os padrões definidos na *DSOA* e as interações do *uP*. Além disto, o *middleware* fornece um mecanismo de *plugins* para a integração de plataformas de comunicação distintas.

Capítulo 5

Resultados experimentais

A *DSOA*, o *uP* e o *uOS* formam um conjunto de soluções no auxílio ao desenvolvimento de aplicações para a computação ubíqua. Cada uma tem seu papel e a integração de suas características contribui para se atingir esse objetivo. A fim de validar estas soluções buscou-se duas abordagens. Primeiramente foi construído um protótipo de uma aplicação seguindo os conceitos propostos pela *DSOA*, denominada de *Hydra*. Tal aplicação tem por objetivo validar o funcionamento deste conjunto de soluções, desde a modelagem do ambiente utilizando a *DSOA*, a comunicação utilizando o *uP* e a construção dos aplicativos e *drivers* utilizando o *uOS*. Para a validação dos critérios de leveza da solução criada foram realizados testes de carga da aplicação bem como realizado um estudo comparativo das versões de distribuição desta com relação a outros *middlewares*.

5.1 A aplicação Hydra

A *Hydra* consiste em uma aplicação construída com o objetivo de explorar a forma como o ambiente é decomposto em recursos e serviços para possibilitar uma forma desagregada de acesso a recursos de entrada e saída dos dispositivos. Seu objetivo é permitir que um determinado dispositivo tenha seus recursos de entrada e saída redirecionados para outros dispositivos, repassando a estes o controle de sua operação. Para ilustrar o uso desta aplicação tomemos o seguinte exemplo:

Considere como *smart space* uma sala de reunião onde temos um *laptop Dell*, um *MacBook*, um celular *Sony Ericson* W580i e uma TV *Sony* 52" de LCD conectada a um *PC*. Estes dispositivos encontram-se modelados conforme representado na figura 5.1.

- O *laptop Dell* e o *MacBook* disponibilizam ao *smart space* os recursos de teclado, *mouse*, *webcam* e saída de vídeo (tela).
- O celular *Sony Ericson* w580i provê os recursos de teclado e *mouse* ao *smart space*.
- O *PC* provê o recurso de saída de vídeo (tela) através da TV *Sony* 52" de LCD.

Figura 5.1: Exemplo de *smart space* utilizado pela aplicação *Hydra*.

Neste *smart space* inicia-se uma reunião contando com a presença de Estevão (dono do *MacBook*), Carla (dona do *laptop Dell*) e Ricardo (no momento com seu celular *w580i*). O objetivo desta reunião é discutir os avanços no desenvolvimento de uma solução sob a responsabilidade da equipe. Como Carla é a responsável pela coordenação do projeto, instalou em seu *laptop* a aplicação *Hydra* com o intuito de facilitar a interação entre os presentes. Cada um dos demais possuem seus dispositivos habilitados no padrão *uP* de acordo com o utilizado pelo *uOS*. Tornando assim todos os recursos nestes dispositivos disponíveis ao ambiente.

Carla inicia a reunião e deseja apresentar um pedaço de código para que todos possam visualizar. Utilizando a aplicação *Hydra*, encontra-se no ambiente três saídas de vídeo disponíveis, o *MacBook*, o *laptop Dell* (utilizado por Carla) e a TV de LCD. Carla escolhe a TV como mais adequada para a tarefa de apresentar sua saída de vídeo, e solicita o redirecionamento para este recurso através da aplicação. Com o avanço das discussões, Ricardo sente a necessidade de apontar algumas questões no código exibido. Ele então solicita a Carla que redirecione o *mouse* para seu celular. Ela então repete o mesmo procedimento e redireciona o recurso de *mouse* para o celular *w580i* de Ricardo. Com as observações apresentadas por Ricardo, a discussão avança até o momento em que Estevão sugere uma possível solução. Porém, apenas verbalmente Estevão não consegue descrever sua proposta. Sendo assim ele solicita a Carla que direcione o recurso de teclado ao seu *MacBook* de maneira que ele possa apresentar sua solução na tela. Carla

utiliza novamente a aplicação para redirecionar este recurso. Ao final, a solução apresentada por Estevão se mostra satisfatória para atender as observações de Ricardo. Decide-se então seguir a estratégia apresentada e finaliza-se a reunião.

Figura 5.2: Configuração final do *smart space* ao final da reunião.

Ao final da reunião a configuração dos recursos no *smart space* se mantém a mesma. Porém, a aplicação *Hydra* proporciona uma visão diferente do *laptop Dell* pertencente a Carla. Conforme representado na figura 5.2, o *laptop* tem seus recursos de entrada e saída direcionados para três outros dispositivos no ambiente. Desta maneira o *laptop* pode ser visto como uma composição de recursos distribuídos no ambiente e não apenas um único dispositivo.

A facilidade de acessar os recursos do ambiente é proporcionada através das interfaces definidas pela *DSOA*. Através da modelagem proposta pela arquitetura, a definição das interfaces dos recursos está apenas relacionada às funcionalidades disponibilizadas. De maneira que a aplicação utiliza-se desta modelagem para prover a facilidade de redirecionamento entre os recursos de entrada e saída.

A utilização do *uP* garante que a implementação dos recursos em cada um dos dispositivos, independente de plataforma, é capaz de interagir sem a necessidade de adaptações. Como os dispositivos possuem interfaces definidas e conhecidas (como estabelecido pela *DSOA*), basta a aplicação *Hydra* conhecer estas interfaces para que o acesso aos recursos se torne possível.

5.1.1 O protótipo

Utilizando-se a idéia apresentada, foi construído um protótipo da aplicação *Hydra*. Este protótipo tem por finalidade validar a modelagem de um ambiente utilizando-se a proposta apresentadas pela *DSOA* bem como a utilização do protocolo *uP* e o *middleware uOS*.

Para este protótipo foi modelada a interface do recurso de *mouse* através de um serviço assíncrono que notifique o cliente sobre as interações do usuário através do mesmo. Esta interface encontra-se definida a seguir:

- Nome do recurso: "br.unb.unbiquitous.ubiquitos.driver.mouse.MouseDriver"
- Serviços:
  - "*registerListener*": Serviço síncrono que recebe como parâmetro a "*eventKey*" do serviço assíncrono ao qual deseja ser notificado.
  - "*unregisterListener*'': Serviço síncrono que recebe como parâmetro a "*eventKey*" do serviço assíncrono ao qual não mais deseja ser notificado.
  - "*mousevent*": Serviço assíncrono que representa a interação de um usuário junto ao *mouse*. Possui dois parâmetros definidos conforme segue:
    - "*mouseCommand*": Comando executado pelo usuário junto ao *mouse*, podendo ser: "*up*", "*down*", "*left*", "*right*", "*left_button_pressed*", "*right_button_pressed*", "*left_button_released*", "*right_button_released*", "*reset_mouse_position*" ou "*center_mouse_position*".
    - "*moveFactor*": Inteiro representando a quantidade de *pixels* em que a posição do *mouse* foi alterada.

