Francisco Fabrício de Paula Lima

SysSU - Um Sistema de Suporte para Computação Ubíqua

Fortaleza – Ceará – Brasil

2011

Francisco Fabrício de Paula Lima

SysSU - Um Sistema de Suporte para Computação Ubíqua

Dissertação submetida à coordenação do programa de Mestrado e Doutorado em Ciência da Computação, da Universidade Federal do Ceará como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientadora: Rossana Maria de Castro Andrade, PhD.

Universidade Federal do Ceará

Fortaleza – Ceará – Brasil

2011

Resumo

As tecnologias de hardware necessárias para a realização da Computação Ubíqua (*e.g.*, *smartphones*, *tablets*, sensores e eletrodomésticos inteligentes) evoluíram e, atualmente, componentes de software que possibilitam novas formas de interação, capazes de reconhecer a face e a voz dos usuários, rastrear a sua localização e prover formas de incorporar realidade aumentada, necessários em ambientes ubíquos, estão sendo largamente desenvolvidos. Além disso, a Engenharia de Software evoluiu e passou a incorporar novas técnicas de desenvolvimento buscando aumentar a qualidade e diminuir o tempo de produção desses artefatos de software. Contudo, na Computação Ubíqua as dificuldades surgem no projeto das arquiteturas, na modelagem da colaboração entre os componentes de software heterogêneos, na engenharia do sistema, e na comercialização, implantação e validação do sistema como um todo. Assim, o desenvolvimento de sistemas ubíquos ainda apresenta inúmeros desafios devido à grande diversidade e complexidade dos seus requisitos. Particularmente, em ambientes ubíquos, onde a volatilidade e a heterogeneidade de dispositivos, serviços e aplicações são características comuns, formas de interações desacopladas e interoperáveis entre as entidades de software distribuídas no sistema tornam-se essenciais. Essa volatilidade e heterogeneidade influencia a forma como os serviços são disponibilizados no sistema, como são descobertos e invocados e como suas atividades são coordenadas para se atingir o objetivo final das aplicações. Nesse contexto, este trabalho de dissertação propõe uma nova infraestrutura de software, na forma de um sistema de suporte, denominada *SysSU* (*System Support for Ubiquity*). Esse sistema de suporte é apresentado por meio de uma arquitetura de referência e de uma especificação formal. A arquitetura é baseada nos modelos *Linda* e *publish/subscribe* com o propósito de se atingir o desacoplamento desejado. Já a especificação formal determina a semântica das primitivas utilizadas para as interações entre os componentes de software e a sintaxe das mensagens trocadas entre eles com o objetivo de padronizar as implementações. Para validar o trabalho proposto, uma implementação de referência, baseada na arquitetura e na especificação formal, foi construída. Essa implementação foi utilizada no desenvolvimento de um estudo de caso consistindo de duas aplicações distintas, *GREat Tour* e *UbiPrinter*, que apresentam características de volatilidade e heterogeneidade.

Conteúdo

1 Introdução

Esta dissertação tem como objetivo apresentar o *SysSU* (*System Support for Ubiquity*), uma infraestrutura de software, na forma de um sistema de suporte, que fornece soluções para requisitos presentes no desenvolvimento de sistemas ubíquos. Este capítulo apresenta a contextualização e a motivação que levaram ao desenvolvimento dessa solução, bem como os benefícios a serem alcançados com a sua utilização. Assim, na seção 1.1 é introduzida a Computação Ubíqua. Em seguida, na seção 1.2 é apresentada a motivação para o desenvolvimento do *SysSU*. Os objetivos desta dissertação e as contribuições pretendidas são expostos na seção 1.3. Por fim, na seção 1.4 é mostrado como a dissertação está organizada.

1.1 Contextualização

As pesquisas em Computação Ubíqua tiveram início com Mark Weiser em 1988 no Xerox PARC (*Palo Alto Research Center*) quando dirigia o CSL (*Computer Science Laboratory*) (KRUMM, 2009). Nessa época, Weiser imaginava um futuro em que a tecnologia estaria embarcada nos objetos do cotidiano das pessoas auxiliando a execução de suas tarefas diárias. Uma descrição detalhada sobre a pesquisa conduzida por Weiser no Xerox PARC pode ser encontrada no artigo *The Computer for the 21st Century* (WEISER, 1991), o qual traz a expressão: "*As tecnologias mais profundas são aquelas que desaparecem. Elas se misturam com os objetos do dia a dia até se tornarem indistinguíveis no ambiente*".

A Computação Ubíqua é um paradigma da Ciência da Computação que propõe o uso de um conjunto de computadores dos mais variados tamanhos, formatos e funções, que de forma coordenada e autônoma, auxiliam as pessoas na realização das diversas tarefas cotidianas (KRUMM, 2009). Esse auxílio é realizado de tal forma que a infraestrutura computacional responsável fica "escondida" no ambiente. Assim, nesse paradigma, os computadores e outros diversos dispositivos eletrônicos passam a ser sob medida e embutidos

em diversos locais e nos mais variados objetos utilizados no cotidiano, dessa forma, mais naturais às atividades das pessoas. Tais dispositivos se tornam praticamente invisíveis, as pessoas os utilizam quase sem se dar conta deles, da mesma forma que utilizam a energia elétrica. Eles possuem o objetivo principal de auxiliar nas atividades dos usuários sem que eles notem a infraestrutura computacional ao seu redor ou, como proposto por Mahadev Satyanarayanan, *"com a minima distração do usuário"* (SATYANARAYANAN, 2001).

De certa forma, a Computação Ubíqua é uma evolução da ideia da computação pessoal, onde cada pessoa possui seu próprio computador, que já foi uma evolução da ideia anterior dos *mainframes*, em que um único computador era usado por várias pessoas através de terminais. Além disso, muda-se o foco de interesse da tecnologia computacional para as pessoas e suas necessidades (GRIMM et al., 2004). Tom Moran e Paul Dourish notaram que os esforços em Computação Ubíqua expandem o lugar e o modo de interação além do *desktop*, indo para o mundo real em que se vive e em que as interações são efetivamente realizadas (MORAN; DOURISH, 2001). Eles também argumentam que o principal desafio em tais sistemas é tornar a computação útil nas mais variadas situações que possam ser encontradas no cotidiano dos usuários.

Nos últimos anos, a sociedade tem vivenciado um avanço tecnológico acelerado. Em particular, as áreas de microeletrônica e telecomunicação têm contribuído significativamente para o surgimento de novas tecnologias de redes de computadores e para a evolução dos dispositivos móveis (*e.g.*, *smartphones*, *tablets* e *laptops*) em termos de conectividade, poder de processamento e armazenamento. Tudo isso torna possível a concepção de sistemas computacionais onde componentes de hardware e software podem se agrupar de maneira *ad-hoc* para a realização de tarefas, através de uma execução coordenada de atividades, a qual inclui troca de informações, invocação de serviços e compartilhamento de recursos (BARESI; NITTO; GHEZZI, 2006; COSTA; YAMIN; GEYER, 2008; MAIA; ROCHA; ANDRADE, 2009). Esses avanços podem ser vistos como um caminho rumo a concretização da Computação Ubíqua.

É importante destacar ainda que muitas vezes são tratadas sem distinção a Computação Pervasiva (HANSMANN et al., 2003) e a Computação Ubíqua, os quais se referem a segmentos distintos de pesquisa e desenvolvimento tecnológico. A Computação Ubíqua integra os avanços da Computação Móvel (BELLAVISTA; CORRADI, 2006) e da Computação Pervasiva. Enquanto a Computação Móvel busca o incremento das possibilidades de locomoção de serviços entre ambientes, a Computação Pervasiva trata da capacidade de dispositivos computacionais serem embutidos no universo físico para a obtenção de

informações do meio para a construção dinâmica de novos modelos computacionais. Em síntese, a Computação Pervasiva busca maior integração entre a computação e o ambiente físico no qual ela está imersa (NIEUWDORP, 2007).

Assim, o grande desafio da Computação Ubíqua é a constante busca pela fusão da mobilidade da Computação Móvel com a capacidade de interação com o meio agregada pela Computação Pervasiva. Em um contexto prático, na Computação Ubíqua, o computador, independente de sua forma, desloca-se com o usuário e comunica-se com os recursos computacionais do ambiente em que se encontra para configuração dinâmica de novos serviços de tal forma que as necessidades do usuário sejam satisfeitas de maneira natural e transparente (NIEUWDORP, 2007).

1.2 Motivação

Em sua essência, sistemas ubíquos são sistemas distribuídos abertos (BARESI; NITTO; GHEZZI, 2006), voláteis (KINDBERG; FOX, 2002; COULOURIS; KINDBERG; DOLLIMORE, 2005), heterogêneos (SATYANARAYANAN, 2001) e centrados no usuário (WEISER, 1991) que apresentam como principais características a interoperabilidade espontânea e a integração com o mundo físico (KINDBERG; FOX, 2002). Para tanto, eles precisam atender, simultaneamente, a requisitos não triviais, tais como: coordenação, interoperabilidade, mobilidade, sensibilidade ao contexto e autonomicidade (COSTA; YAMIN; GEYER, 2008; MAIA; ROCHA; ANDRADE, 2009; JAROUCHEH; LIU; SMITH, 2009).

Assim, o desenvolvimento de sistemas ubíquos não é uma tarefa simples quando comparados a sistemas distribuídos tradicionais (SATYANARAYANAN, 2001; COULOURIS; KINDBERG; DOLLIMORE, 2005). Surgem novos paradigmas e desafios para o desenvolvimento de software, agora necessitando ser projetado para lidar intrinsecamente com certos requisitos e limitações, como volatilidade, heterogeneidade e reduzida capacidade dos dispositivos (SATYANARAYANAN, 2001; KINDBERG; FOX, 2002). Por esse motivo, o estudo das técnicas de desenvolvimento de software ubíquo, observando as boas práticas da Engenharia de Software, torna-se uma necessidade e um fator diferencial para os desenvolvedores. Nesse contexto, para suprir as características e requisitos de sistemas ubíquos, obedecendo as suas restrições, são necessárias infraestruturas de software apropriadas, como *frameworks*, *middlewares*, sistemas de suporte, entre outros (NIEMELä; LATVAKOSKI, 2004). Essas infraestruturas são importantes, pois auxiliam o trabalho

dos projetistas e desenvolvedores de sistemas ubíquos fornecendo soluções prontas e reutilisáveis para os principais problemas encontrados.

Um modelo de coordenação de atividades pode ser entendido como um modelo de interação que descreve a forma como os componentes de software presentes no sistema devem proceder para que o requisito esperado seja atendido (COULOURIS; KINDBERG; DOLLIMORE, 2005). Devido à mobilidade ou à exaustão de recursos computacionais (e.g., esgotamento de bateria e memória dos dispositivos móveis), os componentes de software (aplicações ou serviços) que executam nesses dispositivos podem entrar e sair do sistema de maneira imprevisível. Essa volatilidade sugere que poucas suposições sobre o estado ou a configuração do sistema podem ser feitas (MAIA; ROCHA; ANDRADE, 2009) e, por esse motivo, o modelo de coordenação adotado deve evitar ao máximo o acoplamento entre os componentes do sistema (EUGSTER et al., 2003). Além disso, devido à natureza aberta dos sistemas ubíquos (BARESI; NITTO; GHEZZI, 2006), não é possível conhecer, em tempo de projeto, todas as características de hardware e software dos componentes de software que comporão o sistema. A heterogeneidade impõe barreiras à interoperabilidade e, consequentemente, a coordenação de atividades entre esses componentes fica restrita àqueles que possuem tecnologia compatível.

Além disso, a dinamicidade e a heterogeneidade influencia na forma como os serviços disponíves no sistema, que precisam ser coordenados, são providos e descobertos, pois os sistemas ubíquos podem não apresentar uma arquitetura estática. Logo, não é possível identificar a localização precisa de um dispositivo que provê um serviço desejado e nem conhecer antecipadamente como o serviço é invocado ou se ele está disponível (ZHU; MUTKA; NI, 2005; LIU; PENG; CHEN, 2006; HANDOREAN; ROMAN, 2007). Dessa forma, uma infraestrutura desacoplada e interoperável também se faz necessária para solucionar os problemas advindos dessas características.

Assim, devido a sua volatilidade e heterogeneidade, em um sistema ubíquo as interações devem ser realizada preferencialmente de forma desacoplada e interoperável. Além disso, essa forma de comunicação se faz necessária também em outros requisitos comuns aos sistemas ubíquos, como sensibilidade ao contexto e autonomicidade, visto que para a realização deles existe a necessidade de alguma forma de comunicação entre os componentes de software envolvidos.

Infraestruturas de comunicação apropriadas para sistemas ubíquos devem prover mecanismos que promovam o desacoplamento entre esses agentes[1] e possuir uma implemen-

[1]Nesta dissertação esses componentes de software são chamados de agentes, e representam qualquer

tação que garanta independência de plataforma de desenvolvimento e execução, para que a interoperabilidade desejada seja alcançada. Apesar de existirem tecnologias que garantem interoperabilidade, como o *CORBA* (OMG, 2011) e os *web services* (ALONSO et al., 2004), e modelos de coordenação desacoplados, como os baseados no modelo *Linda* (CARRIERO; GELERNTER, 1989), no modelo *publish/subscribe* (EUGSTER et al., 2003) ou em ambos, como o *LIME* (MURPHY; PICCO; ROMAN, 2006), as únicas soluções encontradas que endereçam os dois problemas conjuntamente são o *GigaSpaces* (LTD, 2011) e o *X-MARS* (CABRI; LEONARDI; ZAMBONELLI, 2000). Porém, essas duas soluções apresentam desvantagens quando usadas em sistemas ubíquos. O primeiro é destinado a sistemas empresariais de grande porte, não se adequando a dispositivos de pouca capacidade de processamento. Já na segunda, a interoperabilidade é voltada para a representação de dados e não para a interação entre os agentes. Além disso, as tecnologias de descoberta de serviços atuais, como o *Sun Microsystems Jini Network Technology*, o *Microsoft Universal Plug and Play* (*UPnP*) e o *Apple Rendezvous*, não são adequadas, por completo, para ambientes ubíquos (ZHU; MUTKA; NI, 2005; LIU; PENG; CHEN, 2006; EDWARDS, 2006).

1.3 Objetivo e Contribuição

Devido à lacuna existente de soluções que enderecem conjuntamente os requisitos de coordenação e descoberta de serviços em sistemas ubíquos, este trabalho tem o objetivo de fornecer uma infraestrutura de software, denominada *SysSU* (*System Support for Ubiquity*). Essa infraestrutura é fornecida como um sistema de suporte (GRIMM et al., 2004) e possui como principal objetivo prover mecanismos apropriados para a troca de mensagens, coordenação de atividades e descoberta/invocação de serviços para sistemas ubíquos considerando o desacoplamento e a interoperabilidade necessária.

Nesta dissertação, esse sistema de suporte é apresentado por meio de uma arquitetura de referência e de uma especificação formal. A arquitetura é baseada nos modelos *Linda* e *publish/subcribe* com o propósito de se atingir o desacoplamento desejado. Já a especificação formal determina a semântica das primitivas utilizadas para as interações entre os componentes de software e a sintaxe das mensagens trocadas entre eles com o objetivo de padronizar as implementações. Com essa padronização, é possível tornar a implementação independente em relação à plataforma de desenvolvimento e linguagem de programação. Assim, é alcançada a interoperabilidade e a independência de plataforma de

entidade de software presente no sistema capaz de realizar algum processamento e se comunicar.

execução utilizadas na construção/execução das aplicações e dos serviços que farão parte do sistema. Além disso, a arquitetura adotada e as primitivas de comunicação disponibilizadas podem ser usadas e/ou estendidas para o suporte aos requisitos de sensibilidade a contexto e autonomicidade.

Portanto, a principal vantagem da utilização do *SysSU* é o suporte a vários requisitos importantes usando uma mesma solução e arquitetura, facilitando, assim, o trabalho no desenvolvimento e implantação de sistemas ubíquos e de seus serviços e aplicações. Apesar do foco dessa solução ser sistemas ubíquos, pois a maioria deles apresentam os requisitos discutidos na seção anterior, ela pode ser usada, também, no desenvolvimento de sistemas que apresentam os mesmo problemas relacionados a esses requisitos, como por exemplo, sistemas móveis com grande heterogeneidade.

1.4 Organização da Dissertação

Esta trabalho é dividido nos seguinte capítulos, além do capítulo 1:

- No capítulo 2 é apresentado um levantamento de um conjunto de características que determinam um sistema ubíquo e de um conjunto de requisitos de software em nível de sistema para implementá-las.
- No capítulo 3 é descrito um conjunto de trabalhos relacionados, dos quais nenhum se mostrou completo em relação aos requisitos levantados, principalmente em relação aos que são foco deste trabalho: coordenação, descoberta de serviços e interoperabilidade.
- No capítulo 4 é apresentado o *SysSU* por meio de uma arquitetura de referência e de uma especificação formal das primitivas de comunicação.
- No capítulo 5, uma implementação real dos conceitos, da arquiteura e dos componentes presentes no capítulo 4 é mostrada. Nesse capítulo é discutido como os conceitos são transformados em código executável e funcional para a sua utilização no desenvolvimento de sistemas ubíquos. Além disso, é apresentada a utilização da implementação do *SysSU* no contexto de um estudo de caso.
- No capítulo 6 é apresentada a conclusão sobre esta dissertação, bem como os trabalhos futuros a serem desenvolvidos.

2 Características e Requisitos de Sistemas Ubíquos

De certa forma, um sistema é considerado ubíquo no momento em que os serviços, informações ou facilidades computacionais são disponibilizados aos usuários de forma que os dispositivos responsáveis por isso não sejam visíveis. Eles são sistemas distribuídos abertos (BARESI; NITTO; GHEZZI, 2006), voláteis (COULOURIS; KINDBERG; DOLLIMORE, 2005), heterogêneos (SATYANARAYANAN, 2001) e centrados no usuário (WEISER, 1991). Além disso, ao contrário de sistemas distribuídos convencionais, a transparência de distribuição deve ser evitada (GRIMM et al., 2004; TANENBAUM; STEEN, 2007). Dessa forma, um agente no sistema pode estar ciente do fato que o ambiente computacional está em constante mudança (GRIMM et al., 2004).

Neste capítulo, na seção 2.1 é mostrada uma caracterização que define se um sistema pode ser enquadrado como ubíquo. Já na seção 2.2, um conjunto de requisitos de software para implementar essas caraterísticas é apresentado. Esse levantamento tem como principal objetivo guiar o desenvolvimento de recursos reutilizáveis de software, como sistemas de *middleware* e *frameworks*, que auxiliem o projeto e desenvolvimento de sistemas ubíquos. Por fim, na seção 2.3 são sumarizados os principais pontos relacionados a esse levantamento.

2.1 Características Fundamentais de Sistemas Ubíquos

Essa caracterização teve como base os trabalhos de Rodrigo Spínola *et. al.* (SPíNOLA; MASSOLLAR; TRAVASSOS, 2007), Tim Kindberg e Armando Fox (KINDBERG; FOX, 2002) e Debashis Saha e Amitava Mukherjee (SAHA; MUKHERJEE, 2003). No primeiro trabalho foi realizado uma revisão sistemática de projetos de sistemas ubíquos existentes com o objetivo de se chegar a um conjunto de características comuns entres eles. Nos

outros dois são levantados requisitos gerais e desafios para o desenvolvimento de sistemas ubíquos.

As características levantadas são:

C1 – Captura de experiências e intenções: habilidade de perceber as necessidades dos usuários e suas atividades com o objetivo de não ser necessário os usuários as determinarem explicitamente;

C2 – Comportamento adaptável: habilidade de se adaptar dinamicamente ao contexto físico e lógico do ambiente e dos dispositivos para ficar em um estado operacional otimizado e personalizado para a realização de determinada tarefa para o usuário;

C3 – Descentralização: as responsabilidades são distribuídas entre os vários dispositivos presentes no ambiente, onde cada dispositivo é responsável pela execução de um conjunto de tarefas e funções específicas;

C4 – Descoberta de serviços automática: os serviços desejados são descobertos de forma proativa, sem a intervenção do usuário para a realização de escolhas ou configurações manuais;

C5 – Heterogeneidade de dispositivos e serviços: capacidade de suportar, adaptando-se, se necessário, a uma grande variedade de dispositivos e serviços;

C6 – Interoperabilidade espontânea: os diversos dispositivos e serviços interagem de forma automática e sem a intervenção do usuário;

C7 – Mínima intervenção do usuário: os dispositivos eletrônicos que formam o ambiente ubíquo devem ficar o máximo possível longe da percepção dos usuários, com o principal intuito de não ser necessária qualquer operação direta sobre eles;

C8 – Onipresença dos serviços: os usuários podem se mover nos limites físicos do ambiente ou trocar de dispositivos com a sensação de que estão "carregando" os serviços em uso com eles; e

C9 – Tolerância a falhas: habilidade de se recompor depois da ocorrência de falhas sem envolver os usuários nesse processo.

2.2 Requisitos Básicos para o Desenvolvimento de Sistemas Ubíquos

Tomando como base um conjunto de trabalhos (SATYANARAYANAN, 2001; KINDBERG; FOX, 2002; NIEMELä; LATVAKOSKI, 2004; STANFORD, 2004; SCHOLTZ; CONSOLVO, 2004; GRIMM et al., 2004; COULOURIS; KINDBERG; DOLLIMORE, 2005; WANT; PERING, 2005; SPíNOLA; MASSOLLAR; TRAVASSOS, 2007; COSTA; YAMIN; GEYER, 2008; MAIA; ROCHA; ANDRADE, 2009), é possível definir um conjunto mínimo de requisitos de software necessários para se atingir as características levantadas. Esses trabalhos apresentam conjuntos divergentes de requisitos relacionados a sistemas ubíquos e pervasivos. Alguns são voltados para ambientes específicos e outros são mais genéricos. Entretanto, é possível definir pontos em comum entre a maioria deles. Esses requisitos estão relacionados ao sistema como um todo e influenciam as interações entre os agentes no ambiente ubíquo. Além deles, as aplicações e serviços possuem seus próprios requisitos, variando de acordo com o propósito estabelecido para o sistema ubíquo.

Essa diferenciação de requisitos é similar a apresentada por Eila Niemelä e Juhani Latvakoski (NIEMELä; LATVAKOSKI, 2004). Nesse trabalho, os autores dividem os requisitos em três categorias: (1) *Sistema*, semelhante a um sistema de suporte; (2) *Software*, referente ao serviços e componentes de um sistema ubíquo realizado por tecnologias de software, que neste trabalho são representados pelas aplicações e serviços; e (3) *Negócio e Organização*, referente ao conjunto de atores que proveem os serviços e componentes usados no desenvolvimento e implantação de um sistema ubíquo e os métodos de desenvolvimento utilizados na integração desses componentes e serviços.

Um ponto importante no trabalho de Eila Niemelä e Juhani Latvakoski, compartilhado por esta dissertação, é eles considerarem que um sistema ubíquo é formado por três camadas de software: aplicações e serviços, *middleware* e infraestrutura de sistema. Adicionalmente, eles ponderam que padronizações e arquiteturas de referência conjuntamente com tecnologias de software genéricas e especializadas são essenciais para o desenvolvimento e implantação de sistemas ubíquos.

Os requisitos são: *Coordenação*, *Descoberta/Descrição de Serviços*, *Interoperabilidade*, *Sensibilidade ao Contexto*, *Invisibilidade* e *Autonomicidade*. A Tabela 2.1 apresenta as relações entre as características e esses requisitos. Na explicação de cada requisito, realizada nas próximas seções, essas relações são detalhadas.

Tabela 2.1: Relação entre características e requisitos de software em sistemas ubíquos.

Requisitos	Características								
	C1	C2	C3	C4	C5	C6	C7	C8	C9
Coordenação			X			X		X	
D./D. de Serviços			X	X		X		X	
Interoperabilidade			X		X	X			
Sensibilidade ao Contexto	X	X		X	X		X		X
Invisibilidade	X	X		X			X		
Autonomicidade		X			X			X	X

2.2.1 Coordenação

Devido a característica de descentralização (C3), para realizar efetivamente suas funcionalidades e atender às expectativas dos usuários, um sistema ubíquo necessita da participação de vários agentes de software, como serviços e aplicações. Cada agente realiza uma parcela de trabalho para que, ao final, através de uma execução de atividades adequada e controlada, o resultado esperado seja alcançado. Dessa forma, mecanismos adequados de comunicação e coordenação de atividades entre agentes são imprescindíveis para garantir que o sistema atinja o seu objetivo final.

Sistemas ubíquos são essencialmente dinâmicos e, assim, os agentes presentes nele precisam estar aptos a interagir e coordenar suas atividades com outros agentes desconhecidos (BARESI; NITTO; GHEZZI, 2006). Além disso, os sistemas ubíquos se tornam mais poderosos e úteis se a mobilidade física e lógica forem consideradas (MURPHY; PICCO; ROMAN, 2006). A mobilidade física está relacionada com a capacidade dos nós (dispositivos móveis) poderem se mover junto com o usuário pelo ambiente enquanto mantém conexões com outros nós (dispositivos móveis ou não). Por outro lado, a mobilidade lógica está relacionada com um estilo arquitetônico dinâmico que remove as ligações entre os componentes de software (serviços) e os nós (dispositivos móveis ou não). Essa flexibilidade permite que componentes migrem de um nó para outro ou que sejam substituídos dinamicamente, melhorando a flexibilidade e o desempenho do sistema como um todo.

Como a mobilidade faz parte do cotidiano das pessoas, a ausência de suporte à ela pode tornar o usuário ciente da computação ao seu redor (SATYANARAYANAN, 2001). Isso se deve ao fato de que, decorrente da mobilidade, desconexões podem ocorrer e serviços podem ficar indisponíveis e, dessa forma, as atividades do usuário podem ser interrompidas. Essa interrupção desvia a atenção do usuário da sua atividade para problemas relacionados ao mundo computacional, o que vai contra as aspirações da Computação Ubíqua (SATYANARAYANAN, 2001; GARLAN et al., 2002). Por isso, um suporte adequado à mobilidade (física e lógica) e, consequentemente, à onipresença de serviços nos lugares

frequentados pelos usuários (i.e., ambientes ubíquos) torna-se um requisito indispensável (SATYANARAYANAN, 2001; GARLAN et al., 2002; SPíNOLA; MASSOLLAR; TRAVASSOS, 2007) e exige um modelo desacoplado de interação (EUGSTER et al., 2003).

Assim, o requisito de coordenação deve ser especializado para se preocupar com a dinamicidade do ambiente, pois os agentes participantes podem entrar, sair e se mover no ambiente de forma imprevisível (SCHELFTHOUT; HOLVOET, 2005) ou serem acometidos por falhas (CHETAN; RANGANATHAN; CAMPBELL, 2005). Outra preocupação é com a incerteza da comunicação sem fio, visto que esta é limitada ao raio de alcance de cada nó (CHETAN; RANGANATHAN; CAMPBELL, 2005). Essa dinamicidade afeta o contexto computacional percebido por um agente, não somente limitado às partes comunicantes, mas também aos dados disponíveis para a realização da sua atividade, o conjunto de serviços acessíveis e os demais recursos necessários. Desse modo, tal coordenação deve evitar ao máximo o acoplamento direto entre os agentes que segundo Eugster *et al.* (EUGSTER et al., 2003), podem ser de três formas:

1. **Espacial ou Referencial**: onde os agentes que interagem não precisam saber da existência um do outro nem possuir referência direta para o outro;

2. **Temporal**: no qual os agentes que interagem não precisam estar presentes e ativos ao mesmo tempo; e

3. **Assíncrono**: no qual os agentes podem receber mensagens contendo notificações de eventos no sistema enquanto realizam um processamento qualquer.

Esse desacoplamento entre comportamento e comunicação permite separar a função desempenhada pelos agentes do mutável contexto computacional em que eles estão inseridos (MAMEI; ZAMBONELLI, 2009). Por exemplo, se um serviço ou recurso mudar de dispositivo, as aplicações que o utilizam não serão afetadas devido ao desacoplamento espacial. Da mesma forma, devido ao desacoplamento temporal, um agente que precisa interagir com outro não necessita esperar até que aquele esteja conectado ao sistema. Já o desacoplamento assíncrono possibilita que o agente continue a execução da tarefa para o qual foi designado enquanto espera pela ocorrência de algum evento de interesse. Quando o evento ocorrer, o agente é notificado sem a necessidade de saber qual agente gerou o evento.

Assim, um modelo de coordenação que contempla as três formas de desacoplamento poderia ser usado para resolver os problemas advindos da mobilidade física e lógica. É

importante salientar que a responsabilidade do gerenciamento da mobilidade física é dada à infraestrutura de rede e ao sistema operacional, que devem manter o dispositivo conectado, visível e com a mesma identificação. Adicionalmente, a mobilidade lógica pode ser desempenhada por outras soluções, dependendo dos requisitos da aplicação. Maiores detalhes sobre soluções para mobilidade em sistemas ubíquos podem ser obtidos no trabalho de Márcio Maia *et. al.* (MAIA; ROCHA; ANDRADE, 2009).

Modelo de Coordenação *Linda*

Um modelo de coordenação que dá suporte adequado ao desacoplamento espacial e temporal é o baseado em espaço de tuplas, conhecido também como comunicação geradora (CABRI et al., 2006). Esse modelo foi inicialmente proposto como base para a implementação do modelo *Linda* (CARRIERO; GELERNTER, 1989), um modelo de coordenação e comunicação de processos paralelos. Nesse modelo, os processos se comunicam através de uma linguagem de coordenação formada por um conjunto limitado de primitivas que acessam espaços de tuplas. As atividades de coordenação entre os agentes são realizadas através da troca indireta de tuplas utilizando, para isso, esse conjunto de primitivas. A Figura 2.1 ilustra esse tipo de coordenação com os seus principais conceitos:

Tupla: é um conjunto de campos tipados, que podem variar em quantidade de uma tupla para outra, que representam alguma informação relevante para o sistema e servem para trocar informações entre os processos. Um exemplo de tupla poderia ser `<"Fabrício Lima"; 28.9; 32.7>`, formada por três campos: um do tipo texto e os outros dois do tipo real.

Espaço de tuplas: é um conjunto de *tuplas* heterogêneas, alocado na forma de repositório, acessado concorrentemente por vários processos.

Produtores: são agentes responsáveis por inserir tuplas no espaço de tuplas. As tuplas só podem ser adicionadas a um espaço de tuplas através da primitiva `out(t)`.

Consumidores: são agentes responsáveis por verificar se determinadas tuplas existem no espaço de tuplas e retirá-las se necessário. Para isso, eles usam as primitivas: `rd(p)`, que copia uma tupla presente do espaço de tuplas e `in(p)`, que retira uma tupla. Essas operações usam mecanismo associativos baseados em um padrão de tuplas descrito a seguir.

***Middleware*:** representa uma camada que separa os agentes do mecanismo de tuplas

e provê uma visão uniforme do mecanismo em face da heterogeneidade do sistema distribuído.

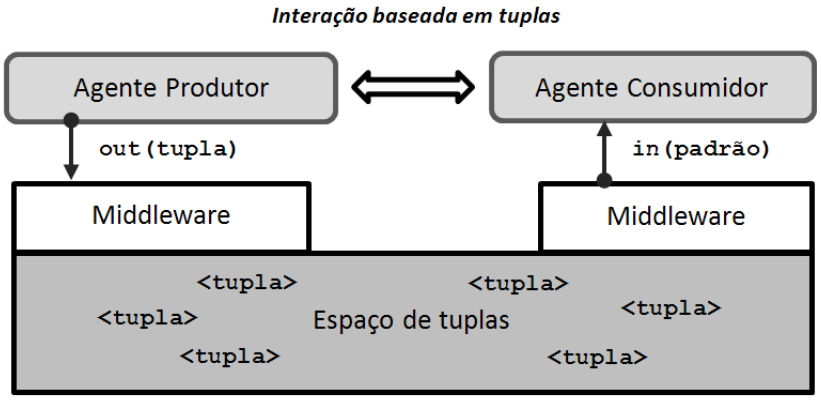

Figura 2.1: Esquematização da coordenação baseada em tuplas.

Nesse modelo três pontos precisam ser destacados:

1. A distinção entre produtor e consumidor só é realizada no momento de uma interação, pois o que distingui um do outro para um espaço de tuplas é a primitiva que está sendo utilizada.

2. Não existe uma ordem definida para que as primitivas sejam utilizadas, por exemplo, na Figura 2.1 o consumidor pode usar a primitiva `in(p)` antes ou depois do consumidor usar a primitiva `out(t)`.

3. A forma como o espaço de tuplas é implementado pode ser de forma centralizada, como o modelo original, distribuído, mas com acesso centralizado (*e.g.*, *GigaSpaces* (LTD, 2011)), ou totalmente distribuído (*e.g.*, *LIME* (MURPHY; PICCO; ROMAN, 2006)). O *middleware* utilizado se responsabiliza por abstrair essa implementação.

As tuplas não possuem identificação e nem seguem esquemas pré-definidos (como em banco de dados relacionais). A busca em um espaço de tuplas é realizada através de um mecanismo associativo. Nesse mecanismo, as tuplas de um espaço são comparadas a um padrão de tupla que possui o mesmo formato que uma tupla normal, mas com algumas especializações. No modelo original, os campos em um padrão podem ser *actual*, os quais representam valores a serem comparados, ou *formal*, os quais representam tipos a serem verificados. Na tupla `<"Fabrício Lima";28.9;32.7>` todos os campos são *actual*, já em `<"Fabrício Lima";?double;?double>` os dois últimos campos são do tipo *formal*. Assim, uma execução `in(<"Fabrício Lima";?double;?double>)`, por exemplo, retornaria

a tupla `<"Fabrício Lima";28.9;32.7>` ou outras que correspondessem ao padrão, *i. e.*, tivesse três campos, sendo o primeiro com o valor `"Fabrício Lima"` e os dois seguintes do tipo `double` com qualquer valor.

Existem várias implementações desse modelo, variando em arquitetura e funcionalidades. Alguns deles já fornecem, também, o desacoplamento assíncrono, através do modelo de eventos. A Tabela 2.2 apresenta algumas implementações. Esta dissertação dá maior destaque ao *Egospace*, *TOTA* e *LIME*, pois foram desenvolvidos focando em sistemas pervasivos e ubíquos. Os mesmos são melhor discutidos no capítulo 3.

Tabela 2.2: Exemplos de implementações do modelo de tuplas.

Soluções	
Objective Linda (KIELMANN, 1995)	*SwarmLinda* (CHARLES *et al.*, 2004)
Limbo (DAVIES et al., 1997)	*LIME* (MURPHY; PICCO; ROMAN, 2006)
TuCSoN (OMICINI; ZAMBONNELI, 1999)	*Egospace* (JULIEN; ROMAN, 2006)
JavaSpaces (FREEMAN; ARNOLD; HUPFLER, 1999)	*TOTA* (MAMEI; ZAMBONELLI, 2009)
X-MARS (CABRI; LEONARDI; ZAMBONELLI, 2000)	*TCP Linda* (ASSOCIATES et. al., 2011)
KLAIM (NICOLA; LEONARDI; ZAMBONELLI, 2001)	*GigaSpaces* (LTD, 2011)
TSpaces (LEHMAN et al., 2001)	

Modelo de Coordenação *publish/subscribe*

Outro modelo de interação desacoplado é o *publish/subscribe* ou de eventos (EUGSTER et al., 2003). Nesse modelo, os agentes se comunicam gerando e recebendo notificação de eventos que, normalmente, são iniciados por uma mudança no estado de um agente. Para isso, um mecanismo de notificação de eventos ou um *middleware* de publicação/subscrição medeia os agentes no sistema. Ele transmite as notificações dos agentes produtores (ou publicadores) para os consumidores (ou subscritores) que registraram seu interesse previamente em tal tipo de evento. A Figura 2.2 ilustra esse tipo de coordenação com os seus principais conceitos:

Evento: é qualquer interesse que pode ser observado por um agente (*e.g*, um evento físico como a entrada de uma pessoa em um ambiente, um evento de *timer* ou uma mudança arbitrária do estado computacional de um agente).

Notificação: representa um conjunto de dados que descreve um evento. A notificação é criada por um observador do evento e pode simplesmente indicar a ocorrência do evento ou pode conter informações adicionais descrevendo as circunstâncias do evento.

Mensagens: são pacotes de dados que contêm as notificações que são transmitidas pelo mecanismo de comunicação utilizado.

Produtores: são agentes que publicam notificações. A decisão de quando publicar e quais dados adicionais são fornecidos depende da implementação de cada um. A publicação não é endereçada a nenhum agente ou conjunto de agentes específicos, ao invés disso, as notificações são enviadas para um mecanismo de eventos presente no sistema que se preocupa com a disseminação da notificação.

Anúncios: são informações enviadas ao mecanismo de eventos que determinam as notificações que o produtor é capaz de publicar.

Consumidores: são agentes que reagem às notificações enviadas pelo mecanismo de eventos. Para isso, eles registram quais tipos de notificações eles estão interessados.

Subscrição: descreve um conjunto de notificações que o consumidor está interessado. Essas descrições são registradas no mecanismo de eventos que os avalia e, posteriormente, entrega as notificações que se relacionarem com qualquer das subscrições. Subscrições são, essencialmente, filtros formados por expressões lógicas que avaliam uma notificação e possui expressividade dependente do modelo de dados e de filtros utilizado. O modelo de dados determina qual é o formato das notificações e o modelo de filtros determina como os consumidores realizam suas subscrições.

Mecanismo de eventos: é o serviço de mediação que desacopla os produtores dos consumidores. Ele é responsável por receber as notificações dos produtores e as encaminhar para os consumidores de acordo com as subscrições registradas. Essas operações são realizadas através de um conjunto de primitivas que compõe a interface do mecanismo. As primitivas são: `adv`, para a realização de anúncios, `pub`, para realizar notificações, `sub` e `unsub`, relacionados a subscrição dos consumidores e `notify`, que entrega as notificações aos consumidores interessados. A Figura 2.2 ilustra um cenário de uso dessas primitivas.

***Middleware*:** representa uma camada que separa os agentes do mecanismo de eventos e provê uma visão uniforme do mecanismo em face da heterogeneidade do sistema distribuído.

A principal vantagem na utilização do modelo *publish/subscribe* é que nem as subscrições nem as notificações são dirigidas diretamente a um agente específico. O mecanismo de eventos desacopla os agentes de forma que os produtores não conhecem quem são os

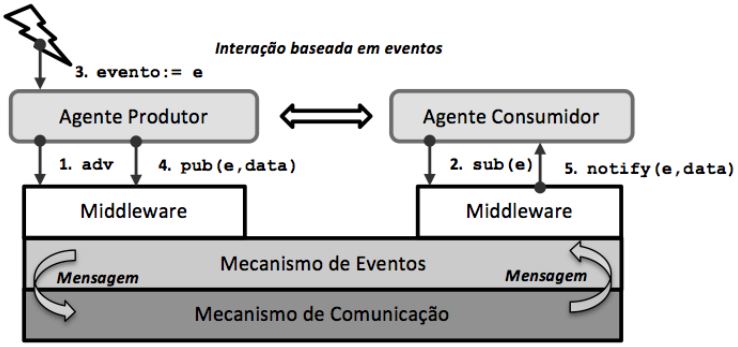

Figura 2.2: Esquematização da coordenação baseada em eventos

consumidores das suas notificações e os consumidores se preocupam somente com os eventos publicados e suas informações e não com quem os publica. Assim, esse modelo facilita a integração de agentes heterogêneos e autônomos em sistemas complexos e dinâmicos que precisam ser evoluídos e escalonados (MüHL; FIEGE; PIETZUCH, 2006).

Contudo, sua principal característica é a possibilidade do comportamento assíncrono entre os agentes que podem realizar outras atividades enquanto não são notificados sobre a ocorrência de eventos de interesse. Porém, esse modelo, a princípio, não oferece o desacoplamento temporal, pois os agentes produtores e consumidores precisam estar ativos no sistema ao mesmo tempo (TANENBAUM; STEEN, 2007). Contudo, algumas implementações como o *JMS* (ORACLE, 2011) permitem notificação tardias e, assim, garantem um desacoplamento temporal.

Existem várias implementações desse modelo, variando em dois principais conceitos: a forma de expressar as subscrições, e se são centralizados ou distribuídos. A tabela 2.3 apresenta algumas implementações importantes.

Tabela 2.3: Exemplos de implementações do modelo *publish/subscribe*.

Soluções
Gryphon (CARZANIGA, 1998)
Siena (AGUILERA *et al.*, 1999)
Jedi (CUGOLA; JACOBSEN, 2002)
Rebeca (PARZYJEGLA *et al.*, 2010)
JMS (ORACLE, 2011)
CORBA (OMG, 2011)

.

O ponto de diferenciação principal do modelo *publish/subscribe* em relação ao modelo *Linda* é a forma como o agente consumidor das mensagens interage. No modelo *Linda*, o agente consome as mensagens proativamente. Já no modelo de eventos, ele assume uma

forma reativa, esperando pelo agente produtor de mensagens publicar os eventos de seu interesse.

Coordenação e Sensibilidade ao Contexto

Outros pontos devem ser considerados quando se trata de mecanismos de coordenação em sistemas ubíquos. Um deles é que a coordenação, às vezes, precisa ser sensível ao contexto (COUTAZ et al., 2005). Um agente pode executar sua função e combinar esforços com outros agentes somente se ele está, de alguma forma, "ciente do que está acontecendo ao seu redor", ou seja, ciente do seu contexto (CABRI et al., 2006), seja ele computacional ou físico. Giacomo Cabri *et. al.* descrevem a coordenação em tais ambientes como a união de interação e sensibilidade ao contexto (CABRI et al., 2006).

Para essa sensibilidade ao contexto ser possível, é necessário a presença de uma infraestrutura de contexto que permita aos agentes consultar informações contextuais de interesse e serem notificados da ocorrência de possíveis alterações nelas. O modelo *Linda* pode ser eficientemente utilizado para compartilhar as informações contextuais necessárias, pois essas informações podem ser representadas através de tuplas, como realizado pelo *LIME* e pelo *Egospace*. Contudo, a pesquisa de tuplas precisa ser melhorada. Como um simples exemplo considere um cenário em que um agente deseja saber o menor valor de uma informação contextual (*e.g.* temperatura, umidade, distância, entre outras). No modelo *Linda*, o agente teria que ler todas as tuplas correspondentes e verificar qual possui a informação com valor menor. Coletar todas essas tuplas e realizar a comparação necessária torna-se inviável se existirem muitas tuplas e o dispositivo possuir pouca capacidade, além de ser necessário elevada transmissão de dados na rede. Esse mesmo conceito pode ser aplicado a subscrições de eventos, por exemplo, onde o agente deseja ser notificado toda vez que um menor valor de uma informação for gerado.

Em alguns casos, uma tupla ou representação de evento não significa uma informação útil para um agente, mas uma derivação feita por meio de uma operação específica, sim. Por exemplo, um agente precisa saber quais dispositivos estão a setenta metros de distância de sua localização, mas as tuplas disponíveis somente informam a posição em termos de coordenadas geográficas `(x,y)`. Coletar todas essas tuplas e realizar o cálculo necessário torna-se inviável pelos mesmos motivos citados anteriormente. Esse exemplo representa uma agregação de campos, mas, às vezes, é necessário uma agregação de tuplas ou eventos. Por exemplo, um agente gostaria de ser avisado quando determinadas cinco pessoas estiverem na sala de reunião. Novamente, esse processo pode ser muito

custoso para o agente realizar podendo ainda ser mais complexo se forem usadas medidas de tempo como notificar quando essas cinco pessoas estiverem a mais de dez minutos na sala de reunião.

Das implementações citadas na tabela 2.2, o *Egospace* e o *TOTA* permitem elevada expressividade. O *LIME/LighTS*, além de ser um modelo misto baseado no *Linda* e em eventos, considera todos esses pontos, caracterizando-se como uma das melhores soluções. Porém, essa expressividade é determinada em tempo de projeto e implementação, diminuindo sua dinamicidade de forma que se forem necessárias novas formas de pesquisa e agregação, uma nova versão do sistema precisa ser implantada. Além disso, nenhuma das três soluções são interoperáveis, possuindo, assim, restrições em relação a heterogeneidade dos sistemas ubíquos. Das implementações do modelo *publish/subscribe* apresentadas na tabela 2.3, o *Rebeca* e o *Jedi* oferecem a expressividade apropriada. Além desses, o modelo proposto por Eugster *et. al.* (EUGSTER; GARBINATO; HOLZER, 2005) considera a sensibilidade ao contexto como uma característica importante no seu mecanismo de eventos.

2.2.2 Descrição e Descoberta de Serviços

A orientação a serviço (ERL, 2005) permite que um problema maior seja decomposto em partes atômicas menores na forma de serviços disponíveis em um sistema distribuído, facilitando o desenvolvimento, implantação, gerência, manutenção e evolução de sistemas de software. Um sistema ubíquo é um sistema distribuído normalmente construído usando uma arquitetura baseada em serviços (THANH; JORSTAD, 2005; ZHU; MUTKA; NI, 2005), onde cada serviço é responsável por um subconjunto das tarefas necessárias para atingir os requisitos funcionais do sistema e de suas aplicações.

Um serviço se torna disponível em um sistema distribuído no momento que ele é registrado em algum mecanismo de descoberta de serviços ou através de anúncios periódicos sobre sua presença no sistema. Para isso, é utilizada uma descrição de serviço, que são informações descritivas, como as funcionalidades oferecidas e atributos diversos, bem como informações que os clientes precisam para utilizá-lo, como endereço IP e o número de porta TCP (EDWARDS, 2006). A ideia é prover uma forma de associar as necessidades de uma aplicação ou outro serviço cliente e as funcionalidades e capacidades do serviço (MAIA; ROCHA; ANDRADE, 2009).

A descoberta de serviços é o processo em que uma entidade (o cliente) busca pelos serviços desejados em mecanismos responsáveis por registrá-los ou é notificado da dispo-

nibilidade deles. Para tanto, um cliente interessado determina critérios, de acordo com seus interesses, que são usados em consultas no mecanismo de descoberta ou usados como filtros dos anúncios.

Muitos protocolos para descrição e descoberta de serviços já foram propostos e efetivamente implementados (ZHU; MUTKA; NI, 2005; EDWARDS, 2006). Todos esses protocolos realizam descobertas de serviços em ambientes computacionais em termos de localização e topologia de rede e foram projetados e desenvolvidos para sistemas distribuídos convencionais com um limitado conjunto de características. Nesses sistemas, os serviços na rede operam em um escopo limitado, protegido por *firewalls* e gerenciados por administradores de sistemas. Contudo, os sistemas ubíquos são, como discutido no requisito de coordenação, bastante dinâmicos. Novos dispositivos entram e saem do ambiente constantemente, movem-se, são desligados ou sofrem falhas, alterando constantemente o escopo da rede, e nem todos os serviços disponíveis são gerenciados por administradores de sistemas. Além disso, os dispositivos carregados pelos usuários e os agentes de software sendo executados nesses dispositivos podem possuir papel duplo, serem tanto clientes de serviços como prestadores de serviços e podem estar em níveis de administração diferentes. Outra questão relevante é o fato de que os sistemas ubíquos precisam ser projetados e desenvolvidos para que novos serviços possam ser adicionados no futuro à medida que mais usuários usem o sistema, mais dispositivos sejam usados pelos usuários e mais funcionalidades os dispositivos ofereçam.

Desse modo, um sistema ubíquo geralmente não apresenta uma arquitetura estática, onde se pode identificar a localização precisa de um dispositivo que provê um serviço desejado e, principalmente, quais são os serviços disponíveis em um dado momento. Mesmo que um serviço esteja sempre localizado em um dispositivo fixo no ambiente, um aplicativo ou serviço em um dispositivo recém chegado no ambiente não sabe necessariamente da sua disponibilidade, funcionalidade, localização e/ou de como invocá-lo. Além disso, mesmo que um cliente saiba da localização de um serviço e de como utilizá-lo, ele pode se mover de um dispositivo hospedeiro para outro para fins de otimização.

Portanto, para uma descoberta de serviços adequada em sistemas ubíquos, deve-se ter disponível uma forma flexível de expressá-los e disponibilizá-los no sistema ubíquo. A descrição de serviços tradicional foca, principalmente, nas funcionalidades do serviço e suas interfaces e são muito rígidas para representar todas as características dos serviços (MAIA, 2009). Por exemplo, uma determinada descrição de serviço fornece informações sobre parâmetros de entrada e saída, pré-condições, efeito e performance (LIU; PENG;

CHEN, 2006), endereço e porta, ou protocolo de acesso. Contudo, em ambientes ubíquos, devido à escassez de recursos em alguns dispositivos e falta de um conhecimento completo do estado do sistema, existem informações sobre os serviços que podem ser tão importantes quanto a sua funcionalidade propriamente dita. Assim, é necessário uma expressividade considerável na consulta e nas descrições dos serviços.