Foi realizada a implementação do *MouseDriver* utilizando a versão *JME* do *uOS*. O envio dos eventos capturados pela interface pode ser visualizado na listagem 5.1. Neste código podemos destacar dois pontos:

- No método "*notifyMouseEvent*" é montado o objeto que representa a mensagem de *Notify* utilizada pelo *uP* contendo os dois parâmetros especificados pelo serviço assíncrono "*mouseevent*".
- No método "*notifyRegisteredDevices*" é percorrida a lista de dispositivos registrados para este *driver* e estes são notificados da ocorrência do evento.

```
private void notifyMouseEvent(String command){
    Notify notify = new Notify("mousevent");

    notify.addParameter("mouseCommand", command);
    notify.addParameter("moveFactor", mouseMoveFactor);

    notifyRegisteredDevices(notify);
```

```
}

private void notifyRegisteredDevices(Notify notify) {
    for (int i = 0 ; i < listennerDevices.size(); i++){
        try {
            UpDevice device = (UpDevice)
                listennerDevices.elementAt(i);
            gateway.
                sendEventNotify(notify, device);
        } catch (UosException e) {
            logger.error(e);
        }
    }
}
```

Listagem 5.1: Notificação de eventos no *MouseDriver*

Para a implementação da aplicação *Hydra* foi utilizada a versão *JSE* do *uOS*. Nesta aplicação são verificados os dispositivos no ambiente que possuem instâncias do *driver* de recurso que se deseja (o *MouseDriver*). As instâncias disponíveis são exibidas ao usuário que então seleciona a qual delas deseja redirecionar seu *mouse*. A listagem 5.2 apresenta como é realizada a consulta no ambiente para encontrar as instâncias de *drivers* do recurso de "*MouseDriver*". Nesta implementação a descoberta de novos dispositivos e seus *drivers* de recurso é realizada através dos radares do *uOS*.

```
public List<RemoteDriverData> listMouseDrivers(){
    return applicationContext.
        getDeviceManager().
            listDrivers(null,
    "br.unb.unbiquitous.ubiquitos.driver.mouse.MouseDriver",
                null);
}
```

Listagem 5.2: Busca de instâncias do *MouseDriver* no ambiente.

```
public void registerForDriver(RemoteDriverData driverData)
    throws NotifyException{
        applicationContext.
            getAdaptabilityEngine().
                registerForEvent(this,
                    driverData.getDevice(),
                    driverData.getDriver().getName(),
                    "mouseevent");
}
```

Listagem 5.3: Registro para acesso ao serviço de "*mouseevent*" em uma instância

do *MouseDriver* escolhida.

Já na listagem 5.3 temos como é realizada a solicitação de registro no serviço assíncrono "*mouseevent*" utilizando a *AdaptabilityEngine* e informando a própria aplicação como *listener* do evento (parâmetro "*this*" na linha 5). Escolhida a instância que se deseja registrar e devidamente registrada, basta à aplicação aguardar a notificação do evento. Para tal, a aplicação utiliza o método "*handleEvent*" da interface do *UosEventListener* conforme apresentado na listagem 5.4.

```
public void handleEvent(Notify event) {
    if (event.getEventKey().equals("mouseevent")){
        String command = event.getParameter("mouseCommand");
        int moveFactor = 1; // default value
        moveFactor = Integer.
            parseInt(event.getParameter("moveFactor"));

        /** Handle event according to parameters **/
    }
}
```

Listagem 5.4: Método responsável por tratar as notificações de evento recebidas.

5.1.2 Análise da solução

Neste protótipo é possível observar características da *DSOA*, do *uP* e do *uOS* na construção de uma solução em um ambiente de computação ubíqua. A utilização da *DSOA* é vista no primeiro passo da construção da solução. Primeiramente mapeamos o *smart space*, seus dispositivos e recursos. Para o recurso *mouse* foi definida uma interface que expresse como este provê suas funcionalidades no ambiente. Podemos ver no protótipo que o *MouseDriver* foi modelado com base nos serviços que ele pode prover no ambiente e não em como a aplicação *Hydra* opera. Isolando assim, a modelagem do ambiente da construção da solução.

Este tipo de abordagem permite tirar proveito de uma arquitetura orientada a seviços, onde as capacidades (neste caso os serviços dos recursos) estão disponíveis de maneira unificada as aplicações. Ao mesmo tempo a utilização de recursos provê coesão entre os serviços com a adição de apenas uma camada de abstração (os *drivers*), solução mais simples do que a proposta do MPACC [54] (*Model Presentation Adapter Controller Coordinator*) utilizado no projeto Gaia, que além de estabelecer dois novos níveis de abstração limita o modelo de interação entre aplicações e recursos.

Definida a interface do *MouseDriver*, esta foi implementada utilizando o *uOS* e os protocolos *uP*. Fica clara a influência dos protocolos na forma como o *driver* foi construído utilizando os objetos que representam as mensagens do protocolo. Pode ser observada também que o *driver*, desenvolvido utilizando o *uOS*, não trata detalhes da plataforma de comunicação, sendo isto transparente através do uso do *middleware*.

Na construção da aplicação vemos que o acesso aos protocolos complementares do *uP* são abstraídos pelo *middleware*, como visto na chamada ao método "*listDrivers*". Como os estes protocolos são acessíveis através de serviços síncronos, temos um exemplo de como é realizado o acesso a estes pelo *middleware*. É visto também como uma solicitação de serviços assíncronos opera desde seu registro até a notificação de eventos a aplicação.

5.2 Testes de Carga

O conjunto de soluções formado pelo *uP* e o *uOS* tem por objetivo não limitar os dispositivos que possam acessar diretamente o *smart space*. O *uP* endereça a questão de suporte multi-plataforma ao utilizar um formato estruturado (*JSON*) e a codificação *UTF-8*. Porém, uma limitação importante para muitos dispositivos está na quantidade de poder computacional disponível. Caso a carga proporcionada pelo conjunto protocolo-*middleware* seja impactante, quando comparada ao tempo de comunicação de rede, o uso desta solução pode inviabilizar a aplicação almejada.

A fim de validar a carga proporcionada pelo conjunto *uP-uOS* foram realizadas baterias de testes para medir o tempo gasto pela aplicação para tratar uma chamada de serviço. Para tal tarefa estes testes mediram o tempo gasto entre a chamada do serviço (*AdaptabilityEngine.callService*) e o envio da mensagem para a interface de rede. A medida de tempo foi realizada utilizando a chamada *Java* "*System.nanoTime()*" que fornece o tempo com precisão em nano segundos, através de chamadas ao sistema operacional. Todo o controle dos tempos de início e fim do tratamento foram realizados utilizando armazenamento em memória a fim de minimizar os impactos ao se utilizar *buffers* de escrita (consoles, arquivos ou *logs*). Ao fim das baterias de teste, os dados eram extraídos da memória e compilados em arquivos contendo os dados experimentais.