Uma vez que os serviços são descobertos, a sua invocação apresenta os mesmos problemas relacionados ao requisito de coordenação, pois se trata de uma interação entre agentes de software. Além disso, os clientes não sabem necessariamente quais são os parâmetros necessários nem como é o formato dos resultados, se houver. No *JINI*, por exemplo, o acesso ao serviço é realizado na forma de objetos *Java* que são migrados para o cliente. Entretanto, para o uso adequado, o cliente precisa ter um conhecimento prévio dos métodos e das assinaturas desses métodos no objeto *Java* recebido e, em um sistema ubíquo, isso nem sempre é possível.

Assim, uma infraestrutura de serviços que forneça mecanismos para a descrição, publicação, descoberta e invocação de serviços apropriados para o cenário da Computação Ubíqua se torna necessário. Tal infraestrutura tem que possibilitar formas flexíveis e expressivas de descrever e descobrir serviços, analisando as características e o contexto do cliente, do serviço e do sistema como um todo de forma a escolher o mais apropriado para a realização de determinada atividade com a mínima intervenção dos usuários. Ela deve representar sempre o estado atual do sistema referente aos serviços disponíveis, considerando a volatilidade inerente dos sistemas ubíquos. Além disso, essa infraestrutura precisa possibilitar que os serviços sejam invocados pelos clientes sem a necessidade de conhecimentos de protocolos de redes, qual o dispositivo que fornece o serviço e como ele é implementado (considerando plataforma de desenvolvimento e execução).

2.2.3 Interoperabilidade

A interoperabilidade em sistemas pode ser entendida como a capacidade de um componente se comunicar com outros componentes, compartilhando dados ou invocando processos independentes de sua plataforma, arquitetura, linguagem de programação ou sistema operacional (COULOURIS; KINDBERG; DOLLIMORE, 2005). As partes se comunicam de forma transparente independentemente da maneira como foram desenvolvidas e organizadas. O principal desafio é garantir que os componentes sejam desenvolvidos de forma que não sejam afetados pelo contexto computacional em que eles serão usados, permitindo seu uso em diferentes ambientes computacionais e aplicações. Portanto, se o contexto em

que um componente for implantado mudar, suas interconexões com outros são modificadas de acordo com o novo contexto.

Uma característica importante dos sistemas ubíquos é a possibilidade da presença de uma grande heterogeneidade de dispositivos computacionais (C5). Esses dispositivos vão de simples sensores a grandes computadores, passando por relógios, *smartphones*, *tablets* e *laptops*, que possuem diferentes funcionalidades (e.g., presença de acelerômetro, câmera fotográfica e sistema de posicionamento global) e capacidade computacional variada (e.g., tamanho de memória, poder de processamento e resolução de tela). Além da heterogeneidade dos dispositivos computacionais, existe a heterogeneidade de plataforma de desenvolvimento (e.g., *Sun Microsystems* JME, *Google Android*, .NET *Compact Framework* e *Apple* iOS SDK), linguagem de programação, (e.g., *Java*, *C#*, *Objective-C*), plataforma de execução (e.g., *Linux*, *Windows Mobile* e iOS) e tecnologia de comunicação (e.g., *Bluetooth*, *Zigbee* e *Wi-Fi*). Adicionalmente, esses dispositivos computacionais, plataformas, linguagens de programação e tecnologias de comunicação são providos por diferentes fornecedores que, em sua grande maioria, adotam padronizações diferentes.

Devido a essa heterogeneidade entre os agentes de software, que precisam coordenar suas atividades para prover serviços aos usuários, a interoperabilidade é necessária em todos os níveis em um sistema ubíquo (NIEMELä; LATVAKOSKI, 2004). No nível de aplicação, por exemplo, os agentes precisam descobrir serviços e recursos e se comunicar com diversos outros agentes. Entretanto, esses agentes podem ter sido desenvolvidos com uma grande variedade de plataformas de desenvolvimento, sistema de *middleware* ou linguagens de programação, ou possuírem interfaces de comunicação diferentes. Dessa forma, a interoperabilidade de plataforma de desenvolvimento e execução é extremamente necessária para garantir a comunicação entre esses agentes de software.

Assim, em sistemas ubíquos, as interações entre agentes necessárias na coordenação e na descoberta e invocação de serviços (e em outros requisitos) precisam ser independentes de plataformas de desenvolvimento e execução dos agentes envolvidos. Essa transparência é, geralmente, provida através de *middlewares* que utilizam uma das seguintes abordagens (COULOURIS; KINDBERG; DOLLIMORE, 2005):

- Aderir a padronizações de interfaces publicadas/abertas; e
- Utilizar serviços que convertem a interface do sistema para a interface de outro sistema em tempo de execução.

Em ambas as abordagens, existem protocolos formalizados que determinam as regras

que governam o formato, o conteúdo e o significado das mensagens trocadas entre os agentes. Nesses protocolos as operações são especificadas através de interfaces, que costumam ser descritas em uma linguagem de definição de interface (*Interface Description Language* - IDL) (TANENBAUM; STEEN, 2007). Definições de interfaces escritas em IDL quase sempre capturam apenas a sintaxe das operações. Elas, geralmente, especificam os identificadores das operações disponíveis, os parâmetros, os valores de retorno e as possíveis exceções que podem surgir. A parte difícil é especificar com precisão a semântica das operações, que, na prática, são feitas de modo informal por meio de linguagem natural (TANENBAUM; STEEN, 2007). Contudo, linguagens formais e semi-informais, como lógica de primeira ordem, diagramas de sequência e de estados da UML e OCL são utilizadas em alguns contextos (CLEMENTS et al., 2003).

Esses protocolos, se adequadamente especificados, permitem que um agente arbitrário, que necessite de uma certa operação, se comunique com outro agente que fornece aquela operação. Contudo, para se atingir a interoperabilidade de plataformas de desenvolvimento e execução, tal protocolo deve ser especificado de forma que duas partes independentes construam implementações totalmente diferentes que são executadas em plataformas também diferentes e ainda assim possam se comunicar apropriadamente.

Um bom exemplo da primeira abordagem é o conjunto de padronizações que foram desenvolvidos para a *Web* (e.g., TCP/IP, HTTP e HTML). Como exemplo da segunda, podemos citar o *Common Object Request Broker Architecture* (CORBA) (OMG, 2011) e seu *Object Request Broker* (ORB) (COULOURIS; KINDBERG; DOLLIMORE, 2005). Essas duas abordagens são comumente utilizadas através de sistemas de *middleware* que abstraem a implementação de camadas mais inferiores no software. Os agentes se comunicam com o *middleware* através de uma interface de mensagens padronizadas e o *middleware* as repassa para um sistema de suporte o qual realizará as devidas ações.

Contudo, a tecnologia que virou sinônimo de interoperabilidade são os *web services* (ALONSO et al., 2004). Os *web services* podem ser definidos como um tipo de arquitetura baseada em protocolos abertos (e.g., HTTP e SOAP), funcionando e respondendo à requisições HTTP com mensagem XML no formato SOAP, vindas de qualquer ponto e plataforma conectados na mesma rede. Assim, os *web services* garantem a interoperabilidade através do uso de tecnologias padronizadas e abertas que possuem implementações apropriadas para cada plataforma envolvida. Dessa forma, para uma aplicação desenvolvida em uma plataforma **A** usar um serviço construído em uma plataforma **B**, devem existir implementações dessas tecnologias para que ambas consigam "conversar" indepen-

dente de suas plataformas, atingindo, assim, a interoperabilidade. Outras variações de *web services* foram desenvolvidas visando maior performance e simplicidade. O XML-RPC[1] foi uma das primeiras e se baseia também em XML, mas permite outras formas de transmissão dos dados além de HTTP. O JSON-RPC[2] é similar ao XML-RPC, porém a representação de dados usando o JSON é menor que o XML (ERFIANTO; MAHMOOD; RAHMAN, 2007), o tornando, assim, mais eficiente na transmissão dos dados.

2.2.4 Sensibilidade ao Contexto

Uma característica importante dos sistemas ubíquos é o fato de que é impossível conceber um que não seja sensível ao contexto (COUTAZ et al., 2005). Pelo fato de que tais sistemas tem que operar com a mínima intervenção do usuário (C7), eles precisam alterar seu comportamento e/ou estado de acordo com informações contextuais, obtidas das mais variadas formas (SATYANARAYANAN, 2001). Além disso, os serviços oferecidos devem se adaptar para diferentes tipos de dispositivos e de infraestruturas de redes, bem como para o estado em que cada um se encontra, além das características do ambiente e dos usuários. Dessa forma, eles devem possuir um comportamento adaptável (C1), que só é possível através de informações de contexto obtidas tanto na implantação quanto na execução. Portanto, em sistemas ubíquos, devem existir mecanismos que permitam a captura, o processamento e a distribuição de informações contextuais de interesse entre os agentes.

Definições e caracterizações variadas para contexto já foram propostas para torná-lo um conceito operacional em termos computacionais. Mary Bazire e Patrick Brézillon apresentam 150 definições diferentes (BAZIRE; BREZILLON, 2005). Essas variações são influenciadas à medida que se transita entre diferentes domínios e apesar da variedade de definições, podemos destacar em comum: (1) o contexto está sempre associado com alguma entidade, como pessoas, tarefas e lugares; (2) o contexto é um conjunto de itens associados a uma entidade, como conceitos, regras e proposições; e (3) um item é considerado como parte de um contexto se for útil para apoiar na realização da tarefa a ser desempenhada (BAZIRE; BREZILLON, 2005).

Portanto, o contexto de uma entidade, seja ela um objeto, uma pessoa ou um lugar, é um aspecto de suas circunstâncias, tanto físicas quanto lógicas, que tenha relevância para determinar o comportamento ou estado do sistema. Isso inclui valores relativamente

[1] `http://www.xmlrpc.com/`
[2] `http://json-rpc.org/`

simples tais como localização; tempo ou espaço de tempo; temperatura, identificação e preferências de um usuário, que pode ser do usuário utilizando o sistema ou usuários próximos tanto em relação à localização quanto ao tempo. Contexto também pode ser caracterizado por atributos mais complexos, como atividades que estão sendo desempenhadas pelo usuário. Por exemplo, um telefone celular sensível ao contexto quando recebe uma ligação, verifica se o usuário está em uma reunião ou no cinema para decidir se o avisa com alertas sonoros ou não.

Uma análise dos trabalhos de Matthias Baldauf *et. al.* (BALDAUF; DUSTDAR; ROSENBERG, 2007), Anusuriya Devaraju *et. al.* (DEVARAJU; HOH; HARTLEY, 2007), Anind K. Dey (DEY, 2001) e Daniel Salber *et. al.* (SALBER; DEY; ABOWD, 1999) permite enumerar as principais dificuldades com a Computação Sensível ao Contexto:

1. As informações de contexto são obtidas de sensores não convencionais. Dispositivos móveis como telefones celulares, por exemplo, devem adquirir informações de localização através de dispositivos GPS externos ou internos ou usando técnicas de triangularização de sinais de antenas. Detectar a posição *indoor* de pessoas e a sua movimentação pode exigir o uso de crachás inteligentes, sensores de presença ou processamento de imagens de vídeo;

2. As informações contextuais geralmente são usadas em um grau maior de abstração. Por exemplo, em sistemas de guias móveis, as coordenadas dos visitantes são fornecidas por um serviço de triangularização de sinais no formato `(x,y)`, baseado em algum ponto de origem no museu. Porém, a aplicação de guia de visitas pode fazer melhor uso, informando nomes dos setores e salas onde aquela coordenada está relacionada;

3. Às vezes, as informações de contexto são obtidas de fontes distribuídas e heterogêneas. Para saber de uma localização *indoor*, pode ser necessário obter informações de vários sensores e combinar o uso de diversas técnicas como processamento de imagens e de áudio, isto é, fazer uma agregação ou composição de informações seguindo certa lógica e requisito da aplicação;

4. As informações de contexto são dinâmicas. Mudanças no ambiente devem ser detectadas em tempo real e as aplicações devem processar essas constantes mudanças. Em sistemas de guias móveis, as informações são mostradas de acordo com a sala em que o usuário se encontra. O histórico dessas mudanças também pode ser valioso, como por exemplo, fazer avaliações de preferências do usuário; e

5. As aplicações e serviços podem estar interessados em somente uma porção do contexto disponível. Portanto, deve ser possível especificar um subconjunto dos dados contextuais disponíveis para facilitar a solução das dificuldades anteriores.

Baseado nisso, pode-se destacar sete elementos funcionais necessários em uma infraestrutura de software que auxilie sistemas sensíveis ao contexto:

Representação do contexto: a forma como os dados contextuais são representados;

Persistência: onde e como os dados contextuais são armazenados;

Aquisição de contexto: como os dados contextuais são fornecidos para os agentes interessados;

Agregação: como os dados contextuais obtidos são mesclados para formar novos dados contextuais;

Busca: como os dados contextuais são pesquisados e selecionados;

Notificação: como a mudança no estado dos dados contextuais ou a existência de novos são informadas aos interessados; e

Visões: como o conjunto de dados contextuais de interesse é definido.

Essa infraestrutura de software pode ser implementada de forma genérica para uso nos mais diversos sistemas ubíquos. Matthias Baldauf *et. al.* (BALDAUF; DUSTDAR; ROSENBERG, 2007) e Anusuriya Devaraju *et. al.* (DEVARAJU; HOH; HARTLEY, 2007) apresentam um *survey* das principais soluções para as dificuldade enumeradas. Assim, os agentes podem usar tal infraestrutura para persistir e pesquisar informações contextuais dos mais variados formatos para a realização adequada de suas funcionalidades. Porém, em sistemas ubíquos, tal infraestrutura precisa se preocupar também com os problemas relacionados ao desacoplamento e a interoperabilidade nas interações, pois os produtores e os consumidores de informações contextuais são voláteis e heterogêneos.

Essa infraestrutura de contexto possui características semelhantes aos mecanismos de coordenação e de descrição e descoberta de serviços abordados anteriormente tais como: meios proativos e reativos de interação, mecanismos expressivos de pesquisa e interoperabilidade na comunicação. Assim, uma mesma solução pode servir para ambos. O *LIME/LighTS* (MURPHY; PICCO; ROMAN, 2006; BALZAROTTI; COSTA; PICCO, 2007) foi desenvolvido usando esse pensamento, porém seu mecanismo de descoberta de

serviços (HANDOREAN; ROMAN, 2007) não possui os recursos necessários discutidos na seção 2.2.2 e não é interoperável.

2.2.5 Invisibilidade

A Invisibilidade é a principal característica mencionada por Mark Weiser no seu trabalho (WEISER, 1991). De certa forma, ela pode ser tratada como uma proatividade do sistema, de forma que ele realize suas funcionalidades com a mínima intervenção do usuário (C7) (SATYANARAYANAN, 2001). Essa proatividade é baseada nas intenções e experiências do usuário, no contexto e em adaptações inferidas a partir de regras especificas relacionadas as atividades desempenhadas. Assim, são necessários mecanismos que capturem essas intenções, bem como o contexto que possa influenciar a adaptação e o serviço prestado pelo sistema.

Esse requisito se relaciona com o objetivo dos sistemas ubíquos de esconder a infraestrutura computacional, que está embutida no ambiente, dos usuários para que esses foquem exclusivamente nas atividades sendo realizadas. Eunju Kim *et. al.*, por exemplo, propõem uma técnica de reconhecimento de atividades humanas simples (KIM; HELAL; COOK, 2010) e David Garlan *et. al.* (GARLAN et al., 2002) propõem um mecanismo para representar as intenções dos usuários em tarefas computacionais realizadas por um conjunto de serviços.

O requisito da Invisibilidade é muito específico de cada sistema e se relaciona muito com conceitos de inteligência artificial e interações homem-máquina. Contudo, para a sua implementação devem existir mecanismos de obtenção e consulta das informações contextuais necessárias bem como mecanismos de troca de informações entre os agentes, para que esse requisito seja apropriadamente implementado e executado.

2.2.6 Autonomicidade

Sistemas autônomos são sistemas auto gerenciados constituídos por agentes que apresentam certo nível de independência, que tomam decisões baseadas em informações contextuais ou não, para se manter sempre em um estado desejável (HUEBSCHER; MCCANN, 2008). Para tanto, tais sistemas lidam com características do ambiente como dinamicidade, heterogeneidade e incerteza, tomando decisões baseadas em parâmetros de configuração fornecidos pelo administrador do sistema (STERRITT et al., 2005). Uma importante característica dessa autonomia é a habilidade do sistema de continuar for-

necendo os serviços mesmo sofrendo ataques, falhas ou acidentes. Assim, através dessa autonomicidade, o sistema se mantem com certo nível de segurança, desempenho, confiabilidade, disponibilidade e robustez (NIEMELä; LATVAKOSKI, 2004), características que em conjunto são chamadas de dependabilidade.

O requisito de Autonomicidade mantém estreitas relações com os requisitos de sensibilidade ao contexto e é normalmente caracterizado através de quatro atributos (HUEBSCHER; MCCANN, 2008):

Autoconfiguração: o sistema modifica seu comportamento e/ou sua organização de acordo com políticas pré-estabelecidas e informações contextuais relevantes sem o controle de algum sistema externo;

Auto-otimização: o sistema, continuamente, procura formas de aperfeiçoar e evoluir o comportamento para tornar-se mais eficiente em termos de desempenho e custo;

Autocura: o sistema deve detectar, diagnosticar e reparar problemas resultantes de falhas de software e hardware ou ainda decorrentes da falta de recursos como baixo nível de bateria ou alta latência da rede; e

Autoproteção: o sistema deve se defender contra ataques maliciosos ou falhas em cascata e ainda antecipando-se a possíveis problemas baseado em informações contextuais.

Além disso, devido ao grande número de dispositivos, aplicações e serviços, que formam um sistema ubíquo e a mobilidade dos mesmos, exigem que tais sistemas sejam escaláveis para suportar tal demanda. Assim, os ambientes ubíquos podem crescer em sofisticação oferecendo novos serviços e informações, aumentando também as interações dos usuários com o ambiente.

Portanto, mesmo que tais políticas sejam específicas de cada sistema, definidas pelos seus requisitos não-funcionais e atributos de qualidade de serviço (QoS), esse requisito precisa de um sistema de suporte que ofereça mecanismos para implementar essas políticas. As funcionalidades desse sistema de suporte estariam relacionados à interação e coordenação de atividades entre os agentes e a captura das informações contextuais necessárias.

2.3 Conclusão

Este capítulo teve como finalidade definir um sistema ubíquo através da enumeração de suas características essenciais e do levantamento de um conjunto de requisitos de software genéricos necessários para o seu desenvolvimento. Esse levantamento, tanto de características como de requisitos, foi realizado tomando como base trabalhos com finalidades semelhantes a esta dissertação. É importante destacar que esses trabalhos possuem finalidades semelhantes, mas com visões divergentes ou incompletas sobre as características e requisitos, quando comparados entre si, o que foi essencial para definir as contribuições desta dissertação.

O principal ponto levantado neste capítulo é a dinamicidade e a heterogeneidade apresentadas pelos sistemas ubíquos. Essas características influenciam consideravelmente os requisitos de sistemas presentes no desenvolvimento de sistemas ubíquos como a coordenação e a descrição/descoberta de serviços. Contudo, soluções são possíveis para esses problemas, como a adoção de uma infraestrutura de serviços e de comunicação que seja desacoplada, sensível ao contexto e interoperável. Tal infraestrutura, pelo exposto, pode ser usada também para o desenvolvimento de soluções para os requisitos de sensibilidade ao contexto, invisibilidade e autonomicidade.

Essas características e requisitos foram levantados com a principal finalidade de especificar e construir soluções de software que auxiliem o projetista e desenvolvedor na construção e manutenção de sistemas ubíquos. Nesse contexto, soluções já foram desenvolvidas e algumas delas são apresentadas no próximo capítulo.

3 Trabalhos Relacionados

Várias infraestruturas de software foram projetadas e desenvolvidas para facilitar o desenvolvimento de sistemas ubíquos e pervasivos. Uma grande parte delas é descrita no *survey* de Christoph Endres *et. al.* (ENDRES; BUTZ; MACWILLIAMS, 2005). Outras soluções foram propostas para auxiliar o desenvolvimento de outros tipos de sistemas distribuídos, que não apresentam todas as características de sistemas ubíquos, como sistemas móveis e sensíveis ao contexto, mas que podem ser utilizadas no desenvolvimento de sistemas ubíquos.

Este capítulo está dividido inicialmente em seis seções onde são apresentadas seis infraestruturas consideradas mais abrangentes, pois oferecem suporte a um conjunto maior dos requisitos: *AURA*, *Gaia*, *one.world*, *LIME/LighTS*, *Egospace* e *TOTA*. As três primeiras oferecem suporte a vários requisitos e características apresentados no capítulo 2 e as três últimas focam em sistemas móveis e sensíveis ao contexto, mas podem ser utilizadas no desenvolvimento de sistemas ubíquos. Por fim, na seção 3.7 é apresentado uma conclusão da análise dessas infraestruturas em relação aos requisitos discutidos no capítulo 2.

3.1 AURA

Baseado na premissa que os recursos mais preciosos em um sistema computacional não são mais processadores, memória, espaço em disco ou largura de banda, e sim a atenção do usuário, o projeto *AURA* (GARLAN et al., 2002) se preocupa em minimizar distrações nessa atenção criando um ambiente que se adapta automaticamente ao seu contexto e suas necessidades. Mais precisamente, essa atenção se refere a habilidade do usuário em se concentrar em suas atividades primárias, ignorando distrações geradas pelo sistema computacional, como baixa performance e falhas. Tal atenção é o recurso mais precioso em ambientes ubíquos, pois os usuários estão preocupados em andar, dirigir, ou interagir com o mundo físico. Além disso, a mobilidade impõe mais restrições como conexões intermitentes e de banda variável e existem limitações nos dispositivos, como

bateria e poder de processamento, que podem vir a distrair o usuário.

Para o *AURA* atingir tal objetivo, ele se preocupa com todos os níveis de um sistema computacional, do hardware até os usuários finais, passando pelo sistema operacional e aplicações. Ele se baseia em dois principais conceitos que juntos minimizam a necessidade de distrações:

Proatividade: a habilidade de uma camada do sistema de, antecipadamente, saber as requisições da camada superior; e

Auto-otimização: cada camada sabe se adaptar, observando as demandas impostas, ajustando sua performance e as formas de utilização de seus recursos.

Para tanto, as maiores capacidades do *AURA* são seu suporte a mobilidade do usuário e a proteção fornecida a eles contra as variações na disponibilidade dos recursos. Quando um usuário se move de um ambiente para outro, o *AURA* se preocupa em reconfigurar o novo ambiente de forma que ele possa continuar a execução das tarefas iniciadas no ambiente anterior. À medida que os recursos de um ambiente mudam (como a largura de banda disponível), o *AURA* se preocupa em se adaptar e adaptar as novas tarefas dos usuários de acordo com as mudanças, possivelmente reconfigurando certos serviços ou até mesmo trocando os serviços utilizados por outros mais apropriados.

A Figura 3.1 apresenta a arquitetura geral do AURA. O *Odyssey* (NOBLE; PRICE; SATYANARAYANAN, 1995; NOBLE; SATYANARAYANAN, 1999) oferece suporte ao monitoramento de recursos e adaptação sensível ao contexto a aplicações, e o *CODA* (SATYANARAYANAN et al., 1990) provê suporte de acesso a arquivos de forma nômade, desconectado e adaptável. Eles são reutilizados de outros trabalhos adaptando-os para o contexto da computação ubíqua. Já o componente *Spectra* é um mecanismo de execução remota adaptável que usa informações contextuais para decidir a melhor forma de executar uma chamada remota.

Além disso, o *AURA* provê vários e úteis componentes de software para atingir seus objetivos, como: *cyber foraging*, que usa o *CODA* para o acesso a arquivos quando os recursos dos dispositivos são limitados; o *wireless bandwidth advisor*, que auxilia na determinação da melhor conexão para os usuários, e o *WaveLAN-based people locator*, utilizado para localização de usuários.

Contudo, o principal componente no *AURA* é uma nova camada, a *task layer*, situada em cima da camada das aplicações e serviços, mas embaixo dos usuários, implementada pelo componente *Prism*. Essa camada é responsável pelas principais capacidades

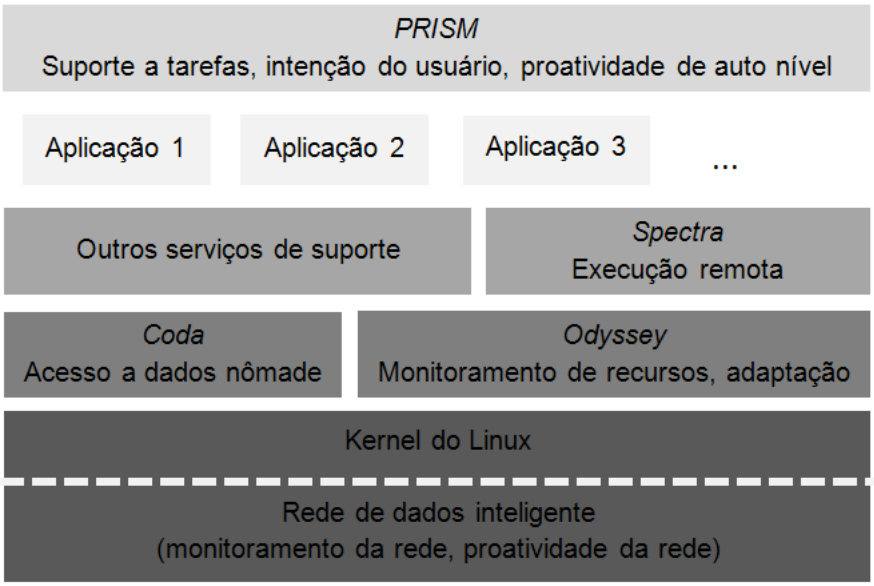

Figura 3.1: Arquitetura do AURA. (Adaptado de (GARLAN et al., 2002)).

do *AURA* discutidas anteriormente. Ela representa as intenções dos usuários através de tarefas que são disponibilizadas para o resto do sistema formando uma base que serve para adaptações e para antever as necessidades dos usuários. Tais tarefas são formadas no sistema por uma coligação de serviços disponíveis e administradas de acordo com o contexto do ambiente.

O *AURA* é uma elegante e expressiva solução que possui foco principalmente nas características de minima intervenção do usuário, onipresença dos serviços, autonomicidade e comportamento adaptável. Ele representa tanto um *framework* como um *middleware*, além de um conjunto de serviços, e, assim, seu uso impõe algumas restrições à arquitetura do sistema ubíquo sendo desenvolvido limitando a flexibilidade da sua utilização. Na sua descrição, nada é mencionado sobre a coordenação entre os agentes no sistema, métodos de agregação e como os serviços são descritos e pesquisados. Contudo, a sua principal desvantagem é não possuir uma especificação ou implementação que propicie interoperabilidade.

A solução proposta neste trabalho não oferece tantos recursos como o *AURA*. Ele se concentra nos aspectos mais básicos e fundamentais para a construção de sistemas ubíquos, como as interações entres os agentes considerando as requisitos impostos. Ela tem o propósito, também, de servir de base para o desenvolvimento de soluções mais completas que ofereçam mais recursos. Como exemplo, o próprio *AURA* poderia ser refatorado utilizando essa solução como base.

3.2 Gaia

O *Gaia* (ROMAN et al., 2002), assim como este trabalho de dissertação, se baseia no fato de que não existem infraestruturas de software adequadas para desenvolver aplicações para ambiente ubíquos. Ele é representado como um meta-sistema operacional (KON et al., 1998) construído como uma infraestrutura de *middleware* distribuída que coordena entidades de software e dispositivos heterogêneos conectados em rede em um ambiente físico.

O *Gaia* expõe um conjunto de serviços, os quais usam os recursos existentes e informações contextuais, para serem encontrados e utilizados pelas aplicações. Além disso, ele provê um *framework* para desenvolver aplicações móveis, centrada no usuário, sensível a recursos, multidispositivo e sensível ao contexto. Contudo, a principal contribuição do *Gaia* não é o conjunto de serviços que ele provê, mas sim as funcionalidades resultantes da interação desses serviços.

O *Gaia* possibilita aos seus usuários configurar suas aplicações para se beneficiar dos recursos disponíveis no ambiente. Eles podem interagir com múltiplos dispositivos simultaneamente, reconfigurar as aplicações dinamicamente, suspender e retomar grupos de aplicações e definir o comportamento das aplicações de acordo com o contexto. Essa interação permite que usuários e desenvolvedores abstraiam os ambientes de computação ubíqua como uma entidade reativa e programável em vez de uma coleção de dispositivos heterogêneos. Para isso, o *Gaia* é baseado no conceito de *active spaces* que é um ambiente físico coordenado por uma infraestrutura de software sensível ao contexto que aumenta a habilidade dos usuários móveis em interagir e configurar suas aplicações em uso de forma transparente.

Em um *active space*, as sessões associam dados do usuário e aplicações com os usuários. Isso permite que eles se movam no ambiente ubíquo e tenham seus dados e aplicações sempre acessíveis. Quando um usuário entra em um *active space*, a sua sessão é dinamicamente carregada e mapeada para os recursos disponíveis. Nos *active spaces*, o relacionamento um-para-um entre o usuário e o mouse, o teclado e o *display* não existe. Ao invés disso, a complexidade das aplicações ubíquas encoraja um relacionamento entre ele e o *active space*.

De uma forma geral, o *Gaia* administra os recursos e serviços presentes em um *active space*, provê localização e informações contextuais, coordena serviços através de eventos e armazena os dados relacionados aos próprios *active spaces* e usuários. A arquitetura,

ilustrada na Figura 3.2, é formada por três componentes principais: o *kernel*, o *framework* de aplicações e as aplicações.

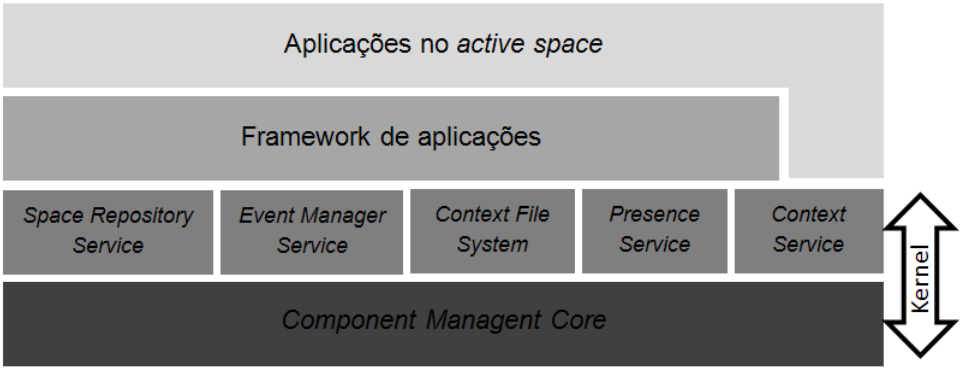

Figura 3.2: Arquitetura do **Gaia**. Adaptado de (ROMAN et al., 2002).

O principal componente dessa arquitetura é o *kernel*. Ele é formado pelo *component management core* e por um conjunto de serviços básicos que são usados por todas as aplicações. O *component management core*, dinamicamente, carrega, descarrega, transfere, cria e destrói todos os serviços e aplicações em um *active space*. Os serviços básicos disponibilizados são:

***Event manager*:** responsabiliza-se por distribuir os eventos no *active space* através de canais e, assim, permitindo uma coordenação desacoplada (referencial e assíncrono) entre os agentes;

***Context service*:** permite que aplicações pesquisem por informações contextuais necessárias para adaptar o seu comportamento, além de possibilitar subscrição de eventos contextuais;

***Presence service*:** responsabiliza-se por detectar entidades digitais e físicas no ambiente (as principais são: aplicações, serviços, dispositivos e pessoas);

***Space repository*:** armazena informações sobre todas as entidades de software no *active space*, como nome, tipo, dono, etc., e provê funcionalidades para pesquisa dessas informações;

***Context file system*:** incorpora contexto no modelo de sistema de arquivos tradicional para prover suporte a usuários móveis, dispositivos heterogêneos e a uma melhor organização dos dados.

Das infraestruturas estudadas, o *Gaia* apresenta a infraestrutura de software mais completa. Apesar de somente a coordenação por eventos ser mencionada, ele disponibiliza

de forma indireta um desacoplamento espacial e temporal, mesmo que não tão abrangente, através do *space repository* e do *context file system*. Além disso, possui uma infraestrutura de contexto considerável, fornecendo recursos avançados para pesquisa (RANGANATHAN; CAMPBELL, 2003; RANGANATHAN; AL-MUHTADI; CAMPBELL, 2004) e uma infraestrutura de serviços baseada em componentes com vários recursos. Tudo isso se preocupando com interoperabilidade (KON et al., 2000).

Entretanto, o *Gaia* não oferece, por completo, os requisitos descritos no capítulo 2. Em relação ao serviço de descoberta, ele não implementa nenhum mecanismo automático e se apoia no serviço de nomes do *CORBA*. Assim, não é possível realizar descrições e pesquisas mais flexíveis. Apesar do mecanismo de contexto ser baseado em lógica de primeira ordem e álgebra boleana e, assim, permitir maior expressividade nas consultas, nenhum mecanismo de agregação é fornecido.

Assim como o *AURA*, o foco principal do *Gaia* é prover uma infraestrutura de software completa para o desenvolvimento de sistemas ubíquos, fornecendo recursos básicos e serviços avançados. Contudo, seu uso é limitado em abrangência de problemas, pois exige que os desenvolvedores usem implementações prontas, como o *context file system*, sobre tecnologias pré-definidas, como o *CORBA*. A diferenciação realizada entre o *AURA* e a solução proposta nesta dissertação também se aplica ao *Gaia*. Além disso, a solução proposta se preocupa com a flexibilidade de uso, não se atrelando a determinadas tecnologias e nem impondo uma arquitetura rígida.

3.3 *one.world*

O *one.world* (GRIMM et al., 2004) representa uma arquitetura, um *framework* e um modelo de programação destinado ao desenvolvimento de aplicações para ambientes pervasivos. Ele se baseia, principalmente, em três requisitos os quais os autores consideram característicos dos sistemas pervasivos:

1. Não esconder das aplicações as mudanças no contexto físico e computacional, tornando-as uma parte essencial do sistema;

2. Incentivar composições *ad-hoc* entre agentes e dispositivos e não assumir um ambiente computacional estático com um número limitado de interações; e

3. Reconhecer a importância do compartilhamento de informações.

Os autores argumentam que as técnicas convencionais de desenvolvimento de sistemas distribuídos não contemplam esses três requisitos ao mesmo tempo. Isso se deve principalmente ao fato que essas tecnologias escondem os detalhes da distribuição, enquanto que para atender aos três requisitos, as mudanças devem ser explicitas para que as aplicações realizem adaptações necessárias. Assim, para o *one.world*, as aplicações devem perceber as mudanças e se adaptar em relação a elas, ao contrário de forçar os usuários a repetidamente reconfigurar suas aplicações. A Figura 3.3 esquematiza essas ideias.

Figura 3.3: Caracterização do *one.world*. Adaptado de (GRIMM et al., 2004).

Baseados nos três requisitos citados, a arquitetura do *one.world*, ilustrada na Figura 3.4, é construída baseada em quatro princípios:

1. Um conjunto de serviços fundamentais destinados a atender aos três requisitos;
2. Serviços de sistemas específicos, construídos a partir dos serviços fundamentais disponíveis como blocos comuns para a construção de aplicações;
3. Utilizar os conceitos de *user/kernel* (COULOURIS; KINDBERG; DOLLIMORE, 2005) com os serviços fundamentais e de sistemas sendo providos pelo *kernel* e as bibliotecas, utilitários de sistemas e aplicações sendo executadas no espaço pervasivo do usuário (*User Space*);
4. Essa implementação deve ser neutra em relação a outros requisitos, por exemplo, se vai ser usado cliente/servidor ou P2P.

O *one.world* utiliza o conceito de tuplas e eventos para a troca de informações e coordenação dos agentes, possibilitando assim uma coordenação totalmente desacoplada. Contudo, apesar de ser uma solução baseada principalmente em informações contextuais, o suporte a pesquisas por tuplas mais avançada e agregações de tuplas não existem e seu mecanismo de descrição e descoberta de serviços é limitado. Além disso, a interoperabilidade não é totalmente atendida, sendo basicamente dependente de uma única máquina virtual como *Java* ou *Microsoft Common Language Runtime*.

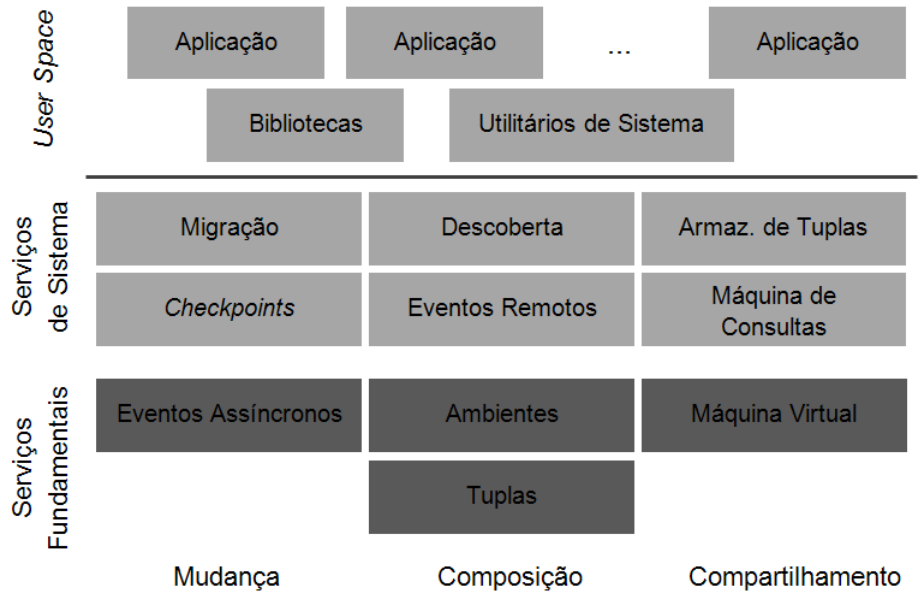

Figura 3.4: Arquitetura do ***one.world***. Adaptado de (GRIMM et al., 2004).

A solução proposta nesta dissertação possui algumas semelhanças com o *one.world*, sendo a principal que ambos representam um sistema de suporte. Eles representam infra-estruturas de softwares que oferecem recursos e funcionalidades reutilizáveis importantes para o desenvolvimento de sistemas ubíquos sem impor maiores restrições à arquitetura do sistema. Mesmo que o *one.world* seja considerado um *framework* também, sua especificação proporciona certo nível de flexibilidade em relação ao *AURA* e ao *Gaia*. Além disso, ambos são baseados nos conceitos de tuplas e eventos. Contudo, a solução proposta se diferencia por fornecer mais recursos nos mecanismos de serviços e de coordenação e por ser interoperável.

3.4 LIME/LighTS

O *LIME* (MURPHY; PICCO; ROMAN, 2006) é um mecanismo de coordenação para sistemas baseados em redes *ad-hoc* que apresentam tanto mobilidade física como lógica, baseado no modelo de coordenação *Linda*. Além disso, oferece suporte ao mecanismo de eventos através de reações a mudanças no estado dos espaços de tuplas. Ele foi desenvolvido de acordo com as seguintes premissas:

- Os problemas da mobilidade, seja física ou lógica, podem ser resolvidos com soluções de coordenação;
- Um mecanismo de coordenação desacoplado é o ideal para ambientes com intensa

volatilidade;

- O mesmo mecanismo usado para comunicação e coordenação entre os agentes em um sistema pode ser usado para compartilhar informações contextuais.

A principal característica do *LIME* é o compartilhamento de tuplas de forma distribuída e transiente. No sistema, cada agente possui um espaço de tuplas próprio, acessado pelo agente mesmo quando desconectado. Quando os agentes estão aptos a se comunicar de forma *ad-hoc* e utilizam o *middleware* do *LIME*, os vários espaços de tuplas dos agentes formam um único e global espaço de tuplas. Cada agente no sistema acessa esse espaço de tuplas lógico como se fosse único. Dessa forma, esse modelo possibilita um desacoplamento na comunicação, mesmo em redes móveis *ad-hoc*.

O principal componente do *LIME* é o *LighTS* (BALZAROTTI; COSTA; PICCO, 2007), que representa o espaço de tuplas em si. O *LighTS* implementa todas as características do requisito de sensibilidade ao contexto de forma simples e extensível, porém não interoperável. Ele usa o próprio conceito de tuplas e espaços de tuplas, usados como veículos de coordenação, para representar os dados contextuais e executar operações.

Apesar do foco do *LIME/LighTS* ser a coordenação de agentes móveis em redes *ad-hoc*, ele se mostra como uma solução elegante para os problemas de sensibilidade ao contexto, pois oferece expressividade, agregações e visões. Além disso, pelo fato de ser um *framework*, possibilita sua extensão incluindo novas funcionalidades e pode ser usado para descoberta de serviços (HANDOREAN; ROMAN, 2007).

Porém, assim como as soluções que empregam espaços de tuplas distribuídos e transientes, o desacoplamento temporal tem que ser feito pelos próprios agentes, visto que quando um agente sai do sistema ele leva consigo as tuplas presentes em seu espaço de tuplas. Além disso, o mecanismo de descrição e descoberta de serviços que usa o *LIME* como base apresenta pouca expressividade. Nesse mecanismo, o uso dos serviços não é realizado de forma desacoplada, um *proxy* do serviço é transferido para o cliente, exigindo mecanismos especiais quando o serviço é movido de um dispositivo para outro. Além disso, o *LIME/LighTS* possui a desvantagem de não ser interoperável e não apresenta formas de implementá-lo dessa maneira.

A solução proposta neste trabalho possui várias semelhanças com o *LIME/LighTS*: baseado em tuplas e espaços de tuplas, mecanismo de eventos baseados em tuplas e o uso de tuplas para representar dados contextuais. Contudo, a solução apresenta formas mais flexíveis de consulta aos espaços de tuplas e na descoberta de serviços, além de ser

interoperável. Porém, possui a desvantagem de não ser aplicável em sistemas baseados em redes *ad-hoc*.

3.5 Egospace

O *Egospace* (JULIEN; ROMAN, 2006) é uma solução destinada a sistemas pervasivos, móveis e sensíveis ao contexto, onde as aplicações e serviços não precisam conhecer os outros agentes presentes no sistema, permitindo, assim, uma comunicação desacoplada. Ele é baseado no fato de que sistemas com tais características apresentam dificuldades em gerenciar a grande quantidade de dados contextuais e a dinamicidade do ambiente. Assim, seu principal objetivo é "esconder" os detalhes da mobilidade, da distribuição e da conectividade transiente.

No *Egospace* é assumido o seguinte modelo de computação: um sistema é formado por dispositivos que podem se mover no ambiente físico e por uma comunidade de agentes de software móveis que podem migrar de um dispositivo para outro. Um agente é a unidade de modularidade e mobilidade, enquanto um dispositivo é um *container* de agentes que é caracterizado, entre outras coisas, por sua localização física. A comunicação entre agentes e a migração pode acontecer sempre que os dispositivos que os contêm podem se comunicar. Por fim, um conjunto de dispositivos próximos e conectados formam uma rede móvel *ad-hoc*. Esse modelo também pode ser usado para o *LIME/LighTS* descrito anteriormente bem como o *TOTA*, descrito na próxima seção.

O *Egospace* fornece abstrações que facilitam o desenvolvimento de tais agentes, criando uma infraestrutura para computação sensível ao contexto que facilita o desenvolvimento de agentes móveis adaptativos. Em princípio, as informações contextuais percebidas por um agente consistem de toda informação disponível no sistema. Toda essa disponibilidade de dados torna difícil sua administração e uso por um agente de forma isolada quando o sistema é muito grande. O *middleware* do *Egospace* permite que seja definido uma limitação aos dados de contexto que o agente interage de acordo com suas necessidades a partir de várias fontes heterogêneas. Tal *middleware* se responsabiliza por toda essa administração, deixando para o desenvolvedor somente as preocupações do que fazer com esses dados.

Basicamente, o *Egospace* é fundamentado nas seguintes características:

- O contexto deve ser generalizado de forma que os agentes interajam com diferentes tipos de forma similar;

- Diferentes agentes necessitam de dados contextuais baseados em suas necessidades particulares;
- As informações contextuais podem ser coletadas de forma distribuída e sem o conhecimento prévio das fontes; e
- Devido a ambientes com alta dinamicidade e escalabilidade, os agentes necessitam de interações descentralizadas com o contexto.

A Figura 3.5 representa a arquitetura básica de um dispositivo usando o *middleware* do *Egospace*. Um dispositivo possui uma localização física e um perfil descrevendo suas propriedades. Um agente também possui um perfil e uma localização, o qual é o dispositivo em que ele está veiculado. O *Egospace* também usa o conceito de espaços de tuplas através de um compartilhamento transiente semelhante ao *LIME*. Nessa arquitetura, o destaque são para as *Views*. Uma *View* é uma projeção de toda informação contextual percebida pelo agente. Cada agente pode definir quantas *Views* desejar e elas podem ser redefinidas posteriormente de acordo com suas necessidades. Além disso, as *Views* são mantidas dinamicamente em face das constantes mudanças que podem ocorrer no sistema, mudando o conjunto global das informações que a *View* atua.

Figura 3.5: Arquitetura do *Egospace*. Adaptado de (JULIEN; ROMAN, 2006).

Em resumo, o *Egospace* apresenta uma solução elegante para sistemas móveis e sensíveis ao contexto com um modelo de programação com considerável expressividade e ao mesmo tempo simples. Entretanto, apesar de possuir métodos de pesquisa das informações contextuais expressivos, ele não atende ao requisito de sensibilidade ao contexto

completamente. Por exemplo, as restrições só se aplicam a campos isolados e não na tupla como um todo e não existem mecanismos de agregação. Além disso, não existe o desacoplamento temporal, visto que quando um agente sai do sistema ele leva consigo as tuplas presentes em seu espaço de tuplas. Por fim, apesar de existir uma formalização de sua semântica, não é possível implementá-lo de forma interoperável, pois seria necessário especificar também como as tuplas e as restrições são representadas (*i.e.*, sintaxe).

O *Egospace* possui várias semelhanças com o *LIME/LighTS*, mas é mais voltado para a sensibilidade ao contexto do que para a coordenação. Dessa forma, a solução proposta também apresenta semelhanças com o *Egospace*, como uma infraestrutura de coordenação e de contexto desacopladas. Porém, ela fornece uma infraestruturas de serviços e é interoperável.

3.6 TOTA

O *TOTA* (MAMEI; ZAMBONELLI, 2009) é uma solução, representada por um *middleware* e um modelo de programação, para o desenvolvimento de sistemas ubíquos focando em mobilidade e sensibilidade ao contexto. Ele oferece suporte à coordenação desacoplada e a atividades adaptativas e sensíveis ao contexto nos cenários desses tipos de sistemas. Assim como o *LIME* e o *Egospaces*, ele é baseado em espaços de tuplas e no modelo de eventos.

No *TOTA*, as interações entre os agentes são realizadas através do *middleware*, por meio de trocas de mensagens que contêm tuplas, de forma totalmente desacoplada. Contudo, não há o conceito de espaços de tuplas centralizado. As tuplas são injetadas no sistema a partir de um agente e depois de devidamente clonada, ela é propagada e difundida por todo o sistema de acordo com um padrão de propagação definido. O *middleware* cuida dessa propagação e da adaptação de seus valores em relação as mudanças dinâmicas que podem ocorrer no sistema.

O *TOTA* é apresentado utilizando um sistema de guia de visitação de museus. Nesse sistema, as principais funcionalidades são: (1) guiar os visitantes no museu encontrando obras de artes de interesses deles; e (2) coordenar a visita de grupos. Os autores enfatizam que uma arquitetura totalmente distribuída pode ser mais eficiente, robusta e mais barata, entretanto, não mostram o porquê. O sistema é formado por PDAs (*Personal Digital Assistant*) dos usuários e por dispositivos computacionais localizados nas salas, corredores e nas próprias obras de arte, todos com capacidade de comunicação sem fio e conectados

por uma rede totalmente *ad-hoc*.

No *TOTA*, as tuplas são definidas em nível de aplicação e são caracterizadas por um conteúdo `C`, uma regra de propagação `P` e uma regra de manutenção `M`:

```
T = (C, P, M).
```

As informações em `C` representam um conjunto ordenado de elementos tipados representando as informações na tupla. A regra de propagação `P` determina como a tupla deve ser distribuída e propagada através de todo o sistema. Essa propagação normalmente consiste na clonagem da tupla, sendo armazenada no espaço de tuplas local, e na transmissão desse clone para os nós mais próximos de forma recursiva. Essa regra de propagação define tanto o escopo, a distância e a direção da propagação como transformações que a tupla pode sofrer em cada transmissão de um agente para outro. Por fim, a regra de manutenção `M` determina como a tupla deve reagir a eventos que ocorrem no ambiente.