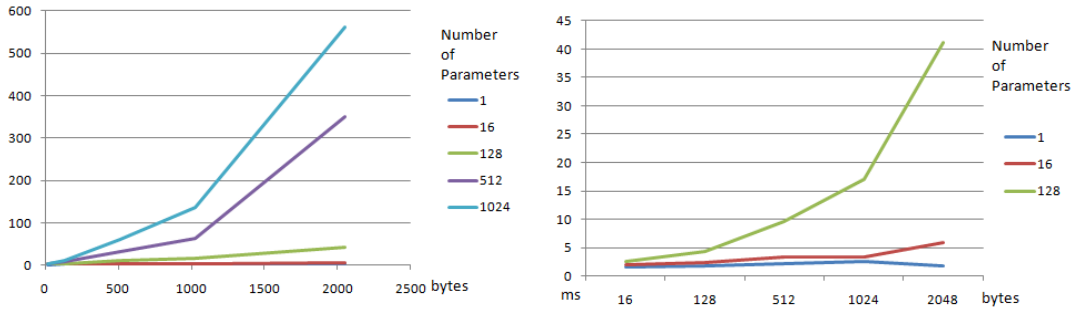
Para a realização dos testes foram realizadas 25 baterias de testes. Cada bateria foi composta por 1024 chamadas de serviço variando-se dois parâmetros na execução de cada bateria: O número de parâmetros e o tamanho dos parâmetros. Para os números de parâmetros foram utilizados os valores 1, 16, 128, 512 e 1024 campos. Para os tamanhos dos parâmetros foram utilizados os valores de 16, 128, 512, 1024 e 2048 bytes. Isto forneceu uma variação dos tamanhos de mensagens entre 195 bytes e 7269448 bytes (6,93 MB). Os testes foram realizados em um Dell Vostro 1500, Intel(R) Core(TM)2 Duo CPU T7500 @ 2.20GHz 2x2 GB DD2 333MHz , Windows Vista SP2.

A tabela 5.1 apresenta os resultados obtidos. Estes resultados são bastante compatíveis com o que é apresentado nos estudos realizados em [16]. A figura 5.3(a) apresenta os gráfico resultante dos dados obtidos. Podemos observar que a tendência é linear para o número de parâmetros variando entre 1 e 16 (figura 5.3(b)), porém o crescimento aparenta exponencial para um número de parâmetros superior a 128. Como as mensagens do *uP* são utilizadas para o acesso a serviços são discretas[1] estima-se que a maior parte do tráfego gire em torno de 1

[1]Mesmo serviços contínuos possuem sua solicitações realizadas no canal de controle de maneira discreta.

NP x BP*	16	128	512	1024	2048
1	1,524 ms	1,876 ms	2,126 ms	2,301 ms	2,524 ms
16	1,957 ms	2,369 ms	3,380 ms	3,313 ms	5,840 ms
128	2,530 ms	4,299 ms	9,665 ms	17,050 ms	41,135 ms
512	3,962 ms	7,580 ms	32,148 ms	63,395 ms	349,411 ms
1024	3,755 ms	11,951 ms	61,145 ms	135,668 ms	561,951 ms
NP : Número de parâmetros (vertical). BP : Bytes por parâmetros (horizontal).					

Tabela 5.1: Resultados dos testes de tempo de carga utilizando o *uP* e *uOS*.



(a) Tempo de carga do *uP-uOS* de acordo com o número de parâmetros e tamanho dos parâmetros.

(b) Detalhe do teste de carga para a variação do número de parâmetros de 1, 16 e 128

a 16 possuindo parâmetros com tamanhos entre 16 e 128 bytes. Tendo isso como base a carga média estimada para os acesso usando o *uP-uOS* é 1,935203 ms.

Foram escolhidos para o estudo comparativo destes resultados os *middlewares* *MundoCore* e *WSAMI* além do projeto *Home SOA*. O *middleware* *MudoCore* foi escolhido devido ao seu foco em plataformas limitadas. O *middleware* se encontra disponível em três versões distintas (Java, Phyton e C++) e possibilita operar em dois modos distintos para a serialização de objetos (modo binário e utilizando descritores *XML*). Dentre os resultados apresentados em [56] podemos utilizar para comparação aqueles que envolvem a configuração *Java+XML*, que se encontra em 2,67 ms. Vale destacar que o melhor resultado obtido pelo *MundoCore* ocorre na configuração *Cpp+Binário* onde o tempo de carga fica em 0,50 ms.

O *WSAMI* é desenvolvido utilizando a plataforma *Java* e o protocolo *SOAP* para a comunicação. Não foram encontrados estudos de desempenho no *WSAMI*, porém podemos tomar como base o desempenho de outras implementações *SOAP* [29] para este comparativo. Tais soluções apresentam como resultado para a plataforma *Windows+Java* em 4,70 ms e os melhor resultado na plataforma *Linux+Cpp* com 0,52 ms.

Outro projeto relevante neste ponto é o projeto *Home SOA* [8]. Este projeto se baseia na montagem de um ambiente reconfigurável utilizando os conceitos de *SOA* e o *middleware OSGi*. Nos testes apresentados o tempo de carga apresentado por esta solução fica em 5,23 ms.

Na figura 5.3 podemos observar o comparativo dos resultados encontrados no *uOS* com relação ao encontrado em outros *middlewares*. A fim de observar o

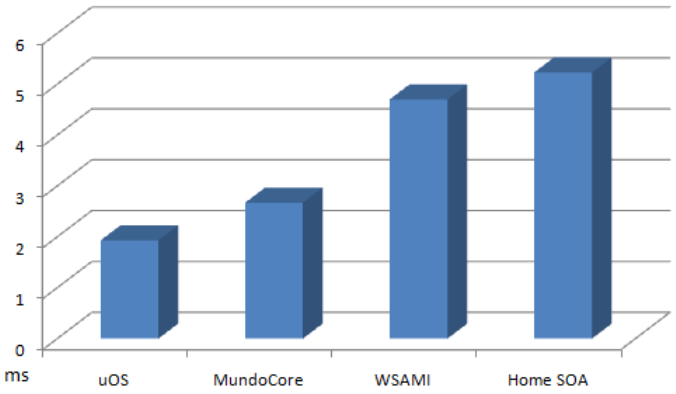


Figura 5.3: Sobrecarga de comunicação nos *middlewares* analisados.

Transmissão	Velocidade	Tempo de transferência	Percentual de Sobrecarga
ZigBee	250 Kbit/s	254464 ms	0,0008%
Bluetooth 1.2	1 Mbit/s	20976 ms	0,0092%
Bluetooth 2.0	3 Mbit/s	6992 ms	0,0276%
802.11 (Wi-Fi)*	54 Mbit/s	399,6 ms	0,4829%
Ethernet*	100 Mbit/s	206,2 ms	0,8943%
GigE*	1000 Mbit/s	20,62 ms	8,9435%
* Utilizando TCP.			

Tabela 5.2: Tempo de carga na utilização da solução *uP-uOS* de acordo com as velocidades máximas de cada tecnologia de comunicação.

percentual gasto pelo *uOS* com relação ao tempo de comunicação, foi elaborada a tabela 5.2. Nesta tabela observamos algumas tecnologias de rede, o tempo mínimo que estas demandariam em uma mensagem média do *uOS* e o percentual de tempo que o *uOS* acrescenta nesta comunicação. Vemos nesta tabela que para a maioria das tecnologias a solução se mostra um baixo impacto com relação ao tempo gasto na comunicação. Apenas na comunicação em *Gigabit Ethernet* é que temos um impacto considerável. Mesmo assim, nos demais resultados o uso do *uP-uOS* se mostra menos degradante que outras soluções, apresentando uma carga inferior a 1% , ao passo que a *Home SOA* apresenta uma carga de 3,30 %.