Como exemplo, a Figura 3.6 apresenta um cenário do uso do *TOTA*. As tuplas são representadas pelos quadrados cujos dados (`C`) que elas contêm são somente um número inteiro que representa a distância em saltos entre a origem e o agente que detém a tupla. Quando a tupla é injetada no sistema, o valor atribuído é 0, depois disso, a cada salto, a regra de propagação `P` incrementa o valor em uma unidade. A regra `M` se responsabiliza por manter os valores consistentes mesmo quando a topologia da rede é alterada, como é mostrado na transição do cenário (a) para a o (b).

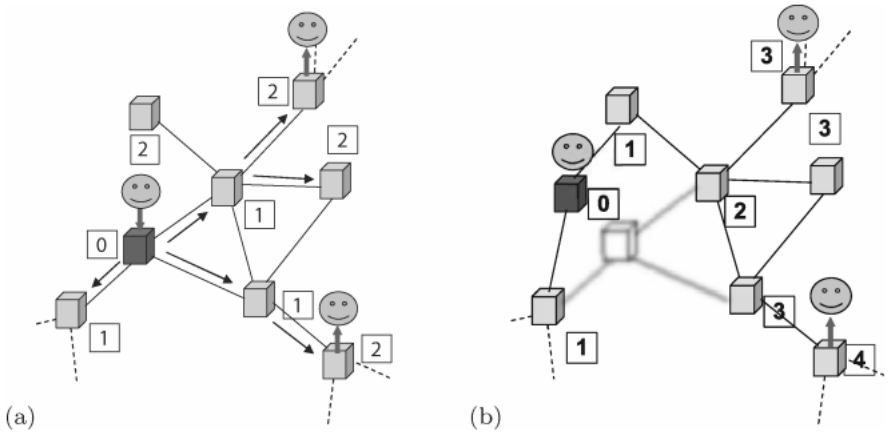

Figura 3.6: Um cenário do *TOTA*. Retirado de (MAMEI; ZAMBONELLI, 2009).

Utilizando o exemplo do museu, se um usuário deseja visitar a sala onde se encontra uma determinada obra de arte, ele injeta uma tupla no sistema que será propagada em direção a todas as obras de arte do museu, usando os dispositivos presentes no sistema que

estejam ao alcance de forma recursiva. Para cada obra, o agente responsável verificará se as informações na tupla de pesquisa se relacionam a ela e responderá com uma nova tupla sobre sua localização e o caminho até sua posição usando a rota pela tupla de pesquisa. Para esse cenário, os autores consideram várias suposições para seu funcionamento adequado.

O *TOTA* permite a realização de uma comunicação desacoplada e assíncrona, além de representar uma infraestrutura de contexto apropriada. Sua maior vantagem é a possibilidade de uso em redes *ad-hoc* e a sua programabilidade. Entretanto, para um desacoplamento temporal, é necessário programá-lo usando regras de manutenção apropriadas. Além disso, usando como base o exemplo do museu, percebe-se que várias trocas de mensagens são necessárias para uma simples funcionalidade. Vários dispositivos têm que receber e retransmitir mensagens em situações que não os envolvem diretamente (o que gasta considerável bateria dos dispositivos móveis). Usando uma abordagem centralizada, uma única troca de mensagens é necessária entre o solicitante e o serviço.

Apesar de ser programável, o *TOTA* são oferece todas as funcionalidades para o requisito de computação sensível ao contexto, como formas mais expressivas de pesquisa, agregação de tuplas e definição de visões. Além disso, os requisitos de descrição e descoberta de serviços não são citados, mesmo que seja possível utilizar a ideia de tuplas para isso ou usar outra solução em conjunto. Por fim, a sua definição e sua implementação não são interoperáveis, porque as regras de propagação e manutenção das tuplas são definidas usando a linguagem de programação e a plataforma que o *middleware* foi desenvolvido, no caso *Java*, na plataforma *JME CDC*. Mesmo que existam implementações para *iOS* e *Android*, não seria possível realizar uma troca de tuplas entre um agente em um *iPhone* e outro em um *smartphone Android*, por exemplo.

Apesar da solução proposta e o *TOTA* compartilharem o uso do conceito de tuplas, na solução é usado um espaço de tuplas centralizado. Porém, seu uso é realizado de forma interoperável em relação as plataformas de execução, além de possuir meios mais flexíveis nos seus métodos de pesquisa por tuplas.

3.7 Conclusão

Apesar dos trabalhos apresentados neste capítulo serem bastante relevantes e solucionarem os problemas alvos de cada um, nenhum deles se adéqua completamente aos requisitos para o desenvolvimento de sistemas ubíquos apresentados no capítulo 2. A

tabela 3.1 sumariza a relação entre esses trabalhos e os requisitos. Dessa forma, novas soluções de software se fazem necessárias para atendê-los apropriadamente.

Tabela 3.1: Relação entre os trabalhos abordados neste capítulo e os requisitos discutidos no capítulo 2.

Requisitos	Soluções					
	AURA	*Gaia*	*one.world*	*LIME/LighTS*	*Egospace*	*TOTA*
Coordenação	◧	◧	◧	■	■	■
D./D. de serviços	◧	◧	◧	◧	□	□
Interoperabilidade	□	◧	□	□	□	□
Sensibilidade ao Contexto	◧	◧	◧	■	■	■
Invisibilidade	■	■	□	□	□	□
Autonomicidade	◧	◧	□	□	□	□

□ Não possui suporte ◧ Possui suporte parcial ■ Possui suporte

Esta dissertação propõe uma nova solução chamada *SysSU* que é apresentado nos capítulos. 4 e 5. Na descrição das soluções deste capítulo foi destacado a diferença do *SysSU* com cada uma delas, mas, em resumo, o *SysSU* provê suporte, especialmente, para os requisitos de coordenação, descoberta/invocação de serviços e interoperabilidade considerando as características especias de cada um no contexto da Computação Ubíqua. Além disso, é possível estendê-lo ou utilizá-lo para atender ao requisito de sensibilidade ao contexto e usá-lo para implementar mecanismos de invisibilidade e autonomicidade.

4 O Sistema de Suporte SysSU

O objetivo deste trabalho é fornecer uma nova infraestrutura de software, na forma de um sistema de suporte, destinada a atender aos requisitos de coordenação, descrição/-descoberta de serviços e interoperabilidade discutidos nos capítulos anteriores. Ela tem como base um modelo misto do modelo *Linda* e do modelo *publish/subscribe*, mas permite uma maior flexibilidade e expressividade no mecanismo de busca associativo no espaço de tuplas e na subscrição de eventos, além de permitir agregações de campos, tuplas e eventos. Além disso, esse mesmo mecanismo de interação é usado para a descrição/descoberta de serviços, pois, devido ao desacoplamento e a expressividade fornecida, é possível solucionar os problemas desse requisito no contexto da Computação Ubíqua.

Contudo, a maior vantagem na utilização do *SysSU* é a sua interoperabilidade. O *SysSU* é apresentado através de uma especificação da sintaxe das mensagens possíveis de serem usadas na interação entre agentes, bem como a semântica de suas operações (*i.e.*, uma linguagem de definição de interface). Essa especificação é independente de linguagem de programação ou plataforma de desenvolvimento e, assim, sendo possível implementá-lo em diferentes plataformas, desde que obedeçam a especificação imposta.

Este capítulo aborda a arquitetura de referência e a especificação formal do *SysSU*, nas seções 4.1 e 4.2 respectivamenente. Já na seção 4.3 é mostrado o conceito de agregadores e na seção 4.4 a conclusão do capítulo.

4.1 Arquitetura de Referência

A arquitetura de referência de um sistema ubíquo está ilustrada na Figura 4.1. Um sistema ubíquo é formado por vários dispositivos, que, por sua vez, pode conter vários agentes (na figura é mostrado somente um por dispositivo). Nessa representação, a plataforma na qual os agentes foram desenvolvidos e a que estão sendo executados não é importante.

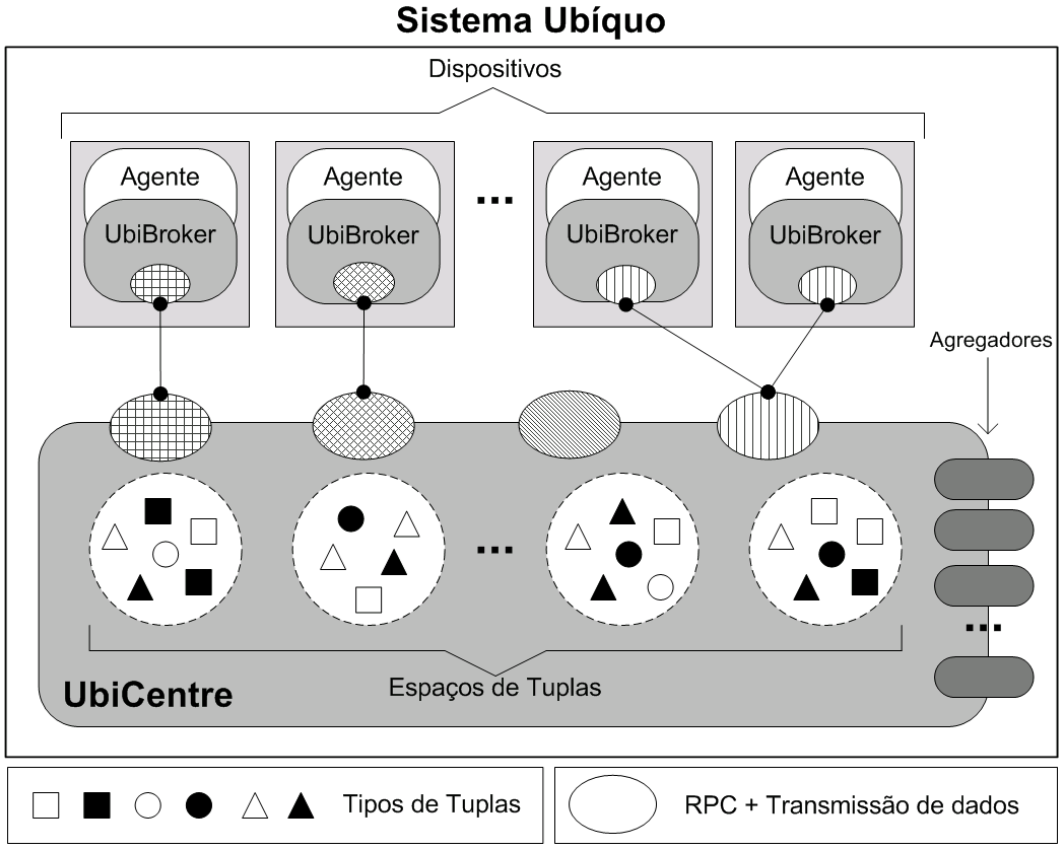

Figura 4.1: Arquitetura do *SysSU*.

O *UbiBroker* é um *middleware* responsável pela comunicação entre os agentes e o *UbiCentre*. Ele é o responsável por fornecer a visão uniforme da arquitetura garantindo, assim, a interoperabilidade. Sua implementação pode ser realizada por meio de um *driver*, acessado pelos agentes, embutido no sistema operacional dos dispositivos ou através de um componente reutilizado na implementação dos agentes que possui meios de comunicação com o *UbiCentre*. Nessas implementações, as características importantes são: estar de acordo com a especificação estabelecida (detalhada adiante), a presença de uma *API* que esteja disponível para uso pelos desenvolvedores dos agentes e que a forma como será realizada a comunicação seja transparante.

Já o *UbiCentre* é um repositório de espaços de tuplas acessado concorrentemente por vários agentes através do seu respectivo *UbiBroker*. Cada agente pode acessar um ou mais espaços de tuplas, os quais são diferenciados por um nome (identificador único). O uso de espaços de tuplas desacopla os agentes no tempo e no espaço, pois a comunicação é realizada por meio do espaço compartilhado de tuplas. Assim, um agente pode inserir uma tupla e sair do sistema enquanto outro agente pode entrar no sistema e retirar essa tupla sem saber qual foi o agente que a inseriu e nem quando isso aconteceu. Além disso,

os espaços de tuplas podem servir como repositórios de informações contextuais na forma de tuplas, os quais permitem que os agentes obtenham informações sobre "o que está acontecendo" no ambiente. Na seção 4.2 são descritas as operações relacionadas com a utilização os espaços de tuplas por parte dos agentes.

O *UbiCentre* pode ser implementado como um serviço ou como uma simples aplicação que aceita conexões por meio de alguma tecnologia de comunicação. Ele pode ser implantado em um único computador ou em vários, o importante é seguir as especificações das operações e das mensagens e estar acessível para os *UbiBrokers* na rede de dados utilizada.

Para cada plataforma de execução que compõe o sistema ubíquo, deve existir uma implementação apropriada do *UbiBroker* capaz de se comunicar com o *UbiCentre*. Essa comunicação é realizada pelo uso de uma tecnologia de *RPC* que permita *callback* (MüHL; FIEGE; PIETZUCH, 2006), sendo as interoperáveis (*e.g.*, *CORBA* ou *web services*) as mais indicadas. Além disso, a implementação é independente da tecnologia de transmissão de dados (*e.g.*, 802.11, *bluetooth*). Essa característica é representada pelas elipses hachuradas na figura. Já o *UbiCentre* pode ser implementado usando apenas uma plataforma de desenvolvimento, pois somente é necessário um *UbiCentre* em um sistema ubíquo. Entretanto, ele deve ser implementado tendo como base todas as tecnologias de comunicação (dados e *RPC*) utilizada pelos *UbiBrokers*. Por esse motivo, para uma maior abrangência, a tecnologia de comunicação deve, preferencialmente, seguir um padrão aberto e interoperável.

Essa independência em relação à forma como o *UbiBroker* se comunica com o *UbiCentre* favorece um maior grau de heterogeneidade no sistema ubíquo. O fato de serem utilizados mecanismos de execução remota de métodos não invalida o propósito de se atingir o desacoplamento desejado. Eles são utilizados somente como veículos de comunicação para a realização das operações disponibilizadas pelo *UbiCentre*. Essas operações, por sua vez, são limitadas e fixas e somente os *UbiBrokers* podem executá-las.

O *UbiCentre* disponibiliza um conjunto de operações que podem ser executadas sobre os espaços de tuplas por ele gerenciadas. Essas operações seguem uma padronização de comportamento, detalhada na subseção 4.2.1, e são executadas a partir dos *UbiBrokers* pelo envio de solicitações em alguma tecnologia de execução remota de métodos. Uma implementação do *UbiBroker* é considerada apropriada quando ele segue a *IDL* estabelecida, a qual é descrita na subseção 4.2.2 e exemplificada no capítulo 5. Essa padronização determina o conjunto de primitivas (relacionadas às operações) que são usadas pelos agentes nas suas interações, bem como os parâmetros e os valores de retorno. Assim, agentes de

software, construídos e executados em plataformas diferentes, podem se comunicar entre si adequadamente através do *UbiCentre*.

A Figura 4.2 apresenta um exemplo de implementação dessa arquitetura. Nela o *UbiCentre* utiliza duas formas de comunicação: *web service SOAP* e *JSON-RPC* (ambos sobre *TCP/IP*). Duas implementações do *UbiBroker* também são representadas, uma que utiliza *JSON-RPC* e outra que usa *SOAP*. Na figura, o retângulo arrendondado maior representa o dispositivo que contém o *UbiCentre* ou o *UbiBroker* e as setas tracejadas representam a comunicação lógica entre o *UbiBroker* e o *UbiCentre*, sendo que a comunicação real é desempenhada pelas camadas *TCP/IP*, representada pelas setas cheias.

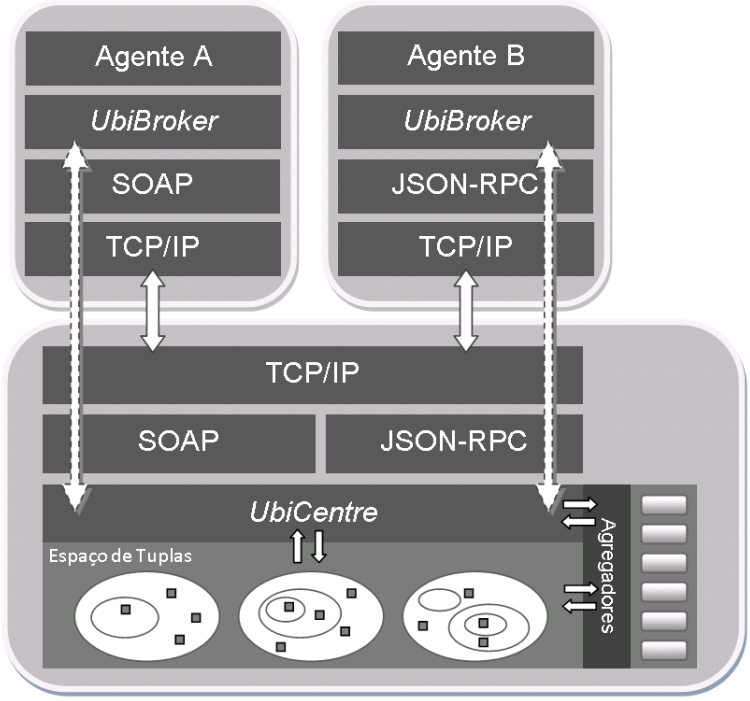

Figura 4.2: Exemplo de implementação da arquitetura do *SysSU*.

Por fim, os agregadores são componentes especiais ligados ao *UbiCentre* com responsabilidades de alterar o comportamento padrão das operações. Esses componentes são desenvolvidos na mesma plataforma do *UbiCentre* seguindo um determinado contrato (interface) e podem ser adicionados e removidos em tempo de execução do *UbiCentre*. Sua funcionalidade é a interceptação de qualquer invocação de operações realizada nos espaços de tuplas no *UbiCentre* alterando o resultado padrão, normalmente retornando tuplas que possuem agregações de campos e/ou de tuplas, pois são capazes de executar operações nos espaços de tuplas e combinar resultados. Eles serão melhor detalhados na seção 4.3.

4.2 Especificação

Os agentes se comunicam com o *UbiCentre* por meio do *UbiBroker* usando um conjunto fixo de primitivas que representam as operações fornecidas pelo *UbiCentre*. Esse conjunto de primitivas foca somente nas questões de descrição/descoberta de serviços e nas interações entres os agentes, as quais não são influenciadas por características de linguagens de programação ou de plataformas de execução.

Na implementação do *UbiCentre* utilizada, essas operações devem ser fornecidas seguindo uma especificação de semântica e sintaxe padronizada para se garantir a interoperabilidade. A especificação da semântica é mostrada usando teoria dos conjuntos e lógica de primeira ordem e possui foco na determinação do comportamento e não na validação formal de propriedades. Já a sintaxe é determinada principalmente pelas tecnologias de *RPC* escolhidas para a implementação do *SysSU*, a qual é detalhada no capítulo 5.

4.2.1 Semântica das Operações

Mecanismo de Associação

A comunicação entre agentes usando um espaço de tuplas é realizado por um conjunto de operações que inserem, leem ou retiram tuplas. A leitura ou retirada de uma tupla presente no espaço é realizada através de um mecanismo de associação. O mecanismo do modelo *Linda* original é simples e limitado (BALZAROTTI; COSTA; PICCO, 2007) e, assim, o modelo de interação deste trabalho o estende com mais funcionalidades e expressividade, a fim de adequá-lo apropriadamente para sistemas ubíquos. Essa associação é denominada *matching* e, para descrevê-la, é necessário, primeiramente, definir alguns conceitos importantes, como campos, tuplas, padrões de tuplas e espaços de tuplas.

Definição 1 [Campo]. Um campo c é um elemento no formato de par nome/valor (n, v) com nome n e valor v. Além disso, existe um função `typeof`(v) que mapeia de forma única o valor do campo v para um tipo t que corresponde ao tipo de dado do campo. Dessa forma, um campo é caracterizado por três atributos: nome, valor e tipo.

Como uma das prerrogativas deste trabalho é a interoperabilidade, os tipos permitidos devem ser fixos e bem determinados e estabelecem o conjunto de tipos permitidos $\boldsymbol{T}$. São eles:

- `string`: um conjunto de caracteres delimitado por aspas duplas;

- `integer`: um número inteiro;
- `float`: um número real (presença de casas decimais);
- `boolean`: um valor lógico: verdadeiro (`true`) ou falso (`false`);
- `array`: representa um conjunto de valores separados por vírgula; e
- `object`: representa uma tupla aninhada.

Exemplos de campos podem ser: *(user, "Fabrício Lima")*, *(latitude, 25.7)*, *(service, "printer")*, *(hasError, `true`)* e *(values, [12, 54.9, 789.569])*.

Essa definição de campo difere da utilizada no modelo *Linda* pela adição do nome do campo. No modelo *Linda* somente o valor e o tipo são possíveis. A adição do nome favorece uma maior expressividade nos dados trocados entre agentes e a implementação de mecanismos associativos mais flexíveis. Esse modelo de representação já é usado por outras soluções como o *LIME/LighTS*, o *Egospace* e o *TOTA*.

Definição 2 [Tupla]. Uma tupla é um conjunto finito $t = \{(n_1, v_1), (n_2, v_2), ..., (n_n, v_n)\}$, onde cada (n_i, v_i) representa um campo tal que, $\forall (n_j, v_j), (n_k, v_k) \in t \ (n_j \neq n_k)$.

Assim, uma tupla é formada por um conjunto de campos que diferem pelo nome, por exemplo: {*(service, "printer"), (color, `false`), (file, "teste.pdf")*} e {*(user, "Fabrício"), (lat, 28.9), (long, 32.7)*}.

Definição 3 [Espaço de tuplas]. Um espaço de tuplas é um conjunto finito e não ordenado $T = \{t_1, t_2, ..., t_n\}$, onde cada t_i representa um tupla.

Definição 4 [Padrão de tupla] Um padrão de tupla é um conjunto finito $t = \{(n_1, v_1), (n_2, v_2), ..., (n_n, v_n)\}$, onde cada (n_i, v_i) representa um campo que pode ser anônimo ($n_i =$ `null`), possuir valor indefinido ($v_i =$ `null`) ou um ter como valor um dos elementos do conjunto T.

São exemplos de padrões de tupla: {*(`null`, `string`), (`null`, `array`)*}, {*(service, `null`), (description, `null`)*}, {(service, "printer"), *(color, `boolean`), (file, "teste.pdf")*} e {*(user, `string`), (lat, `float`), (long, 32.7)*}.

Da mesma forma que na definição de tupla, a definição de padrão de tupla estende à definição original com a adição de nomes de campos e a possibilidade de não definir nenhum valor tanto para nome como para valor. Com as definições de tupla e padrão de tupla, podemos definir, então, o mecanismo associativo que determina se uma tupla se relaciona com um padrão de tupla.

Definição 5 [Associação de um padrão com uma tupla]. Seja t um tupla e p um padrão de tupla, dizemos que t se associa a p, denotado por $\texttt{assoc}(t,p)$, se $\forall\ (n_p, v_p) \in p\ \exists\ (n_t, v_t) \in t$ tal que $(n_p = n_t \vee n_p = \texttt{null}) \wedge (v_p = v_t \vee v_p = \texttt{null} \vee (v_p \in T \wedge v_p = \texttt{typeof}(v_t)))$.

Esse mecanismo de associação é similar ao do *LIME/LighTS*, *Egospace* e do *TOTA* e fornece maior flexibilidade em relação ao mecanismo original. A principal diferença é a independência do tamanho das tuplas e da posição dos campos. Contudo, sua maior vantagem é facilitar a busca por tuplas em situações que o agente somente se interessa ou somente tem conhecimento de alguns campos e permitir a adição de novos campos nas tuplas sem afetar os agentes que já as usavam antes. Além disso, a forma como as tuplas são pesquisadas através de padrões, e não por pesquisa direta por determinada tupla, facilita o desenvolvimento de agentes que não possuem um conhecimento completo do ambiente computacional, que é normalmente extenso e dinâmico. Assim, os agentes podem usar esse mecanismo para lidar com incertezas, dinamicidade e heterogeneidade.

Porém, métodos mais expressivos de pesquisar por tuplas devem ser fornecidos para o desenvolvimento de sistemas ubíquos. Exemplos de expressividade são: dado um campo numérico, verificar se o valor está em um determinado intervalo; no caso de um campo do tipo ***`string`***, verificar se o valor começa com determinado caractere. Contudo, uma maior expressividade é obtida se for possível realizar relacionamentos entre campos. Por exemplo, para tuplas associadas ao padrão {*(x,* ***`float`****), (y,* ***`float`****)*} , verificar se $x > 50 \wedge (x+y) = 0$. Para atingir essa finalidade, o *SysSU* dispõe de filtros que são usados juntos com os padrões de tupla no mecanismo de associação.

Definição 6 [Filtro]. Um filtro é uma função lógica sem estado f aplicada a uma tupla t, *i.e.*, $f(t) \rightarrow \{\texttt{true}, \texttt{false}\}$. Um tupla t passa pelo filtro f, se $f(t) = \texttt{true}$.

Um filtro é composto por uma expressão lógica formada por predicados conectados por operadores lógicos (*e.g.*, $\wedge$, $\vee$, $\neg$), onde esses predicados são critérios para os campos da tupla. Além disso, os filtros podem realizar operações de inserção e consultas em espaços de tuplas (apresentadas da subseção 4.2.1). Um exemplo de filtro usando pseudocódigo pode ser:

```
proc filter(t) ≡
  pattern := Pattern({(user,"fabricio"),(location,string)});
  user = tuplespace.readone(pattern);
  return user <> null ∧ t[location] = user[location].
```

Esse exemplo apresenta um filtro utilizado na consulta de tuplas que representam impressoras (previamente inseridas na inicialização do sistema ou quando uma nova impressora é instalada). Impressoras essas indicadas com sua respectva localização. Ele mostra a sensibilidade ao contexto oferecida por esse mecanismo. O filtro determina que somente as tuplas que representam impressoras com o campo `location` igual ao do usuário "fabricio" (portador do dispositivo) sejam retornadas. Para isso, ele executa o método `readone` do objeto `tuplespace`.

A expressividade desses critérios dependerá da implementação realizada. Por exemplo, na implementação de referência, descrita no capítulo 5 os filtros são definidos usando a linguagem *Javascript* que permite realizar vários tipos de operações e comparações sobre os campos da tupla. O capítulo 5 apresentará exemplos concretos de filtros, com e sem sensibilidade ao contexto.

Essa adição ao modelo associativo já é utilizada no *Egospace*, porém só pode ser realizado um campo por vez, não possibilitando verificar relacionamentos entre campos. No *LIME/LighTS* e no *TOTA* não existe essa limitação, mas os filtros são pré-determinados e fixos na implementação do mecanismo. No modelo proposto, eles são dinâmicos e fazem parte dos parâmetros da operação realizada pelo *UbiBroker* e o *UbiCentre* deve ser capaz de verificar qualquer filtro devidamente definido.

As consultas realizadas no espaço de tuplas são realizadas tendo como argumentos um padrão de tupla e um filtro. Esses dois elementos não são usados de forma isolada em nenhuma operação no espaço de tuplas, assim eles são reunidos no conceito de *query*.

Definição 7 [***Query***]. Uma *query* q é um par ordenado (p, f) onde p é um padrão de tupla e f um filtro.

Seja T um espaço de tuplas e q uma *query*.

Definição 8 [***Matching***]. Definimos a operação de *matching*, denotada como $\texttt{matches}(T, q)$, como o subconjunto $T' \subset T$ tal que $T' = \{t \mid t \in T \wedge (\texttt{assoc}(t, q.p) \wedge q.f(t))\}$[1].

Assim, a operação `matches` retorna um conjunto de tuplas presente no espaço de tuplas ou um conjunto vazio. As tuplas que farão parte do resultado são as que se associam com o padrão de tupla usado como argumento e que passarem pelo filtro especificado.

[1] O operador ponto (.) é usado para acesso ao itens do par ordenado

Operações

O primeiro conjunto de operações é apresentado na Figura 4.3. Elas são realizadas sobre um espaço de tuplas T e as principais são `put`[2] (1), `read` (2) e `take` (3). A primeira, similar a `out` do modelo *Linda*, insere uma tupla no espaço de tuplas (*i.e*, adicionado mais um elemento ao conjunto T), enquanto a segunda recupera um conjunto das cópias de todas as tuplas relacionadas com um determinado padrão e a terceira retira do espaço de tuplas um conjunto de tuplas que satisfazem um determinado padrão (*i.e.*, realiza uma subtração dos elementos de T). Caso nenhuma tupla satisfaça o padrão, um conjunto vazio é retornado como resposta à operação `read` ou `take`.

$$\texttt{put}(t) \Rightarrow \langle \texttt{pub}(t);\ T := T \cup \{t\} \rangle \quad (1)$$
$$T' := \texttt{read}(q) \Rightarrow T' := \texttt{matches}(T, q) \quad (2)$$
$$T' := \texttt{take}(q) \Rightarrow \langle T' := \texttt{read}(q);\ T := T - T' \rangle \quad (3)$$
$$t := \texttt{readone}(q) \Rightarrow t := \texttt{random}(\texttt{read}(q)) \quad (4)$$
$$t := \texttt{takeone}(q) \Rightarrow \langle t := \texttt{readone}(q);\ T := T - \{t\} \rangle \quad (5)$$
$$T' := \overline{\texttt{read}}(q) \Rightarrow \langle \texttt{await read}(q) \neq \emptyset \rightarrow T' := \texttt{read}(q) \rangle \quad (6)$$
$$T' := \overline{\texttt{take}}(q) \Rightarrow \langle T' := \overline{\texttt{read}}(q);\ T := T - T' \rangle \quad (7)$$
$$t := \overline{\texttt{readone}}(q) \Rightarrow t := \texttt{random}(\overline{\texttt{read}}(q)) \quad (8)$$
$$t := \overline{\texttt{takeone}}(q) \Rightarrow \langle t := \overline{\texttt{readone}}(q);\ T := T - \{t\} \rangle \quad (9)$$

Figura 4.3: Operações básicas.

No caso de `read` e `take`, variações de cada operação são fornecidas. Para os casos em que só é necessário ler ou remover uma tupla, podem ser usadas `readone` (4) ou `takeone` (5), respectivamente. Na semântica dessas operações, a operação `random` significa que um elemento é escolhido aleatoriamente do conjunto de tuplas resultantes da execução da operação passada como parâmetro.

As outras variantes são $\overline{\texttt{read}}$ (6), $\overline{\texttt{take}}$ (7), $\overline{\texttt{readone}}$ (8) e $\overline{\texttt{takeone}}$ (9). Elas são similares às suas correspondentes de mesmo nome, porém com a diferença que elas executam de forma síncrona, o que ocasiona o bloqueio do processo que as executa enquanto não houver tuplas no espaço que satisfaçam a *query* especificada. Na semântica dessas operações, a construção $\langle \texttt{await}\ A \rightarrow B \rangle$ indica o bloqueio do processo corrente até que a condição A seja satisfeita (ANDREWS, 1991). Se A se torna verdadeiro, então B é executado sequencialmente. Os parênteses angulados, também presentes em outras operações, indicam que as instruções são executadas de forma atômica, isto é, enquanto B é

[2] A instrução `pub`(t) em `put` é descrita na próxima subseção.

executado A continua verdadeiro e nenhum estado interno de B é visível externamente.

Mecanismo de Eventos

O mecanismo de eventos, assim como o *LIME*, tem como base o modelo de tuplas. Mais precisamente, ele usa o conceito de tuplas como modelo de dados e o de padrão de tuplas como modelo de filtros. Esse tipo de modelo de eventos é comumente chamado de *modelo baseado em conteúdo com padrões* (CARZANIGA; ROSENBLUM; WOLF, 2000). Ele se baseia em registros contendo vários atributos que possuem identificação, na forma de nomes, para representar os eventos (*i.e.*, pares nome/valor). As subscrições são realizadas através de filtros que representam predicados sobre os valores dos registros. Contudo, é possível realizar subscrições e notificações sensíveis ao contexto (EUGSTER; GARBINATO; HOLZER, 2005), pois as tuplas podem ser usadas para representar informações contextuais e elas estão acessíveis aos filtros de notificações.

Um evento é representado por uma tupla que deve conter todas as informações necessárias sobre ele em seus campos. Dessa forma, o mesmo modelo de dados utilizado na comunicação proativa é usada também na comunicação reativa, reduzindo a complexidade. Nesse mecanismo de eventos é usado o conceito de *reações* (MURPHY; PICCO; ROMAN, 2006), que representam o conjunto de ações a serem realizadas pelo agente quando a notificação for realizada. Nas subscrições, o agente utiliza um identificador para a reação, de forma que, ao receber a notificação, ele consiga diferenciar qual das reações deverá ser executada.

Definição 9 [Subscrição]. Uma subscrição é um par ordenado (r,q) onde r representa a identificação de uma reação que é executada quando uma tupla associada à *query* q for inserida no espaço de tuplas T.

Definição 10 [Espaço de subscrições]. Conjunto R formado por elementos do tipo subscrição (r,q),presente em cada espaço de tuplas T.

As operações relacionadas a eventos estão representadas na Figura 4.4. A operação `sub` realiza uma nova subscrição, desde que não exista uma subscrição com a mesma reação. Assim, para *queries* diferentes, novas subscrições precisam ser feitas, mas uma mesma *query* pode estar relacionada a várias reações. A operação `unsub` desinscreve uma notificação usando somente a identificação da reação r para isso.

A publicação de um novo evento é realizada através da operação `pub` (12). Ela recebe como argumento uma tupla t que representa o evento. As notificações são realizadas

$$\texttt{sub}(r,q) \Rightarrow \langle R := R \cup \{(r,q)\} : \forall (r_i,q_i) \in R \; r_i \neq r \rangle \quad (10)$$

$$\texttt{unsub}(r) \Rightarrow \langle R := R - \{(r,q)\} : (\exists (r_i,q) \in R : r_i = r) \rangle \quad (11)$$

$$\texttt{pub}(t) \Rightarrow \langle \texttt{notify}(r,t) : t \in \texttt{matches}(\{t\},q) \wedge (r,q) \in R \rangle \quad (12)$$

$$T' = \texttt{verify}(r) \Rightarrow \langle T' = \{read(q) : (r,q) \in R\} \rangle \quad (13)$$

Figura 4.4: Operações relacionadas a eventos.

antes que a tupla seja inserida no espaço de tuplas (vide operação (1) *put*). Essa operação notifica através da primitiva `notify` os agentes relacionados às reações presentes nas subscrições que possuam *queries* que se associam a tupla t. A primitiva `notify`, assim como `await`, é não determinística, assim, não existe uma ordem específica nas notificações, mas todos os interessados devem ser notificados. Além disso, através da operação `verify` (13) é possível garantir o desacoplamento temporal em relação a eventos. Quando o agente o executa, ele recebe as notificações que deveria receber se estivesse ativo no sistema quando eles ocorreram. Isso é alcançado devido ao fato que os eventos são representados por tuplas no espaço de tuplas e essa operação realiza uma consulta das tuplas relacionadas com a reação utilizando como argumento.

Esse modelo usa a mesma expressividade das tuplas, dos padrões de tuplas e, principalmente, dos filtros relacionados a tuplas. Pelo fato de serem utilizados os mesmos filtros das consultas no modelo proativo, as notificações podem ser sensíveis ao contexto. Assim, um agente pode especificar que só será notificado da inserção de determinada tupla se determinada informação contextual estiver disponível e com determinado valor. Por exemplo, o pseudocódigo na Figura 4.5 exemplifica um filtro que notifica um agente sempre que um valor de temperatura for inserido por um sensor e esse valor for o menor de todos.

```
proc filter(t) ≡
  query := Query({(sensor_id, integer), (temperature, float)}, null);
  tuples[] = tuplespace.read(query);
  for i := 1 to tuples.lenght step 1 do
      if tuples[i][temperature] < t[temperature] then return false; fi
  od
  return true.
```

Figura 4.5: Exemplo em pseudocódigo da lógica de um filtro.

Mecanismo de Serviços

O mecanismo de serviços, assim como o de eventos, utiliza o modelo *Linda* como base, o que diminui a complexidade de implementação do *UbiCentre*, bem como dos *UbiBrokers*, pois muito é reutilizado. Nesse mecanismo, a descrição de serviços é realizada através do conceito de tuplas e a descoberta de serviços é realizada pelo mecanismo de *matching*. Assim, os serviços são descritos através de atributos que determinam suas características e funcionalidades como alguns modelos já propostos (ZHU; MUTKA; NI, 2005; EDWARDS, 2006). Contudo, o *SysSU* se destaca pelas funcionalidades oferecidas na descoberta dos serviços através da pesquisa associativa com o uso de padrões de tuplas e filtros. Além disso, o serviços descobertos podem ser invocados de forma desacoplada utilizando o mesmo mecanismo.

Os serviços são tratados no *SysSU* usando a abstração de caixas pretas (somente sua interface é conhecida). Eles são entidades de software que representam procedimentos que recebem como argumento uma tupla qualquer, realizam alguma ação e retornam um resultado, ou não, na forma de um conjunto de tuplas. A localização, o dispositivo (ou conjunto de dispositivos) e a ação desempenhada pelos serviços não é importante para esse mecanismo.

Definição 11 [Serviço]. Um serviço s é uma relação que associa uma tupla a um conjunto de tuplas, o qual é invocado através da primitiva $\texttt{exec}(s,t)$ que recebe como argumento o identificador do serviço (assim como as reações, os serviços possuem uma identificação) e a tupla que será o argumento dele.

Definição 12 [Espaço de serviços]. Para cada espaço de tuplas T existe um conjunto S, chamado de espaço de serviços, formado por pares ordenados (s,d) onde s representa um serviço e d uma tupla que descreve o serviço.

As operações `reg` (14) e `unreg` (15), representadas na figura 4.6, são responsáveis por registrar e desregistrar serviços no espaço de serviços.

$$\texttt{reg}(s,d) \Rightarrow \langle S := S \cup \{(s,d)\} : \forall (s_i,d_i) \in S \; s_i \neq s \rangle \quad (14)$$

$$\texttt{unreg}(s) \Rightarrow \langle S := S - \{(s,d)\} : (\exists (s_i,d) \in S : s_i = s) \rangle \quad (15)$$

$$S' := \texttt{discovery}(q) \Rightarrow \langle S' = \{s : d \in \texttt{matches}(\{d\},q) \wedge (s,d) \in S\} \rangle \quad (16)$$

$$T' := \texttt{invoke}(s,t) \Rightarrow \langle T' = \texttt{exec}(s,t) : (\exists (s_i,d) \in S : s_i = s) \rangle \quad (17)$$

Figura 4.6: Operações relacionadas a serviços do *SysSU*.

Nesse modelo, um serviço pode ser visto como um agente esperando uma notificação

de evento. A primitiva `exec` é similar a `notify`, o parâmetro *s* similar a *r* e as operações `reg` e `unreg` a `sub` e `unsub`, respectivamente. Assim, o serviço fica esperando por alguma requisição de invocação que é realizada através da operação `invoke` (17).

Depois de devidamente descoberto através da operação `discovery` (16) (essa operação pode retornar mais de um serviço cabendo a implementação do agente cliente escolher qual o ideal), o agente cliente pode invocar o serviço através da operação `invoke` usando como um argumento um tupla (o formato dessa tupla é determinado pela descrição do serviço). Depois disso, o agente fica bloqueado esperando a resposta do serviço, que pode ser um conjunto de tuplas ou um conjunto vazio. Isso ocorre de forma totalmente desacoplada, em que o cliente do serviço não possui conhecimento do servidor responsável pelo serviço e esse não possui conhecimento de seus clientes.

4.2.2 Sintaxe das Mensagens

Após a definição do comportamento das operações oferecidas pelo *UbiCentre*, é necessário agora especificar como elas são invocadas pelos agentes. A sintaxe das operações define a forma das mensagens que o *UbiCentre* é capaz de reconhecer para a invocação de suas operações. Nessa sintaxe estão definidos o identificador das operações, os parâmetros e valores de retorno ou exceções possíveis. Assim, a sintaxe define a interface usada pelos agentes para se comunicar com o *UbiCentre* através do *UbiBroker*. Mais precisamente, o *UbiBroker* fornece o conjunto de operações na forma de *API* para serem usadas pelos agentes. A invocação dessas operações é realizada remotamente e de forma transparente pelo *UbiBroker* através da tecnologia de *RPC* utilizada nas implementações. Assim, a forma das mensagens depende das tecnologias de *RPC* utilizadas.

Dessa forma, para cada tecnologia de *RPC*, é necessário fornecer uma *IDL* apropriada. Por exemplo, se for usado *CORBA*, uma *OMG IDL* é necessária, ou se for usado *web services*, uma descrição *WDSL* tem que ser fornecida (na implementação de referência no capítulo 5 será mostrado um exemplo de *IDL* para *JSON-RPC*). Contudo, existem alguns pontos que devem ser satisfeitos:

1. O nome do espaço de tuplas é obrigatório;

2. As operações possíveis são: `put`, `read`, `readone`, `readsync`, `readonesync`, `take`, `takeone`, `takesync`, `takeonesync`, `sub`, `unsub`, `verify`, `reg`, `unreg`, `discovery` e `invoke`;

3. Os parâmetros são definidos pela operação correspondente de acordo com as definições na subseção 4.2.1;

4. Os valores de retorno são determinados de acordo com as definições na seção 4.2.1;

5. As exceções possíveis são: "*UbiCentre* inacessível", "espaço de tuplas inexistente" e as relacionadas a má formação das mensagens (*e.g*, "tupla mal formada");

6. A representação das tuplas deve fornecer meios de se saber o nome, valor e tipo do valor dos campos;

7. A representação dos padrões de tuplas deve fornecer meios de determinar o valor `null` e relacionados aos tipos para os campos; e

8. Os tipos possíveis são os definidos na subseção 4.2.1.

Por fim, os métodos de *callback* (primitivas `notify` e `exec`) são determinados pela tecnologia utilizada, pois é um método que é diferente de tecnologia para tecnologia. Contudo, a forma dos padrões de tuplas utilizadas nas subscrições e da tupla na notificação são iguais a das operações básicas.

4.3 Agregadores

Como o nome sugere, sua funcionalidade é a realização de agregações, tanto de campos como de tuplas e eventos, com a finalidade de gerar novas tuplas e eventos adequados para a coordenação de determinados agentes no sistema. Eles agem através da interceptação da invocação das operações realizadas em um espaço de tuplas em que esteja associado alterando o resultado padrão retornando um novo resultado ou não, conforme um procedimento determinado. Para tanto, os agregadores definem quais as operações com determinado argumento eles desejam interceptar e o processo que deve ser realizado. Essas duas operações definem a interface que todo agregador precisar implementar para ser usado pelo *UbiCentre*:

1. `check`($op, param$): operação utilizada para verificar se a operação op e a tupla ou padrão de tupla $param$ utilizada na operação no espaço de tuplas é de interesse do agregador. Se sim, o *UbiCentre* executa a operação `aggregate` do agregador com esses mesmos argumentos. Caso seja encontrado mais de um agregador compatível, somente um deles é executado;

2. $T' = \texttt{aggregate}(op, param)$: retorna um conjunto qualquer de tuplas T'.

Por exemplo, em um espaço de tuplas existem várias tuplas com informações de temperatura e umidade fornecidas por sensores espalhados em um ambiente. As tuplas são no formato $\{(sensor\_id, \texttt{integer}), (temperature, \texttt{float}), (humidity, \texttt{float})\}$. Um determinado agente necessita saber a média da temperatura em todo o ambiente coberto pelos sensores. Com o uso de um agregador não é necessário ler todas as tuplas e realizar o cálculo, assim, polpando transmissão de dados. Para realizar esse procedimento, o agregador determina que as operações relacionadas a consultas com o padrão $\{(temperature\_average, \texttt{float})\}$ são de seu interesse e o método $\texttt{aggregate}(q)$ pode ser implementado de acordo com o pseudo código mostrado na Figura 4.7.

```
proc aggregate(op, param) ≡
  query := Query(
    {(sensor_id, integer), (temperature, float), (humidity, float)}, null);
  tuples[] as tuplespace.read(query);
  sum := 0
  for i := 1 to tuples.lenght step 1 do
      sum := sum + tuples[i][temperature];
  od
  return Tuple({(temperature_average, sum/tuples.lenght)}).
```

Figura 4.7: Exemplo em pseudocódigo da lógica de um agregador.

Como exemplo de agregação relacionado a eventos, considere o caso em que um agente deseja ser notificado quando dez pessoas quaisquer passarem juntas no mínimo cinco minutos na sala de reunião. Esse evento pode ser descrito por tuplas no formato $\{(users, \texttt{array}), (room, \text{“}\texttt{sala de reunião}\text{”})\}$ que não são inseridas nos espaços de tuplas presente no *UbiCentre* do sistema ubíquo por nenhum agente. Contudo, existem tuplas no formato $\{(user, \texttt{string}), (room, \texttt{string})\}$ inseridas por um agente que sabe que determinada pessoa está em determinada sala pelo uso de crachás inteligentes. Um agregador pode usar essas tuplas para gerar o evento desejado. A Figura 4.8 apresenta em pseudocódigo esse agregador. No caso, sua operação `check` retorna verdadeiro se a operação for `put` e a tupla usada como argumento se associar ao padrão $\{(user, \texttt{string}), (room, \text{“}\texttt{sala de reunião}\text{”})\}$.

No início do procedimento, é criado uma nova tupla no formato $\{(user, \texttt{string}), (room, \text{“}\texttt{sala de reunião}\text{”}), (time, \texttt{integer})\}$ em que os campos `user` e `room` possuem os valores da tupla usada em `put` e o campo `time` representa o instante em que a tupla foi inserida. Essa nova tupla substituirá a tupla inserida pela execução da operação `put`

```
proc aggregate(op, param) ≡
  tuple := Tuple({(user, param[user]), (room, param[room]), (time, system.time)});
  tuplespace.take(param, null);
  tuplespace.put(tuple);
  query := Query({(user, string), (room, "sala de reunião"), (time, integer)}, filter);
  tuples[] = tuplespace.read(query);
  if tuples.lenght = 10
    then
        users[] = string[];
        for i := 1 to tuples.lenght step 1 do
            users[user] := tuples[i][user];
        od
        tuplespace.put(Tuple({(users, users), (room, "sala de reunião")}))
  fi
  return null.
proc filter(t) ≡
  return (system.time − t[time]) >= Time(00 : 05 : 00).
```

Figura 4.8: Exemplo em pseudocódigo da lógica de um agregador relacionado a eventos.

interceptada. A substituição não afetará os agentes que utilizam a tupla removida, pois os padrões de tuplas ainda se associarão com a nova tupla e os filtros serão executados sem erros, pois os campos da tupla continuam presentes na nova tupla. Após isso, o procedimento verifica se existem dez tuplas no formato da nova tupla utilizando o filtro. Se sim, então um nova tupla {(*users*, `array`), (*room*,"`sala de reunião`")} contendo os nomes de todos os usuários é usada como argumento na execução da operação `pub`, assim, notificando o agente interessado.

Pelo fato de não existir uma comunicação direta entre os *UbiBrokers* e os agregadores, não é necessário nenhuma formalização com o intuito de interoperabilidade. O poder e a expressividade deles são determinados pela linguagem de programação e a plataforma usada na implementação do *UbiCentre*, pois eles estão ligados e usados diretamente por ele. Além disso, por serem determinados e implantados de forma estática, a dinamicidade oferecida pelo *SysSU* é reduzida. O ideal seria que a lógica das operações fosse determinada nas consultas, como o filtro presente em uma *query*. Contudo, essa forma estática facilita a adoção de agregadores mais complexos que necessitam de elevado número de linhas de código e assim de maior processamento. Um exemplo desse tipo de agregador são os que precisam utilizar ontologias (CHEN; FININ; JOSHI, 2003).

4.4 Conclusão

O *SysSU* é um sistema de suporte destinado a fornecer funcionalidades que facilitam a coordenação entre agentes e a descrição/descoberta/invocação de serviços considerando as características dos sistemas ubíquos. Ele foca principalmente em interações desacopladas e interoperáveis baseando-se nos modelos *Linda* e *publish/subscribe* e em uma linguagem de descrição de interfaces (considerando sintaxe e semântica). Pelo exposto, o *SysSU* oferece suporte parcial aos requisitos de coordenação, descrição/serviços e interoperabilidade apresentados no capítulo 2. O parcial se refere ao fato de que mecanismos de agregações ainda são disponibilizados de forma estática, assim, não considerando a abertura e a dinamicidade de tais sistemas.

Este capítulo apresentou o *SysSU*, sua arquitetura, seus principais componentes e seus conceitos utilizando teoria dos conjuntos e lógica de primeira ordem. Essa descrição teve a finalidade de padronizar o comportamento das primitivas de comunicação e as interações entre os componentes da arquitetura, assim, permitindo implementações em variadas plataformas de desenvolvimento e linguagens de programação. Isso favorece sua abrangência e uso, pois não o atrela a tecnologias, plataformas e linguagens específicas. O próximo capítulo apresenta uma implementação de referência e exemplos de códigos de utilização do *SysSU* através de um estudo de caso.