5.3 Carga de Empacotamento

Middlewares visam facilitar a construção de aplicações e, como visto na seção anterior, estas facilidade possuem um custo de tempo na solução desenvolvida. Outro custo relevante neste sentido é o tamanho da distribuição (pacote). A solução composta pelas aplicações, pelo *middleware* e outros componentes correlatos (*drivers*, bibliotecas, etc.) demanda espaço de armazenamento tanto em memória primária como secundária. Sendo o espaço de armazenamento um insumo racionado em diversos dispositivos, esta é uma variável que foi considerada

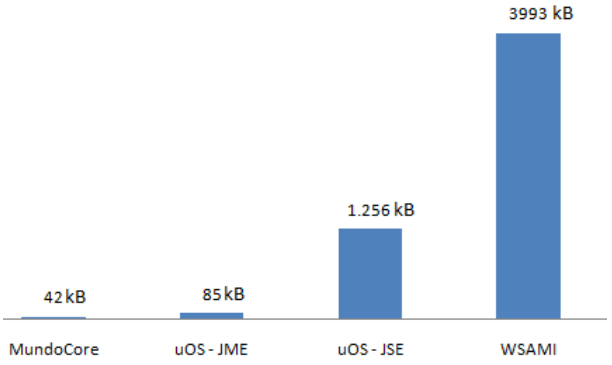


Figura 5.4: Tamanho de empacotamento dos *middlewares* analisados.

neste trabalho.

Como vimos, o *uOS* se encontra distribuído em duas versões distintas. A versão *JSE* almeja em dispositivos mais robustos e conta com uma implementação que se utiliza de maiores facilidades para o desenvolvedor (como o uso de *annotations*, *generics* e *reflections*) providas pela linguagem *Java*. Esta versão se encontra disponível em um pacote de 1.256 KB. Neste empacotamento, 114,9 KB correspondem a binários responsáveis por tratar as mensagens no formato *uP*. Observando as limitações da linguagem *Java* em sua versão *Micro Edition*, a versão *JME* do *uOS* é disponibilizada em um pacote de 85 KB. O tratamento das mensagens *uP* consome 61,7 KB deste empacotamento.

A figura 5.4 apresenta a relação entre o tamanho dos pacotes fornecidos por outros *middlewares* com relação ao encontrado no *uOS*. O projeto *MundoCore* fornece uma distribuição de 42 KB que contempla tudo necessário para que a aplicação realize a distribuição e tratamento da serialização de objetos no ambiente. Já o *WSAMI* é distribuído em um pacote de 3.9 MB que contempla os núcleos de tratamento das mensagens *SOAP* bem como o servidor de aplicação necessário para os serviços *HTTP*. Vemos então que o *uOS JME* apresenta um bom resultado para dispositivos limitados, porém ainda inferior ao encontrado no *MundoCore*. Com relação aos dispositivos mais robustos, a arquitetura do *uOS JSE* se mostra mais leve do que o a solução *WSAMI*.

Seguindo os conceitos da *DSOA* e as definições do *uP*, o *uOS* mapeia os recursos e suas interfaces em *drivers*. A implementação dos *drivers* é realizada em classes *Java* que especificam o comportamento para cada serviço prestado pelo recurso. Já as interfaces são representadas no padrão especificado pelo *uP* em formato *JSON*. Tomando como exemplo o *MouseDriver* temos que sua implementação ocupa 7.751 bytes e sua interface 233 bytes. Observando a solução *Home SOA* temos que o melhor resultado obtido neste sentido está na implementação do *driver* para o protocolo *X10* que ocupa 50 KB com o *base driver* (implementação) e 20 KB com o *refined driver* (interface). A tabela 5.3 sumariza estes resultados. Vemos que o *driver* implementado utilizando o *uOS* corresponde a 19,49% da solução no *Home SOA*.

Solução	Driver (Implementação)	Interface	Total
uOS	7.751 bytes	233 bytes	7.984 bytes
Home SOA	51.200 bytes	20.480 bytes	40.960 bytes

Tabela 5.3: Comparativo do empacotamento para distribuição de *drivers*.

5.4 Resumo do capítulo

Neste capítulo apresentamos a validação do conjunto de soluções apresentado. Uma das validações foi apresentada na forma da aplicação *Hydra* que destacou os benefícios da modelagem da *DSOA* e as facilidades no desenvolvimento utilizando o suporte fornecido pelo *uOS*. Foram apresentados também um conjunto de testes quantitativos para a análise dos critérios de limitação da solução *uP/uOS* onde estes resultados foram comparados a outras soluções existente chegando resultados satisfatórios.

Capítulo 6

Conclusão e trabalhos futuros

Dados do *ITU* [61] apontam que o acesso à telefonia celular já chega a 49% da população mundial. No Brasil este número saltou de 26,6% em 2003 para 78,47% em 2008. Além disto, a cada dia a realidade de um ambiente cercado por dispositivos computacionais mais se aproxima do cotidiano. Tênis [18], roupas [60] [46], jóias [47] e muitos outros elementos que tradicionalmente não apresentavam a necessidade de serem incrementados de aparatos eletrônicos começam a violar esta premissa. Tudo isso colabora com a visão que [63] expôs junto aos conceitos que apresentaram a computação ubíqua ao mundo.

É nesta realidade fértil que os elementos necessários para a *ubicomp* se encontram. Um número crescente e variado de dispositivos presentes nos mais diversos tipos de ambientes, com capacidade de processamento e comunicação. Conforme apontado neste trabalho, as aplicações são de grande importância para o *smart space* e existem diversas iniciativas em busca de facilitar seu desenvolvimento. Com base no que foi apresentado na literatura e os desafios encontrados no desenvolvimento da computação ubíqua chegamos aos três requisitos alvo deste trabalho.

- Tratar a **heterogeneidade** das plataformas dos dispositivos presentes no ambiente inteligente.
- Fornecer suporte a **dispositivos limitados** aumentando a diversidade de componentes e recursos disponíveis as aplicações e usuários.
- Prover mecanismos de tratamento dos detalhes de comunicação característicos de ambientes de *ubicomp* permitindo uma garantia da satisfação destas características no ambiente.

A tabela 6.1 sumariza como as principais abordagens referenciam estes requisitos. Nela podemos observar que o *middleware uOS* atende a todos conforme apresentado ao longo do trabalho. Com relação ao suporte a dispositivos, o WSAMI atende a este requisito de maneira parcial tendo em vista que a solução se restringe a dispositivos com limitação apenas de conectividade e com seus testes sendo realizados em *palms*. Com relação ao tratamento de comunicação a maioria das abordagens não endereça todas as características apresentadas, favorecendo umas em detrimento de outras.

Projeto	Suporte a dispositivos limitados	Tratamento da comunicação	Heterogeneidade
Aura	Não	Parcial	Não
Gaia	Não	Parcial	Não
Gator Tech	Não	Parcial	Não
MediaBroker	Não	Parcial	Não
WSAMI	Parcial	Parcial	Sim
MundoCore	Sim	Parcial	Sim
MoCA	Sim	Parcial	Não
EasyLiving	Não	Parcial	Sim
HomeSOA	Sim	Parcial	Não
UbiquitOS	Sim	Parcial	Não
uOS	**Sim**	**Sim**	**Sim**

Tabela 6.1: Comparativo entre os *middlewares*.