5 *Implementação de Referência e Estudo de Caso*

Neste capítulo, na seção 5.1, é apresentado a implementação de uma versão do *UbiCentre* realizada. Na seção 5.2 é mostrada a implementação de um conjunto de *UbiBrokers* no contexto de duas aplicações: um guia de visitação chamado *GREat Tour* (descrito na seção 5.3) e um sistema de impressão chamado *UbiPrinter* (descrito na seção 5.4). Na descrição das aplicações e dos serviços utilizados por elas é mostrado como os *UbiBrokers* são usados no seu desenvolvimento de acordo com os requisitos discutidos no capítulo 2. Essas aplicações representam um estudo de caso e possui a finalidade de mostrar a validade e a utilidade do *SysSU*.

Essas implementações são ditas de referência porque podem ser utilizadas como base para a implementação em outras plataformas e linguagens, pois é descrito como as especificações descritas no capítulo 4 são transformadas em código executável. Para tanto, este capítulo dá maior ênfase às mensagens necessárias para o *UbiBroker* se comunicar apropriadamente com o *UbiCentre* de forma a facilitar a implementação de novos *UbiBrokers* em plataformas direfentes. Essas implementações estão disponíveis para download e contribuições no endereço web `http://code.google.com/p/syssu/`.

5.1 O *UbiCentre*

Uma única versão do *UbiCentre* foi implementada, cuja plataforma de desenvolvimento foi a *Java SE* versão *6.0*. Essa plataforma foi escolhida devido a sua interoperabilidade em diferentes sistemas operacionais e pelo conhecimento técnico dos desenvolvedores. Já as tecnologias de invocação remota de métodos utilizadas nessa implementação foram *JSON-RPC* e *web services SOAP*, utilizando *TCP/IP* para transmissão de dados. A primeira foi escolhida devido ao fato dela utilizar a linguagem de representação de dados

JSON[1], popular no desenvolvimento de aplicações web, que gera representações mais simples e menores que outros formatos, como o *XML*, assim, reduzindo a transmissão de dados na rede (ERFIANTO; MAHMOOD; RAHMAN, 2007). Além disso, ela se enquadra nas necessidades da implementação, como a diferenciação dos tipos de dados necessária para representar os campos de tuplas. Já a segunda foi escolhida devido a sua popularidade e por existirem variadas implementações em diversas plataformas.

A Figura 5.1 apresenta a arquitetura dessa implementação. Além do *UbiCentre*, estão representados dois agentes com seu respectivo *UbiBroker*, um deles baseado em *SOAP* e outro em *JSON-RPC*. As setas duplas representam as principais interações. A principal delas é entre as camadas *TCP/IP*, onde a transmissão dos dados é efetivamente realizada. Já as setas tracejadas representam a comunicação virtual entre o *UbiBroker* presente no agente e o *UbiCentre*. É através dessa última que o agente invoca as operações fornecidas pelo *UbiCentre* por meio das classes e métodos contidos na *API* fornecida pela implementação do *UbiBroker*.

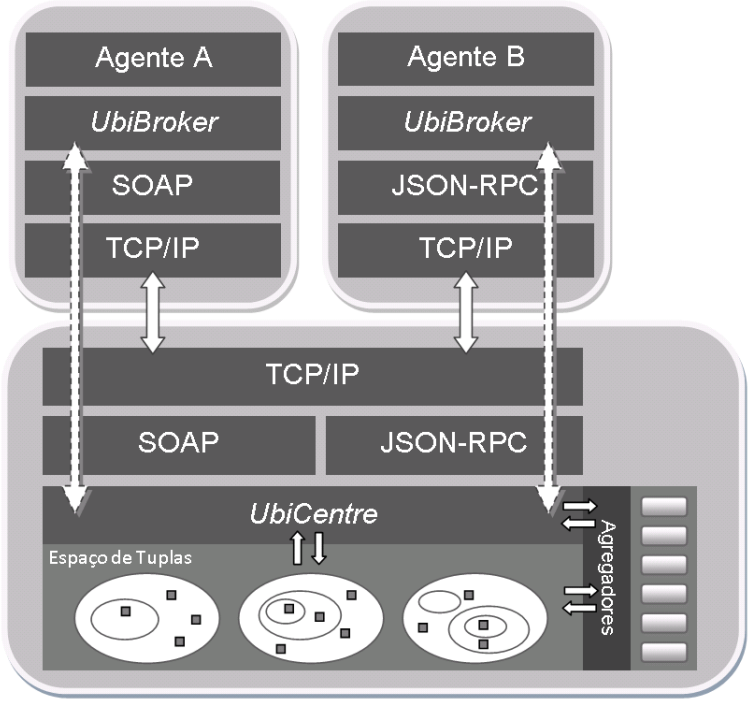

Figura 5.1: Arquitetura da implementação de referência.

5.1.1 Espaços de Tuplas

Na implementação realizada existe um espaço de tuplas denominado `GLOBAL`, criado no início da execução do *UbiCentre*. Os outros espaços de tuplas são criados pelo ad-

[1] `www.json.org`

ministrador do sistema através de um arquivo de configuração e são subconjuntos desse espaço de tuplas maior. Esses novos espaços de tuplas são chamados de domínios (representados por elipses na figura) e podem ser criados, também, como subconjunto de outros domínios criando, assim, domínios aninhados que podem representar escopos. Melhorias serão realizadas para permitir aos agentes a criação e destruição de espaços de tuplas dinamicamente.

Os domínios possuem uma identificação única em todo o *UbiCentre*, o qual deve ser diferente de `GLOBAL`. Essa identificação é formada pelas identificações dos domínios que o contém separados pelo caractere '`.`' (ponto final), por exemplo: `building.2ndfloor.lab2` e `building.2ndfloor.lab3`. Esses domínios mantêm entre si relações de conjunto e subconjunto e, assim, pelas relações de pertinência entre conjuntos, as propriedades dos espaços de tuplas apresentadas no capítulo 4 são mantidas. Dessa forma, se uma operação `read` for realizada no domínio `building`, as tuplas presentes nos domínios `building.2ndfloor`, `building.2ndfloor.lab2` e `building.2ndfloor.lab3` são consideradas, mas se a operação for no domínio `building.2ndfloor.lab2` somente as tuplas nele são avaliadas.

5.1.2 Tuplas e Padrões de Tuplas

As tuplas e os padrões de tuplas são implementados de acordo com o diagrama de classes presente na Figura 5.2. Basicamente, tanto uma tupla (classe `Tuple`) como um padrão de tupla (classe `Pattern`) são uma coleção (classe `AbstractFieldCollection`) de campos (classe `AbstractField`). O que difere um do outro é o método `matches` presente na classe `Tuple`. Já a classe `Query` representa o conceito de `query` apresentado no capítulo 4 e é formada por um objeto `Pattern` e uma representação de filtro, que é um objeto do tipo `java.lang.String` que será melhor detalhado na seção 5.1.3.

Cada campo de tupla (classe `TupleField`) e de padrão de tupla (classe `PatternField`) possui três propriedades: `name`, `value` e `type`. O campo `type` é do tipo `java.lang.String` e representa o tipo do campo através de um valor textual. Esses valores representam os tipos possíveis presentes na especificação do capítulo 4 e são determinados dinamicamente de acordo com o valor do campo `value` usado na inicialização do campo. No caso do padrão de tupla, o seu valor pode ser qualquer um desses valores textuais, caracterizando-o como um campo *formal* (representado pelo método `isFormal`). Os valores textuais e seus tipos relacionados são:

`"?boolean":` tipo `java.lang.Boolean`;

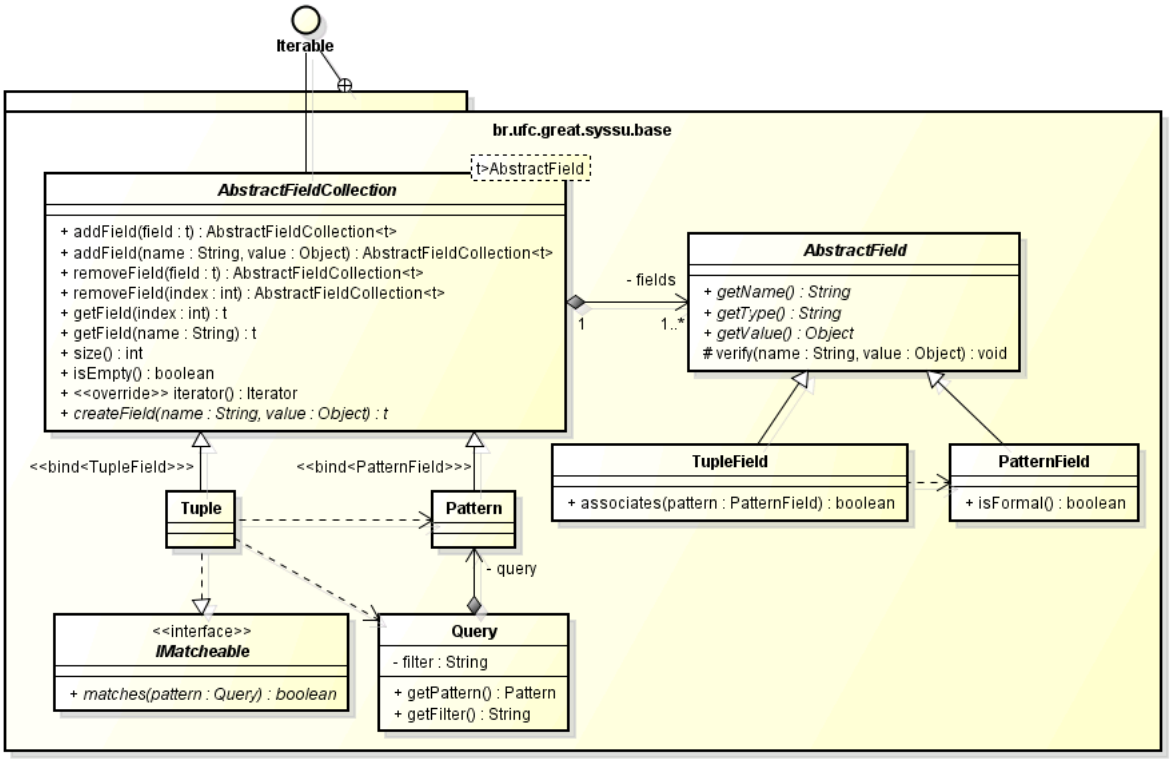

Figura 5.2: Diagrama de classes de implementação das tuplas e padrões de tuplas.

`"?integer":` tipos `java.lang.Short`, `java.lang.Integer` e `java.lang.Long`;

`"?float":` tipos `java.lang.Float` e `java.lang.Double`;

`"?string":` tipo `java.lang.String`;

`"?array":` tipo `java.util.List`; e

`"?object":` tipo `br.ufc.great.syssu.base.Tuple`.

`"?":` para `null`.

Nas mensagens trocadas entre os *UbiBrokers* e o *UbiCentre* as tuplas e os padrões de tuplas são representadas de acordo com a tecnologia de *RPC* utilizada, pois ela que determina como o objeto deve ser serializado para transmissão. No caso da *JSON-RPC*, elas são representadas através da notação *JSON*, cujo alguns exemplos são mostrados na Listagem 5.1.

```
1 /* Exemplos de tuplas */
2 {"service": "printer", "color": false, "file": "teste.pdf"}
3 {"user": "Fabricio", "lat": 28.9, "long": 32.7}
4 {"user": "Fabricio", "out": false}
5 {"room": "lab2", "env": {"temp": 26, "hum": 80}}
```

```
6 {"method": "taxes", "value": [12, 23, 45.5, 156.78]}
7 /* Exemplos de padroes de tuplas */
8 {"?": "?"}
9 {"?": "?object"}
10 {"object": "?"}
11 {"object": "?object"}
12 {"?": "?string", "?": "?array"}
13 {"service": "?", "description": "?"}
14 {"color": "?boolean", "file": "teste.pdf"}
15 {"user": "?string", "lat": "?float", "lon": 32.7}
```

Listagem 5.1: Exemplos de tuplas e de padrões de tuplas em *JSON*.

Através dessa representação é possível atender ao um subconjunto dos requisitos de sintaxe (os outros serão abordados posteriormente neste capítulo) determinados na seção 4.2.2 do capítulo 4:

- Representar os tipos necessários (apesar do *JSON* apresentar o tipo `Number` para todos os tipos de números, é possível fazer uma diferenciação entre números inteiros e reais);
- Determinar, pela representação, nome, tipo e valor dos campos nas tuplas e padrões de tuplas; e
- Especificar e determinar os tipos em campos *formals* de padrões de tuplas, além do valor `null`.

5.1.3 Filtros

Os filtros representam operações lógicas que determinam se uma tupla é adicionada ao resultado de uma operação de consulta em um espaço de tuplas ou não. Para tanto, eles realizam operações sobre os campos da tupla e determinam, de acordo com certos critérios, se a tupla se enquadra corretamente. A expressividade dessas operações e dos critérios é determinada pela tecnologia escolhida para a sua implementação. A tecnologia escolhida na implementação realizada foi a linguagem *Javascript* (padrão ECMA-262[2]). Ela foi escolhida por ser uma linguagem interpretada de amplo uso e por possuir expressividade considerável em termos de construções de linguagem. Na implementação foi utilizado a tecnologia *Rhino*[3] para gerenciamento da execução dos códigos *Javascript* no *UbiCentre*.

[2] `http://www.ecma-international.org/publications/standards/Ecma-262.htm`
[3] `http://www.mozilla.org/rhino/`

Contudo, uma linguagem nova e específica de domínio (*DSL*) para o *SysSU* poderia ser criada para a construção de filtros. Porém, seria um trabalho muito custoso e o *Javascript* atende às necessidades.

A Listagem 5.2 apresenta a sintaxe básica que um filtro deve seguir. Ele é uma função *Javascript* que recebe como argumento um objeto da classe `Tuple` e retorna um valor do tipo `boolean`. Além disso, todo filtro tem acesso a duas variáveis globais que são gerenciadas pelo *UbiCentre*: `domain` e `properties`. A primeira dá acesso ao domínio onde a operação de consulta está sendo executada. Por meio dessa variável é possível realizar operações (representadas nos comentários na Listagem 5.2) sobre o domínio, inclusive sobre seus subdomínios. Através dessa variável é possível implementar lógicas de sensibilidade ao contexto necessária para a coordenação em sistemas ubíquos, como demonstrado no capítulo 4. Contudo, devem existir domínios que armazenem as informações contextuais necessárias. Já a variável `properties` representa um conjunto nome/valor, instanciada no início da execução da operação e destruída no final, que possui o objetivo de armazenar informações entre as execuções do filtro.

```
1 function filter(tuple) {
2   //domain.getDomain("name")
3   //domain.put(tuple)
4   //domain.read(query)
5   //domain.readone(query)
6   //domain.take(query)
7   //domain.takeone(query)
8   //properties.put(name, value)
9   //properties.get(name)
10  return true;
11 }
```

Listagem 5.2: Formato dos filtros.

A Listagem 5.3 mostra a implementação em *Javascript* do filtro exemplificado no capítulo 4. No caso, o objetivo do filtro é determinar se o campo *temperature* da tupla possui o menor valor em todas as tuplas. Para isso, nas linhas 2-5 é instanciado um objeto `Query` para ser utilizada na execução do da operação `read`. Essa execução é realizada no mesmo domínio que o filtro está sendo executado através da variável `domain` (linha 6). Após a execução da operação `read`, é verificado se a tupla utilizada como argumento (variável `tuple`) possui o menor valor para o campo *temperature* quando comparado aos das tuplas resultantes da execução da operação (linhas 7-12).

```
1 function filter(tuple) {
2   var query = new Query(new Pattern()
3     .addField("sensor_id", "?integer")
4     .addField("temperature", "?float"),
5       null);
6   var values = domain.read(query);
7   for (i=0; i < values.length; i++) {
8       if(values[i].getField("temperature").getValue())
9         < tuple.getField("temperature").getValue())
10        return false;
11  }
12  return true;
13 }
```

Listagem 5.3: Exemplo de filtro - versão 1.

A execução do filtro descrita no parágrafo anterior é realizada para cada tupla relacionada com o padrão utilizado na operação de pesquisa (`{"sensor_id":"?integer", "temperature":"?float"}`). Assim, para cada tupla relacionada, a operação `read` é executada, o que torna essa consulta dispendiosa. De forma a melhorar a performance, a Listagem 5.4 apresenta uma nova versão do filtro que usa a operação `readone` (linha 9) com um filtro que verifica se a tupla tem o campo *temperature* menor (linhas 5-8) no lugar da operação `read`. Esse exemplo mostra a possibilidade de usar filtros dentro de filtros, uma vez que as operações disponíveis são similares às disponibilizadas pelo *UbiBroker* aos agentes.

```
1 function filter(tuple) {
2   var query = new Query(new Pattern()
3     .addField("sensor_id", "?integer")
4     .addField("temperature", "?float"),
5       "function filter(tuple) {" +
6       "   return tuple.getField('temperature').getValue() < " +
7             tuple.getField("temperature").getValue() + ";" +
8       "}");
9   return domain.readone(query) == null;
10 }
```

Listagem 5.4: Exemplo de filtro - versão 2.

Contudo, o filtro da Listagem 5.4 ainda precisa executar operações desnecessárias, pois mesmo que a tupla com o menor valor de temperatura tenha sido processada, todas

as tuplas restantes são analisadas. Esse é um exemplo propício para a utilização da variável `properties`. A Listagem 5.5 apresenta uma nova versão do filtro. Nele, a variável `properties` é utilizada para armazenar o valor lógico *done*, caso em que já foi encontrada a menor temperatura, e *values*, que representa os valores obtidos pela operação `read`. Devido a verificação realizada na linha 2, se já foi encontrado o menor valor, então o filtro não realiza mais nenhum processamento além de invalidar a tupla. Além disso, pela verificação na linha 5, a operação `read` é executada somente uma vez.

```
1 function filter(tuple) {
2   if(properties.get("done") != null && properties.get("done")) {
3     return false;
4   }
5   if(properties.get("values") == null) {
6     var query = new Query(new Pattern()
7       .addField("sensor_id", "?integer")
8       .addField("temperature", "?float"),
9         null);
10    properties.put("values", domain.read(query));
11  }
12  var values = properties.get("values");
13  for (i=0; i < values.length; i++) {
14      if(values[i].getField("temperature"))
15        < tuple.getField("temperature"))
16        return false;
17  }
18  properties.put("done", true);
19  return true;
20 }
```

Listagem 5.5: Exemplo de filtro - versão 3.

5.1.4 Sintaxe das Mensagens

Como mencionado na seção 4.2.2 do capítulo 4, a sintaxe das mensagens trocadas entre os *UbiBrokers* e o *UbiCentre* é determinada de acordo com a tecnologia de *RPC* escolhida. A Listagem 5.6 apresenta a descrição[4] (sintaxe) da operação `put`. Essa operação tem como parâmetros o nome do domínio e a tupla (representada de acordo com a seção 5.1.2) e

[4]Na descrição dessa implementação será mostrada somente as mensagens relacionadas ao *JSON-RPC*. Será usada a *JSON Schema Service Descriptor* disponível em `http://groups.google.com/group/json-rpc/web/json-schema-service-descriptor?pli=1`.

não existe valor de retorno. A Listagem 5.7 mostra alguns exemplos de mensagens que invocam o método `put` em domínios distintos e com tuplas diferentes.

```
"put": {
  "type": "method",
  "params": [{"name": "domain", "type": "string", "required": true},
             {"name": "tuple", "type": "object", "required": true}]
}
```

Listagem 5.6: Descrição da sintaxe da operação `put`.

```
-->
{"jsonrpc": "2.0", "method": "put",
 "params": {
     "domain": "environment.users",
     "tuple": {"user": "fabricio", "location": "Lab. de Pesquisa 1"}
  }
}
-->
{"jsonrpc": "2.0", "method": "put",
 "params": {
     "domain": "environment.devices",
     "tuple": {"device": "ASDR455TGG6U",
            "description": {"type": "tablet", "model": "iPad",
               "SO_version": "4.3.2", "memory": 1204}}
  }
}
```

Listagem 5.7: Exemplos de mensagens para o método `put`.

Para as operações `read`, `readsync`, `take` e `takesync` a sintaxe é a apresentada na Listagem 5.8. Apesar da sintaxe apresentada ser da operação `read`, ela é a mesma para as outras operações, mas com o nome do método apropriado. Já para as operações `readone`, `readonesync`, `takeone`, e `takeonesync`, a Listagem 5.9 apresenta a sua sintaxe. Nessas duas descrições, os parâmetros especificados são o nome do domínio o qual a operação será realizada, o padrão de tupla e o filtro. A principal diferença entre elas é o tipo de retorno, a primeira retorna um conjunto de tuplas como resultado e a segunda somente uma tupla (em ambas um valor `null` pode ser retornado).

```
"read": {
  "type": "method",
  "returns": "array",
  "nullable": "true",
  "params": [{"name": "domain", "type": "string", "required": true},
             {"name": "pattern", "type": "object", "required": true},
             {"name": "filter", "type": "string", "required": true}]
}
```

Listagem 5.8: Descrição da sintaxe da operação `read`.

```
"readone": {
  "type": "method",
  "returns": "object",
  "nullable": "true",
  "params": [{"name": "domain", "type": "string", "required": true},
             {"name": "pattern", "type": "object", "required": true},
             {"name": "filter", "type": "string", "required": true}]
}
```

Listagem 5.9: Descrição da sintaxe da operação `readone`.

A Listagem 5.10 mostra alguns exemplos de mensagens relacionadas a essas operações. No primeiro exemplo, o método `take` é invocado com o objetivo de retirar e retornar todas as tuplas que representam a posição do usuário de identificação *fabricio*. Caso nenhuma tupla exista, o valor `null` é retornado como resultado da operação. Já o segundo exemplo invoca o método `takeonesync` com o objetivo de retirar e retornar somente uma tupla que represente um dispositivo do tipo *tablet* contido no domínio `"environment"`. Caso nenhuma tupla exista, o *UbiCentre* não responde ao *UbiBroker* até que uma tupla com essas características seja inserida no domínio ou em qualquer um de seus subdomínios.

```
-->
{"jsonrpc": "2.0", "method": "take", "id": 1
 "params": {
    "domain": "environment.users",
    "tuple": {"user": "fabricio", "location": "?string"},
    "filter": "function filter(tuple) { return true; }"
  }
}
<--
{"jsonrpc": "2.0", "id": 1,
  "result": {"user": "fabricio", "location": "Lab. de Pesquisa 1"}}
```

```
12
13 -->
14 {"jsonrpc": "2.0", "method": "takeonesync", "id": 2
15  "params": {
16     "domain": "environment",
17     "pattern": {"device": "?string", "description": "?object"},
18     "filter":
19       "function filter(tuple) {
20         return tuple.getField('description')
21           .getField('type').getValue() == 'tablet';
22     }"
23   }
24 }
25 <--
26 {"jsonrpc": "2.0", "id": 2,
27       "result": {"device": "ASDR455TGG6U",
28             "description": {"type": "tablet", "model": "iPad",
                  "SO_version": "4.3.2", "memory": 1204}}
```

Listagem 5.10: Exemplos de mensagens para as operações `take` e `takeonesync`.

Em relação aos requisitos de sintaxe do *SysSU* (seção 4.2.2 do capítulo 4), em todas as descrições apresentadas, o nome do espaço de tuplas (representado como domínio) é obrigatório. Já em relação as exceções, no *JSON-RPC* elas não são determinadas na descrição do serviço, mas podem ser retornadas em mensagens de erros. Cabe ao *UbiBroker* interpretar essas mensagens e notificar os agentes sobre a ocorrência da exceção.

5.1.5 Eventos

Como mencionado no capítulo 4, as tecnologias de *RPC* utilizadas na implementação do *SysSU* devem prover mecanismos de *callback*. Isso é necessário para que os *UbiBrokers* recebam as notificações de eventos apropriadamente, pois o uso de tal mecanismo permite que o *UbiCentre* envie diretamente mensagens aos *UbiBrokers* associados a ele.

Os *web services SOAP* já possuem implementações com *callbacks* (QIAN; LIU; TAO, 2006), mas o *JSON-RPC* não. Devido a isso, foi implementado um mecanismo de *callback* para o *JSON-RPC* com a finalidade de enquadrá-lo nos requisitos para utilização na implementação do *SysSU*. Esse mecanismo constitui-se em uma contribuição secundária desta dissertação e pode ser usada em outros contextos e em outras soluções.