Com base nestes três pontos centrais foi elaborada a arquitetura *DSOA*, que embasada nos princípios da *SOA*, apresenta uma forma de se organizar o ambiente e seus detalhes de interação. Seguindo os princípios desta arquitetura foi definido o *uP*, um conjunto de protocolos para a comunicação dos serviços no ambiente de maneira leve e interoperável entre diversas plataformas. O *uOS* foi então desenvolvido em duas plataformas distintas, o (*JME* e o *JSE*), utilizando-se das outras soluções apresentadas. Tal *middleware* forneceu insumos para se validar as características almejadas pela arquitetura e o conjunto de protocolos. Por fim foi elaborado um protótipo a fim de se validar este conjunto de soluções apresentadas neste trabalho.

6.1 Resultados

6.1.1 DSOA

Uma arquitetura visa prover uma visão de como se organizar as soluções para um determinado problema. Seu objetivo é falar"como" sem falar "com o quê". É neste sentido que a *DSOA* se apresenta frente aos ambientes inteligentes, estabelecendo uma maneira de se endereçar as questões específicas deste cenário. Esta arquitetura define os conceitos necessários para se modelar um ambiente inteligente de maneira coesa e desacoplada, simplificando o desenvolvimento de aplicações para *smart spaces*. Juntamente a arquitetura define estratégias a serem seguidas para o tratamento adequado dos detalhes de interação encontrados neste tipo de ambiente. Satisfazendo um dos requisitos importantes para a *ubicomp* e alvo deste trabalho.

6.1.2 *uP*

O segredo para a boa aplicabilidade de um protocolo é a sua simplicidade, sendo este um fator chave para garantir uma fácil adaptação aos diversos cenários onde se busca utilizá-lo. O *uP* foi apresentado como um protocolo simples, seguindo a linha proposta pelo *SLP*. A base do protocolo é construída sobre dois protocolos e três tipos de mensagens responsáveis por definir toda a comunicação prevista pela *DSOA*. É em cima desta base que foi construído outro conjunto de protocolos que complementam os requisitos de descoberta e troca de informações no *smart space*. Esta abordagem permite ao protocolo incorporar novas funcionalidades e aprimoramentos as já existentes sem impactos em implementações vigentes.

Um aspecto a ser destacado no *uP* é a escolha do formato *JSON* para suas mensagens. Este tipo de formatação é importante em três sentidos:

- Por ser um formato estruturado e auto-descritivo, o uso do *JSON* permite uma integração mais simples entre aplicações distintas. O fato de ser estruturado também colabora como um facilitador para ajustes e expansões nos padrões de mensagens estabelecidos sem grandes limitações.

- Um dos formatos estruturados mais conhecidos é o formato *XML*. Porém, conforme foi apresentado, o formato *JSON* se mostra como uma solução mais leve neste sentido. Dados formatados utilizando o *JSON* acabam por resultar em mensagens menores e que demandam menos poder de processamento durante seu tratamento. Fator chave ao se considerar o requisito de atender dispositivos limitados.

- O uso da codificação *UTF-8* garante que o formato *JSON* seja compatível com uma grande variedade de plataformas de *hardware* e *software* distintas.

6.1.3 *uOS*

O objetivo de um *middleware* é simplificar o acesso das aplicações às capacidades fornecidas pelo ambiente. Embasado nas orientações da *DSOA* e utilizando o *uP* como protocolo de comunicação, o *uOS* disponibiliza meios para o desenvolvimento do ecossistema de um *smart space* de maneira simplificada. O *middleware* disponibiliza uma implementação que fornece acesso transparente aos recursos do ambiente provendo interfaces para a construção de aplicações e *drivers* de recursos respeitando os princípios da *DSOA*.

O *middleware* provê meios que, em conjunto com o *uP*, permitem que os dispositivos se organizem de maneira centralizada, distribuída ou híbrida. A configuração do ambiente pode ser espontânea (utilizando-se dos radares e *registers*) ou estática. Os mecanismos de *proxying* e *filtering* asseguram o tratamento de problemas decorrido das questões de conectividade no ambiente. Por fim são providas implementações padrão do *Register Driver* e *Device Driver* contemplando a implementação de um protocolo de autenticação baseando-se nos recursos providos pelo *uP*.

Por fim o *middleware* foi utilizado como base para a validação da arquitetura e do protocolo em um protótipo funcional. Neste desenvolvimento ficaram expostas

as vantagens na utilização destas soluções em um ambiente de computação ubíqua. A aplicação Hydra mostrou que a modelagem de recursos com base nas capacidades que eles tem a oferecer não influi naqueles que usufruem deles, diminuindo o acoplamento entre serviços e aplicações. Complementarmente foram apresentados os resultados de testes utilizando-se o *uOS* onde os valores encontrados se mostraram em sintonia com os requisitos estabelecidos neste trabalho além de favoráveis frente a outras soluções discutidas.

6.2 Trabalhos futuros

Observando-se o projeto que inspirou este trabalho [24], os três pontos pendentes listados foram abordados neste trabalho. Dependência de plataforma, interconectividade reduzida e protocolos não tolerantes a falhas foram pontos que podem ser observados ao longo das características destacadas no conjunto de soluções desenvolvidas. Porém estas soluções abrem novas ramificações de trabalhos a serem desenvolvidos a fim de se chegar a um melhor cenário em *smart spaces*. Dentre estes ramos podemos destacar alguns.

6.2.1 Definir um modelo de dependência entre recursos

A definição de recursos como composição de serviços possibilita a *DSOA* prover um cenário mais coeso [58] com relação as capacidades do ambiente. Porém, nesta abordagem não foram tratadas as áreas de intersecção entre os recursos do ambiente. Como sabemos podemos ter um conjunto de recursos correlatos que podem se apresentar equivalentes em determinados cenários.

Um exemplo de ocorrência deste caso ocorre com o recurso de *mouse*. Um *mouse* "padrão" possui apenas uma interface que define a interação de um ponteiro (cima, baixo, direita, esquerda) e dois botões (direito e esquerdo). Mas este recurso pode encontrar diversas outras características adicionais (múltiplos *scrolls*, um terceiro botão, um sensor gestual) que podem ser almejadas por aplicações no ambiente.

Para solucionar este caso podemos recorrer tanto ao estabelecimento de um modelo hierárquico de recursos no ambiente (assim como proposto pelo modelo de *drivers* nos sistemas operacionais) ou uma abordagem baseada em ontologias.

6.2.2 Composição de *smart spaces*

Como bem sabemos, a *ubicomp* não define uma limitação para os ambiente inteligentes podendo estes serem tanto uma sala de estar ou uma cidade inteira [31]. A definição apresentada pela *DSOA* estabelece o conceito de *smart space* porém não define o que ocorre na existência de múltiplos destes.

A ocorrência de múltiplos *smart spaces* apresenta toda uma gama de interações que devem ser analisadas. Tais ambientes podem ser hierárquicos (uma sala de estar pertence a uma casa), com zonas de intersecção (parte de uma lanchonete faz parte do corredor da escola) ou desconexos entre si. Como tratar a relação

entre os recursos entre estes ambientes em cada caso, bem como a identificação destes ambientes é uma tarefa necessária a ser estudada.

6.2.3 Protocolo de descoberta de dispositivos

O *uP* oferece mecanismos para que os dispositivos encontrem os dados sobre os recursos disponíveis no ambiente. Porém, antes de se trocar alguma informação tais dispositivos devem se encontrar no ambiente. Cada tipo de enlace possui abordagens distintas para solucionar esta questão. Delegar a responsabilidade de descoberta a cada tipo de comunicação utilizada não se mostra adequado quando se deseja ser independente de plataforma. Por isso a definição de um protocolo como o apresentado em [37], proporcionaria uma solução eficiente e independente de plataforma. Além disto, a utilização deste tipo de solução permite o estabelecimento de uma base de conhecimento sobre a rede de conexões no ambiente. Esta informação se mostra de grande valia na tomada de decisões sobre conectividade e *QoS* no *smart space*.

6.2.4 Evolução no tratamento da segurança

O *uP* apresenta a possibilidade de se estabelecer contextos de segurança entre dispositivos no *smart space*. Durante o desenvolvimento deste trabalho ocorreu a elaboração de um protocolo de segurança que se utiliza desse mecanismo [53]. Os trabalhos com segurança devem extrapolar os limites da interação entre dois dispositivos e se basear em contextos maiores.

Em um *smart space* de larga escala, o estabelecimento de chaves compartilhadas entre cada dispositivo se mostra um processo proibitivo. Um desafio neste cenário está em se utilizar de soluções como hierarquias de distribuição de chaves ou redes de confiança para estabelecer os canais seguros no ambiente se atentando aos princípios defendidos pela *ubicomp*.

6.2.5 Aplicação *Hydra*

A aplicação *Hydra* apresenta uma forma desagregada de compor dispositivos no ambiente. Sua implementação neste trabalho se limitou a um protótipo composto por apenas um tipo de recurso. Expandir esta aplicação definindo e abrangendo outros recursos, observando o comportamento conforme as possibilidades de interação ocorrem é de grande valia na validação da abordagem seguida.

6.2.6 Otimização do Empacotamento

Os resultados de empacotamento do *uOS* se mostraram satisfatórios na plataforma *JSE* (se comparada a outras abordagens mais robustas) porém na sua versão *JME* seu resultado ainda poderia ser otimizado. Seu resultado ainda é duas vezes superior ao apresentado pelo *MundoCore*. Além disto, na plataforma *Java Mobile* estar abaixo do limite dos 64kb é exigido em alguns *profiles*. Neste sentido a otimização do tratamento de mensagens bem como o uso de ofuscamento pode ser utilizado..

6.2.7 Desenvolvimento de aplicações

Este projeto apresenta um conjunto de soluções para o desenvolvimento de aplicações para o contexto de computação ubíqua. Seria interessante tentar aplicar estas soluções no desenvolvimento de outros tipos de aplicações, para avaliar a sua flexibilidade e potencial de aplicação.

Apêndice A

Exemplos *uP*

```
{
  "name":"Pah_w580i",
  "networks":[
        {"Bluetooth":"001F81000250"}
  ]
}
```

Listagem A.1: Exemplo de uma representação de um *Device* no *uP*.

```
{
  "name":"br.unb.ubiquitos.webcam.ns60",
  "services":[... <Service Objects> ...],
  "events":[... <Service Objects> ...]
}
```

Listagem A.2: Exemplo de uma representação de um *driver* no *uP*.

```
{
  "name":"snapshot",
  "parameters":{
    "width":"MANDATORY",
    "height":"MANDATORY",
    "encoding":"OPTIONAL"
 }
}
```

Listagem A.3: Exemplo de uma representação de um *Serviço* no *uP*.

```
{
  "name":"br.unb.ubiquitos.webcam.ns60",
  "services":[
    {
```

```
        "name":"snapshot",
        "parameters":{
          "width":"MANDATORY",
          "height":"MANDATORY",
          "encoding":"OPTIONAL"
        }
      },
      {
        "name":"videoStream",
        "parameters":{
          "width":"MANDATORY",
          "height":"MANDATORY",
          "compression":"MANDATORY",
          "framerate":"OPTIONAL"
        }
      }
    ]
  "events":[
      {
        "name":"motionDetection",
        "parameters":{
          "period_start":"OPTIONAL",
          "period_end":"OPTIONAL",
          "sleep_time":"OPTIONAL"
        }
      }
}
```

Listagem A.4: Exemplo da interface de um *driver* e seus *Serviços* no *uP*.

```
{
  "type":"<Tipo da Mensagem>"
  ... Demais Propriedades ...
}
```

Listagem A.5: Formato base da uma mensagem no *uP*.

```
{
    "type":"SERVICE_CALL_REQUEST",
    "driver":"br.unb.ubiquitos.webcam.ns60",
    "instanceId":"ns60_front",
    "service":"snapshot",
    "channelType":"TCP",
    "channelIDs":["14985"],
    "parameters":
      {
```

```
            "width":"320",
            "height":"240"
        }
}
```

Listagem A.6: Exemplo da uma mensagem de *Service Call*.

```
{
    "type":"SERVICE_CALL_RESPONSE",
    "responseData":
        {
            "supportedResolutions":["320x240","800x600"],
            "supportedFormats":["divx","h264"]
        }
}
```

Listagem A.7: Exemplo da uma mensagem de *Service Call*.

```
{
    "type":"NOTIFY",
    "eventKey":"limitReach",
    "driver":"br.unb.ubiquitos.sensor.TermalSensor",
    "instanceId":"TermalSensor_1",
    "parameters":
        {
            "boundReached":"gt25",
            "currentTemperature":"25.15",
            "unit":"CELSIUS"
        }
}
```

Listagem A.8: Exemplo da uma mensagem de *Service Notify*.

Apêndice B

Fontes *uOS*

```
public interface UosDriver {
        public UpDriver getDriver();
        public void init(UOSApplicationContext
                         applicationContext);
        public void tearDown();
}
```

Listagem B.1: Interface *UosDriver*.

```
public void serviceName(ServiceCall serviceCall,
                ServiceResponse serviceResponse,
                UOSMessageContext messageContext);
```

Listagem B.2: Assinatura de um serviço de um *driver*.

```
public void handleServiceCall(ServiceCall serviceCall,
                ServiceResponse serviceResponse,
                NetworkDevice networkDevice);
```

Listagem B.3: Assinatura do método responsável pelo tratamento de serviços de um *driver* em *JME*.

```
ubiquitos.driver.deploylist=
        br.uos.driver.DeviceDriverImpl;\
        br.ubiquitos.context.EchoDriver(pingDriver);\
        br.ubiquitos.context.VGADriver(monitorSamsung);\
        br.ubiquitos.context.KeyboardDriver
```

Listagem B.4: Declaração de *drivers* no arquivo de configuração.

```
public interface UosApplicationLauncher {
        public void start(UOSApplicationContext
                applicationContext);
        public void stop() throws Exception;
```

Listagem B.5: Interface *UosApplicationLauncher*.

```
ubiquitos.application.deploylist=
        br.ubiquitos.applications.Ping(Ping);\
        br.ubiquitos.applications.UosChat(Chat);\
        br.ubiquitos.applications.UVNCApp
```

Listagem B.6: Declaração de aplicações no arquivo de configuração.

Referências Bibliográficas

[1] Gregory Abowd, Chris Atkeson, and Irfan Essa. Ubiquitous smart spaces. A white paper submitted to DARPA (in response to RFI), 1998.

[2] Erwin Aitenbichler. Mundo. Technical report, Darmstadt University of Technology, 2008. Diponível em `https://leda.tk.informatik.tu-darmstadt.de/cgi-bin/twiki/view/Mundo/WebHome` (acessado em 23/11/2008).

[3] Erwin Aitenbichler and Max Mühlhäuser. The talking assistant headset: A novel terminal for ubiquitous computing. Technical report, In Microsoft Summer Research Workshop, 2002.

[4] ZigBee Alliance. Zigbee alliance. Diponível em `http://www.zigbee.org` (acessado em 03/11/2008).

[5] Erwin Aitenbichler and. Development tools for mundo smart environments. In Gerd Kortuem, editor, *Workshop on Software Engineering Challenges for Ubiquitous Computing*, pages 89–90, Lancaster University, 2006.

[6] Philip A. Bernstein. Middleware: a model for distributed system services. *Communications of the ACM*, 39(2):86–98, 1996.

[7] IEEE-SA Standards Board. 802.16 - ieee standard for local and metropolitan area networks. Technical report, IEEE Computer Society, 2004. Diponível em `http://standards.ieee.org/getieee802/download/802.16-2004.pdf` (acessado em 31/03/2010).

[8] André Bottaro and Anne Gérodolle. Home soa : facing protocol heterogeneity in pervasive applications. In *ICPS '08: Proceedings of the 5th international conference on Pervasive services*, pages 73–80, New York, NY, USA, 2008. ACM.

[9] Mark Weiser; John Seely Brow. Designing calm technology. Technical report, Xerox PARC, 1995. Diponível em `http://nano.xerox.com/weiser/calmtech/calmtech.htm` (acessado em 02/11/2008).

[10] Vagner Sacramento; Markus Endler; Hana K. Rubinsztejn; Luciana S. Lima; Kleder Gonçalves; Fernando N. Nascimento; Giulliano A. Bueno. Moca: A middleware for developing collaborative applications for mobile users. *IEEE Distributed Systems Online, vol. 5, no. 10, 2004*, 2004.

[11] Fabricio Nogueira Buzeto, Carlos Botelho de Paula Filho, Carla Denise Castanho, and Ricardo Pezzuol Jacobi. *DSOA: A Service Oriented Architecture for Ubiquitous Applications*. Springer Berlin / Heidelberg, 2010.

[12] R. Cerqueira. Gaia: A development infrastructure for active spaces. *Workshop on Application Models and Programming Tools for Ubiquitous Computing (held in conjunction with the UBICOMP 2001)*, 2001.

[13] Tim Bray; Jean Paoli; C. M. Sperberg-McQueen; Eve Maler; François Yergeau; John Cowan. Extensible markup language (xml) 1.1 (second edition). Technical report, W3C, 2006. Diponível em `http://www.w3.org/TR/xml11/` (acessado em 07/12/2008).

[14] D. Crockford. The application/json media type for javascript object notation (json). Technical report, Network Working Group, JSON.org, 2006. Diponível em `http://www.ietf.org/rfc/rfc4627.txt?number=4627` (acessado em 04/03/2009).

[15] Cristiano André da Costa, Adenauer Corrêa Yamin, and Cláudio Fernando Resin Geyer. Toward a general software infrastructure for ubiquitous computing. *IEEE Pervasive Computing*, 7(1):64–73, 2008.

[16] Carolina Ribeiro de Enoki; Marcelo Bassani de Freitas. Estudo comparativo entre xml e json para protocolos em ambientes de computação ubíqua. *UnB - Universidade de Brasília, Departamento de Ciências da Computação*, 2008.

[17] Ouahiba Fouial; Katia Abi Fadel; Isabelle Demeure. Adaptive service provision in mobile computing environment. *Ecole Nationale Superieure des Telecommunications*, 2002. Diponível em `http://www.infres.enst.fr/~demeure/PUBLIS/ASWN_2002_Final_Version.pdf` (acessado em 02/11/2008).

[18] Stefan Dukaczewski. Asrd - a step in the right direction. Diponível em `http://www.mstrpln.com/asrd/` (acessado em 03/11/2008).

[19] S. Floyd E. Kohler, M. Handley. Datagram congestion control protocol (dccp). Technical report, Network Working Group, 2006. Diponível em `http://tools.ietf.org/html/rfc4340` (acessado em 21/01/2010).

[20] J.-P. Faure. The ieee p1901 project: broadband over power lines. In *Consumer Electronics, 2006. ICCE '06. 2006 Digest of Technical Papers. International Conference on*, pages 159–160, Jan. 2006.

[21] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. Technical report, Network Working Group, 1999. Diponível em `http://tools.ietf.org/html/rfc2616` (acessado em 21/01/2010).

[22] Apache Software Foundation and Sun Microsystems. Jini. Diponível em `http://www.jini.org` (acessado em 20/01/2010).

[23] D. Garlan. Project aura: Toward distraction-free pervasive computing. *Pervasive Computing, IEEE*, 2002.

[24] Alexandre Rodrigues Gomes. Ubiquitos - uma proposta de arquitetura de middleware para a adaptabilidade de serviços em sistemas de computação ubíqua. Master's thesis, Universidade de Brasília; Departamento de Ciências da Computação, 2007. Diponível em `http://monografias.cic.unb.br/dspace/handle/123456789/110` (acessado em 02/11/2008).

[25] Bluetooth Special Interest Group. Bluetooth special interest group site. Diponível em `http://www.bluetooth.org/` (acessado em 03/11/2008).

[26] R. Lanphier H. Schulzrinne, A. Rao. Real time streaming protocol (rtsp). Technical report, Network Working Group, 1998. Diponível em `http://tools.ietf.org/html/rfc2326` (acessado em 21/01/2010).

[27] C. Matthew MacKenzie; Ken Laskey; Francis McCabe; Peter F. Brown; Rebekah Metz; Booz Allen Hamilton. *Reference Model for Service Oriented Architecture 1.0*. OASIS, 2006. Diponível em `http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf` (acessado em 02/11/2008).

[28] C. Matthew MacKenzie; Ken Laskey; Francis McCabe; Peter F. Brown; Rebekah Metz; Booz Allen Hamilton. *Reference Model for Service Oriented Architecture 1.0*. OASIS, 2006. Diponível em `http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf` (acessado em 09/11/2008).

[29] Michael R. Head, Madhusudhan Govindaraju, Aleksander Slominski, Pu Liu, Nayef Abu-Ghazaleh, Robert van Engelen, Kenneth Chiu, and Michael J. Lewis. A benchmark suite for soap-based communication in grid web services. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 19, Washington, DC, USA, 2005. IEEE Computer Society.

[30] IEEE. Ieee 802.3 ethernet working group. Technical report, IEEE, 2008. Diponível em `http://www.ieee802.org/3/` (acessado em 08/12/2008).

[31] GALE INTERNATIONAL and POSCO E&C. Songdo ibd, 2010. Diponível em `http://www.songdo.com/` (acessado em 16/04/2010).

[32] ISO/IEC. 802.11-1999 - information technology- telecommunications and information exchange between systems- local and metropolitan area networks- specific requirements- part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. Technical report, ISO/IEC, 2003. Diponível em `http://ieeexplore.ieee.org/servlet/opac?punumber=9543` (acessado em 25/02/2009).

[33] Daniele Sacchetti; Angel Talamona; Christophe Cerisara; Rafik Chibout; Slim Ben Atallah; Wolfgang Van Raemdonck; Nikolaos Georgantas; Valérie Issarny. Seamless access to mobile services for the mobile user. *Georgia Institute of Technology*, 2005.

[34] Sumi Helal; William Mann; Hicham El-Zabadani; Jeffrey King; Youssef Kaddoura; Erwin Jansen. The gator tech smart house: A programmable pervasive space. *IEEE Computer magazine*, 2005.

[35] JSON.org. Json - java script object notation. Diponível em `http://www.json.org/` (acessado em 07/12/2008).

[36] Tim Kindberg and Armando Fox. System software for ubiquitous computing. *IEEE Pervasive Computing*, 1(1):70–81, 2002.

[37] Ching Law and Kai-Yeung Siu. An o(log n) randomized resource discovery algorithm. *Auto-ID Center*, 2000.

[38] Lucas Caio André Lins. Fluxo de dados contínuo no middleware uos. *UnB - Universidade de Brasília, Departamento de Ciências da Computação*, 2009.

[39] Sumi Helal; Bryon Winkler; Choonhwa Lee; Youssef Kaddoura; Lisa Ran; Carlos Giraldo; Sree Kuchibhotla; William Mann. Enabling location-aware pervasive computing applications for the edlerly. *First IEEE International Conference on Pervasive Computing and Communications (PerCom'03)*, 2003.

[40] Erwin Aitenbichler; Jussi Kangasharju; Max Mühlhäuser. MundoCore: A Light-weight Infrastructure for Pervasive Computing. *Pervasive and Mobile Computing*, 2007. doi:10.1016/j.pmcj.2007.04.002 (332–361).

[41] Valerie Issarny; Daniele Sacchetti; Rafik Chibout; Sami Dalouche; Mirco Musolesi. Wsami: A middleware infrastructure for ambient intelligence based on web services. Technical report, ARLES Research Page, 2005. Diponível em `http://www-rocq.inria.fr/arles/work/wsami.html` (acessado em 23/11/2008).

[42] Roy W. Schulte; Yefim V. Natis. *Service Oriented Architectures Parts 1 and 2.* Gartner, 1996. Diponível em `http://www.gartner.com/DisplayDocument?id=302868andhttp://www.gartner.com/DisplayDocument?id=302869` (acessado em 25/02/2009).

[43] Eric Newcomer. *Understanding Web Services - XML, WSDL, SOAP and UDDI.* Independent Technology Guides, 2002.

[44] Estevão Lamartine Nogueira Passarinho. Uma proposta de arquitetura microkernel na camada de comunicação de dados do middleware ubiquitos para computação ubíqua. *UnB - Universidade de Brasília, Departamento de Ciências da Computação*, 2008.

[45] Charles Perkins. Slp white paper. Technical report, Sun Microsistems, 1998. Diponível em `http://playground.sun.com/srvloc/slp_white_paper.html` (acessado em 09/11/2008).

[46] Philips. Philips - lumalive textile garments. Diponível em `http://www.lumalive.com/` (acessado em 03/11/2008).

[47] Philips. Skintile the electronic sensing jewelry. Diponível em http://www.design.philips.com/probes/projects/electronic_sensing_jewelry/index.page (acessado em 30/03/2010).

[48] G. E. Krasner; S. T. Pope. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Parc Place Systems Inc*, 1988.

[49] Witric Power. Witric power - wi-fi eletricity. Diponível em http://www.witricpower.com/ (acessado em 03/11/2008).

[50] Lucas Paranhos Quintella. Tratamento de conectividade em ambientes computacionalmente ubíquos. *UnB - Universidade de Brasília, Departamento de Ciências da Computação*, 2009.

[51] Martin Modahl; Ilya Bagrak; Matthew Wolenetz; Phillip Hutto; Umakishore Ramachandran. Mediabroker: An architecture for pervasive computing. *Georgia Institute of Technology*, 2004.

[52] Sameer Adhikari; Arnab Paul; Umakishore Ramachandran. D-stampede: Distributed programming system for ubiquitous computing. *College Of Computing, Georgia Institute of Technology*, 2002.

[53] Beatriz Ribeiro, João Gondim, Ricardo Jacobi, and Carla Castanho. Autenticação mútua entre dispositivos no middleware uos. *SBSEG - Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, 2009.

[54] Manuel Roman and Roy H. Campbell. A model for ubiquitous applications. Technical report, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2001.

[55] Mike Seele; Ron Rosenberg. Illuminate & communicate - boston university partners in nsf challenge to create next generation wireless network using visible light. *Boston University College of Engineering*, 2008. Diponível em http://smartlighting.bu.edu/news/articles/SLC_Press%20Kit%2010308.pdf (acessado em 03/11/2008).

[56] Johannes Schmitt, Matthias Kropff, Andreas Reinhardt, Matthias Hollick, Christian Schäfer, Frank Remetter, and Ralf Steinmetz. An extensible framework for context-aware communication management using heterogeneous sensor networks. Technical Report TR-KOM-2008-08, KOM - TU-Darmstadt, Nov 2008.

[57] Barry Brumitt; Brian Meyers; John Krumm; Amanda Kern; Steven Shafer. Easyliving: Technologies for intelligent environments. Technical report, Microsoft, 2000.

[58] W. Stevens, G. Myers, and L. Constantine. *Structured design.* Yourdon Press, Upper Saddle River, NJ, USA, 1979.

[59] Sun. Developer resources for java technology. Technical report, Sun, 2008. Diponível em http://java.sun.com/ (acessado em 08/12/2008).

[60] ThinkGeek. Wi-fi detector shirt. Diponível em http://www.thinkgeek.com/tshirts/illuminated/991e/ (acessado em 03/11/2008).

[61] ITU International Telecommunication Union. Itu - statistics, 2009. Diponível em http://www.itu.int/ITU-D/ict/statistics/ (acessado em 15/04/2010).

[62] W3C. Soap specification. Technical report, W3C, 2000. Diponível em http://www.w3.org/TR/soap/ (acessado em 07/12/2008).

[63] Mark Weiser. The computer for the 21st century. *Scientific American*, 1991. Diponível em http://nano.xerox.com/hypertext/weiser/SciAmDraft3.html (acessado em 02/11/2008).

[64] Mark Weiser. The world is not a desktop. *ACM Interactions*, 1993. Diponível em http://www.ubiq.com/hypertext/weiser/ACMInteractions2.html (acessado em 02/11/2008).

[65] F. Yergeau. Utf-8, a transformation format of iso 10646. Technical report, Alis Technologies, 2003. Diponível em http://www.ietf.org/rfc/rfc3629.txt (acessado em 30/06/2009).