Os *callbacks JSON-RPC* são baseados na versão 2.0 do *JSON-RPC*, que se diferencia

da versão anterior, principalmente, pela adição dos parâmetros nominais e dos códigos de erros. Nessa implementação, uma mensagem de subscrição deve possuir os seguintes campos (no *JSON* a ordem dos campos não é importante):

`event:` representa o nome do evento que o cliente deseja receber notificações (substitui o campo `method` nos outros tipos de mensagens);

`params:` um conjunto de objetos *JSON* que determina os parâmetros usados para as notificações de acordo com o evento escolhido em `event` (similar ao campo de mesmo nome dos outros tipos de mensagens);

`id:` um valor de qualquer tipo usado para verificação das respostas (similar ao campo de mesmo nome dos outros tipos de mensagens);

`port:` um número que indica a porta *TCP* que a implementação do *JSON-RPC* em execução no cliente fica esperando pelas notificações; e

`jsonrpc:` um texto especificando a versão do protocolo *JSON-RPC* que deve ser exatamente 2.1.

Já a resposta para mensagens de subscrição são formadas pelos seguintes campos:

`result:` valor numérico, gerado pelo servidor de notificações, obrigatório em caso de sucesso e omitido em caso de falhas, que representa a identificação da subscrição realizada (no *SysSU* ele é único por *UbiCentre* e usado como identificador de reação);

`error:` um objeto *JSON* que representa as informações relacionadas a algum error ocorrido na subscrição, obrigatório em casos de falha, omitido em caso de sucesso e possui a mesma sintaxe e semântica usada no caso de mensagem de invocação de métodos;

`id:` o mesmo valor usado para esse campo na mensagem de subscrição;

`jsonrpc:` um texto especificando a versão do protocolo *JSON-RPC* que deve ser exatamente 2.1.

No *SysSU*, o único evento disponível para subscrições é o relacionado a operação `put`, cuja descrição é mostrada na Listagem 5.11. As Listagens 5.12 e 5.13 mostram exemplos de mensagens de subscrição e de respostas em caso de sucesso e em caso de falha. Na primeira, uma subscrição para o evento `put` com o padrão `{"user":"fabricio",`

`"location":"?string"}` é realizada com sucesso e o valor `12568` é gerado pelo *UbiCentre* e retornado ao *UbiBroker*. No segundo caso, a tentativa de subscrição ao evento `pub` resulta em uma mensagem de erro com o código `-32601` (padronizado no *JSON-PRC*).

```
"put": {
  "type": "event",
  "returns": "object",
  "params": [{"type": "string", "name": "domain", "required": true},
             {"type": "object", "name": "pattern", "required": true},
             {"type": "string", "name": "filter", "required": true}]
  }
```

Listagem 5.11: Descrição da sintaxe da subscrição de eventos.

```
-->
{"jsonrpc": "2.1", "event": "put", "id": 3, "port": 8181
 "params": {
    "domain": "environment.users",
    "pattern": {"user": "fabricio", "location": "?string"},
    "filter": "function filter(tuple) {return true;}"
  }
}
<--
{"jsonrpc": "2.1", "id": 3, "result": 12568}
```

Listagem 5.12: Exemplos de mensagens de subscrição para evento `put`.

```
-->
{"jsonrpc": "2.1", "event": "pub", "id": 4, "port": 8181
 "params": {
    "domain": "environment",
    "pattern": {"device": "?string", "description": "?object"},
    "filter": "function filter(tuple){
            return tuple.getField('description').getField('type') ==
               'tablet';
          }"
  }
}
<--
{"jsonrpc": "2.1", "id": 4,
    "error": {"code": -32601, "message": "Procedure not found."}
```

Listagem 5.13: Exemplos de mensagens de subscrição de eventos com erro.

O valor no campo `result` nas mensagens de sucesso no *SysSU* é usado para dois propósitos: (1) para o *UbiBroker* diferenciar as notificações realizadas; e (2) o *UbiBroker*

executar as operações `unsub` e `check`. As notificações são realizadas através das mesmas mensagens de resultado de invocação de métodos, mas o campo `id` é preenchido com a identificação da subscrição e não com o campo `id` presente na mensagem de subscrição. Dessa forma, o cliente (no caso o *UbiBroker*) consegue diferenciar qual das subscrições que ele realizou se refere à notificação recebida. A Listagem 5.14 mostra dois exemplos de notificações relacionadas à subscrição exemplificada na Listagem 5.12.

```
<--
{"jsonrpc": "2.1", "id": 12568,
  "result": {"user": "fabricio", "location": "Lab. de Pesquisa 1"}}
<--
{"jsonrpc": "2.1", "id": 12568,
  "result": {"user": "fabricio", "location": "Copa"}}
```

Listagem 5.14: Exemplos de mensagens de notificação de eventos.

5.1.6 Serviços

Todo agente que deseje registrar um serviço no *UbiCentre* precisa invocar a operação `reg`, descrita na Listagem 5.15. Os parâmetros dessa operação são o domínio que o serviço ficará registrado, uma tupla que descreve o serviço, um padrão de tupla que especifica o formato dos parâmetros para a invocação do serviço e a porta *TCP* do serviço. Essa operação retorna um identificador único para o serviço no domínio escolhido (similar às subcrições). Para desregistrar o serviço, o agente executa a operação `unreg` utilizando esse identificador gerado. Com o serviço devidamente registrado no *UbiCentre*, o *UbiBroker* utilizado pelo agente cria um serviço *JSON-RPC* cujo nome é o identificador recebido como resposta da operação de registro. Esse serviço, então, fica esperando por solicitações na porta *TCP* especificada. Contudo, somente o *UbiCentre* pode realizar a invocação do serviço diretamente. A seção 5.2 detalha melhor esse procedimento.

```
"reg": {
  "type": "method",
  "returns": "number",
  "params": [{"name": "domain", "type": "string", "required": true},
            {"name": "description", "type": "object", "required": true},
            {"name": "params", "type": "object", "required": false}
            {"name": "port", "type": "number", "required": true}]
}
```

Listagem 5.15: Descrição da sintaxe da operação `reg`.

Por sua vez, o agente que deseje utilizar o serviço registrado precisa ter conhecimento do identificador do serviço e o domínio o qual ele está registrado. A Listagem 5.16 mostra a descrição da operação `descovery`, a qual realiza uma pesquisa, similar à operação `read`, utilizando o padrão de tupla e o filtro passados como argumentos, nas descrições dos serviços registrados no domínio determinado no parâmetro `domain`. A execução dessa operação retorna uma coleção de objetos *JSON* do tipo `Service`, também descrito na Listagem 5.16, que contém o identificador e os parâmetros de invocação do serviço na forma de um padrão de tupla.

```
"descovery": {
  "type": "method",
  "returns": {"type":"array", "items":{"type":"Service"},
  "nullable": "true",
  "params": [{"name": "domain", "type": "string", "required": true},
             {"name": "pattern", "type": "object", "required": true},
             {"name": "filter", "type": "string", "required": true}]
}

{"name":"Service",
  "properties":{
    "id":{"type":"number",
      "description":"Identificador do servico.",
      "required":true
    },
    "param":{"type":"object",
      "description":"Parametros na forma de um padrao de tupla.",
      "required":false
    }
  }
}
```

Listagem 5.16: Descrição da sintaxe da operação `descovery`.

Com o conhecimento do identificador e dos parâmetros de invocação do serviço, o agente invoca o serviço através da operação `invoke`, descrita na Listagem 5.17. Ao receber uma solicitação de invocação de serviço, o *UbiCentre* pesquisa pelo serviço que possui o identificador usado como argumento na mensagem e o invoca. Nessa invocação, o *UbiCentre* utiliza o endereço *IP*, adquirido no momento do registro do serviço, a porta *TCP*, determinada no registro do serviço, e o argumento presente na mensagem `invoke`. Essa comunicação direta entre o *UbiCentre* e o *UbiBroker* não é realizada pelo mecanismo de *callback* utilizado para a notificação de eventos. O *UbiBroker* possui o mesmo mecanismo

de serviços *JSON-RPC* que o *UbiCentre*, possuindo, assim, duplo papel: cliente e servidor. Esse mesmo procedimento é usado no *UbiBroker SOAP*.

Após a execução do serviço presente no agente, o *UbiCentre* encaminha o resultado dessa execução para o agente que invocou o serviço por meio do *UbiCentre*. Nenhum *UbiBroker* pode invocar um serviço em outro *UbiBroker* diretamente, somente o *UbiCentre* tem permissão para as invocações. Isso é necessário para se garantir o desacoplamento entre os agentes.

```
"invoke": {
  "type": "method",
  "returns": "array",
  "nullable": "true",
  "params": [{"name": "domain", "type": "string", "required": true},
          {"name": "id", "type": "number", "required": true},
          {"name": "param", "type": "object", "required": false}]
}
```

Listagem 5.17: Descrição da sintaxe da operação `invoke`.

5.1.7 Agregadores

Na implementação realizada, os agregadores são representados por pacotes *Java* (arquivos *.jar*) localizados em um diretório específico que possui alguma classe concreta que estende a classe abstrata *Aggregator* (representada no diagrama de classes presente na Figura 5.3). O *UbiCentre* em si é representado por um único arquivo *.jar* executável. O diretório de instalação dos agregadores deve possuir o nome *aggregators* e ficar no mesmo diretório do *UbiCentre.jar*. Nesse diretório, existe um arquivo *XML* de nome *config.xml* que representa as configurações dos agregadores. Nele estão determinados quais agregadores devem ser considerados pelo *UbiCentre* e quais os domínios que eles são aplicados. O *UbiCentre* é capaz de detectar qualquer alteração nesse arquivo e carregar dinamicamente qualquer agregador em tempo de execução. A Figura 5.4 apresenta um exemplo de implantação dos agregadores.

As Listagens 5.18 e 5.19 apresentam parcialmente a implementação dos agregadores exemplificados no capítulo 4. Ambas as classes estendem a classe abstrata `Aggregator` e implementam seus dois método abstratos: `check` e `aggregate`. Além disso, a classe `Aggregator` disponibiliza dois métodos importantes para uso de suas subclasses: `getDomain` e `proceed`. O primeiro retorna a referência ao domínio o em que a operação foi iniciada através da qual o agregador pode realizar as operações disponíveis sobre esse domínio e

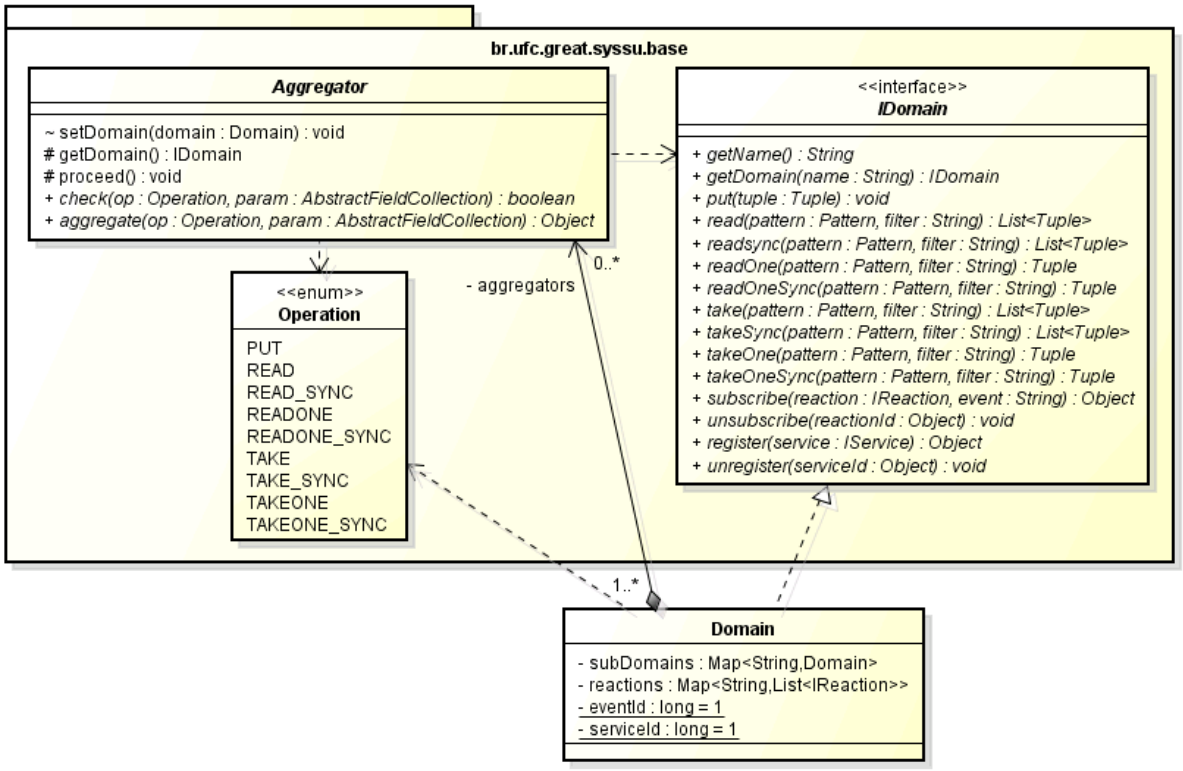

Figura 5.3: Classes relacionadas a agregadores no *UbiCentre*.

qualquer um dos seus subdomínios. O segundo método retorna o resultado padrão da operação como se o agregador não existisse.

```
1 public class TemperatureAverager extends Aggregator {
2 ...
3   @override
4   public Object aggregate(Operation op, AbstractFieldCollection p) {
5     List<Tuple> result = new ArrayList<Tuple>();
6     Pattern pattern = new Pattern()
7       .addField("sensor_id", "?integer")
8       .addField("temperature", "?float")
9       .addField("humidity", "?float");
10    List<Tuple> tuples = getDomain().read(pattern, null);
11    double sum = 0.0;
12    for(Tuple t : tuples)
13      sum += (double)t.getField("temperature").getValue();
14    result.add(new Tuple(
15      .addField("temperature_average", sum/tuples.size())));
16    return result;
17  }
18 }
```

Listagem 5.18: Exemplo de agregador.

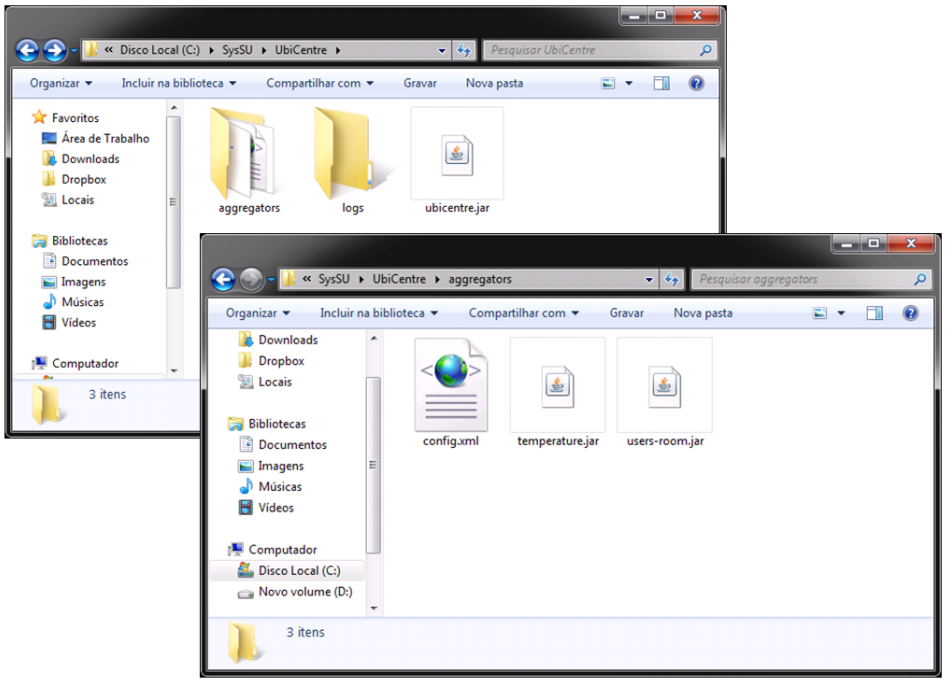

Figura 5.4: Exemplo de implantação do *UbiCentre* e dos agregadores.

```
1 public class RoomMettingNotification extends Aggregator {
2 ...
3   @override
4   public Object aggregate(Operation op, AbstractFieldCollection p) {
5     Tuple tuple = new Tuple(
6       .addField(getField("user"))
7       .addField(getField("room"))
8       .addField("user", Calendar.getInstance().getTimeInMillis());
9     getDomain().take((Pattern)p, null);
10    getDomain().put(tuple);
11    Pattern pattern = new Pattern()
12      .addField("user", "?string")
13      .addField("room", "sala de reuniao")
14      .addField("time", "?integer");
15    String filter = "function filter(tuple) { return " +
16      "new Date(new Date().UTC() - tuple.getField('time').getValue())" +
17        ".getMinutues() >= 5;" +
18      "}";
19    List<Tuple> tuples = getDomain().read(pattern, filter);
20    if(tuples.size() == 10) {
21      List<String> users = new ArrayList<String>(10);
22      for(Tuple t : tuples)
23        users.add(t.getField("user").getValue());
24      getDomain().put(new Tuple().
```

```
25          addField("users", users).addField("room", "sala de reuniao"));
26      }
27      return null;
28    }
29 }
```

Listagem 5.19: Exemplo de agregador para notificações.

Em toda execução de qualquer operação da classe `Domain`, com exceção dos relacionados a eventos e serviços, os agregadores para aquele domínio são verificados através da execução do método `check`. Caso o valor de retorno seja `true`, o método `aggregate` é executado com os mesmos argumentos usados na operação. O resultado da execução do método `aggregate` é, então, utilizado pelo *UbiCentre* como se fosse o resultado da operação.

5.2 Os *UbiBrokers*

A Figura 5.5 mostra um diagrama de classes dos *UbiBrokers* implementados. Tanto a implementação do *UbiBroker* como do *UbiCentre* compartilham um conjunto de classes e interfaces do pacote *br.ufc.great.syssu.base* (na linguagem *Java*) ou do *namespace SysSU.Base* (na linguagem *C#*). Algumas dessas classes e interfaces foram abordados no capitulo anteriror, sendo a principal a interface `IDomain`.

A Listagem 5.20 mostra como é realizada a inicialização de uma instância de um *UbiBroker* (usando a linguagem *C#*). Na inicialização, através de um método estático, é determinado o endereço *IP* e a porta *TCP* para a comunicação com o *UbiCentre* e a porta *TCP* para mensagens de *callback* e de invocação de serviços (linha 9). Esse método inicia uma nova *thread* de execução e dois objetos responsáveis pela comunicação na rede, um que envia mensagens ao *UbiCentre* e outro que fica esperando mensagens de *callback*. Uma opção para esse processo, usada em um dos estudo de casos, é o uso de mensagens *multicast* na rede para determinar automaticamente a localização do *UbiCentre*.

Com o *UbiBroker* instanciado, o método `getDomain` pode ser usado para instanciar os domínios necessários, no caso, instâncias da classe `Domain` (linha 10). As classes `Tuple` e `Pattern` são as mesmas usadas na implementação do *UbiCentre*, ilustradas na Figura 5.2 do capítulo 5. Cabe ao desenvolvedor dos agentes que utiliza a *API* do *UbiBroker* instanciá-las e usar seus métodos para determinar os campos e filtros necessários. Toda a serialização e criação das mensagens é realizada, de forma transparente, por essas classes.

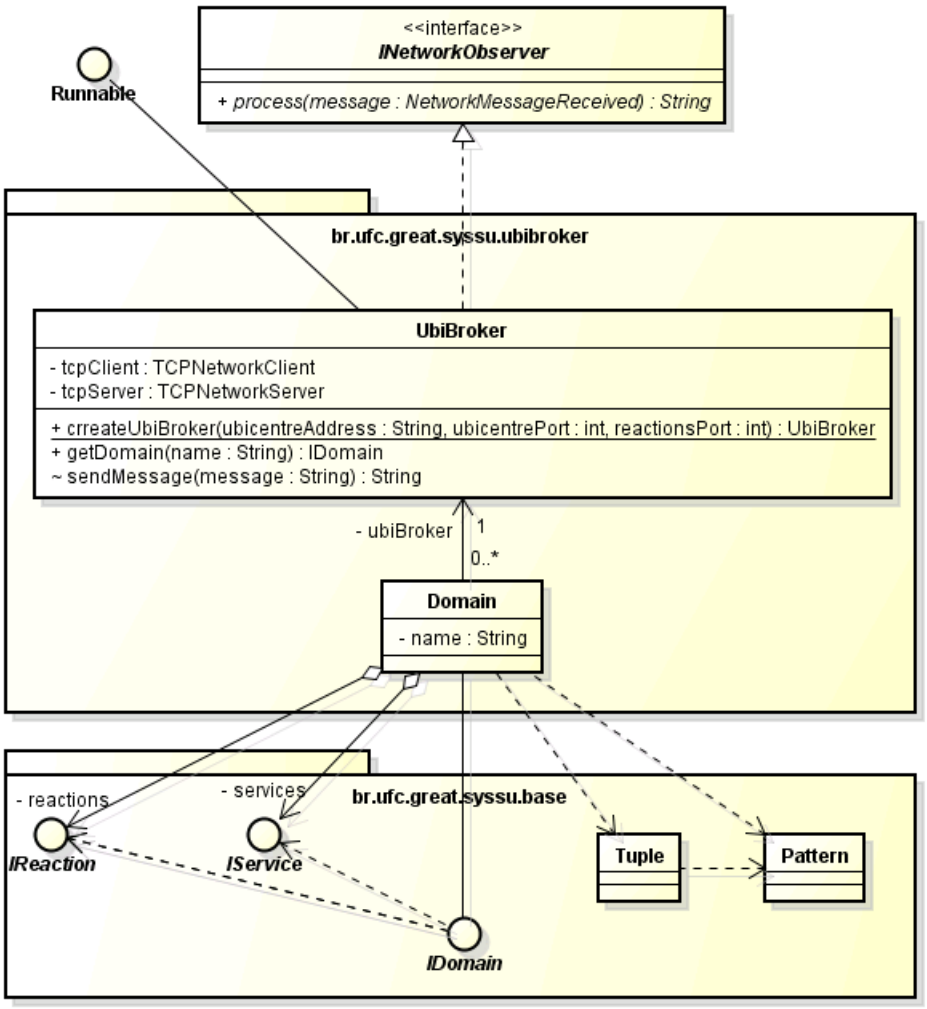

Figura 5.5: Classes relacionadas a implementação dos *UbiBrokers*.

Devido a isso, se o desenvolvedor optar por usar o *UbiBroker SOAP* no lugar do *UbiBroker JSON-RPC*, o código fonte escrito por ele poderá ser o mesmo. Além disso, as respostas são tratadas de forma semelhante, como se estivesse trabalhando com espaços de tuplas armazenados no próprio agente.

```
1 using SysSU.Base;
2 using SysSU.UbiBroker
3 ...
4 private UbiBroker ubiBroker;
5 private IDomain domain;
6 ...
7 try
8 {
9   ubiBroker = UbiBroker.CreateInstane(address, port, callbackPort);
10  domain = ubiBroker.GetDomain("world");
11 }
```

```
12 catch(Exception ex)
13 ...
```

Listagem 5.20: Inicialização do *UbiBroker* em *C#*.

A Listagem 5.21 mostra trechos de códigos de três agentes que utilizam a *API* do *UbiBroker* implementado. O agente A insere uma tupla com a mensagem "Hello World!" para um usuário específico e o agente B fica em um laço infinito esperando, bloqueado por tuplas com tais mensagens. Já o agente C só se interessa por tuplas com mensagens destinadas ao usuário 'fabricio', como determinado pelo filtro utilizado.

```
1  // Agente A:
2  ...
3  public void SayHello(string user)
4    try
5    {
6      var tuple = new Tuple();
7      tuple["message"] = "Hello World! " + user +"!";
8      domain.Put(tuple);
9    }
10   catch{}
11 }
12 ...
13 // Agente B
14 ...
15 try
16 {
17   while(true)
18   {
19     var pattern = new Pattern();
20     pattern["message"] = "?string";
21     var tuple = domain.ReadOneSync(pattern, null);
22     Console.WriteLine(tuple["message"]);
23   }
24 }
25 catch{}
26 ...
27 // Agente C
28 ...
29 try
30 {
31   while(true)
32   {
```

```
33      var pattern = new Pattern();
34      pattern["message"] = "?string";
35      var tuple = domain.ReadOneSync(pattern,
36        "function filter(tuple) {" +
37          "return tuple.getField('message').contains('fabricio')}");
38      Console.WriteLine(tuple["message"]);
39    }
40 }
41 catch{}
42 ...
```

Listagem 5.21: Uso dos métodos do *UbiBroker* em *C#*.

5.2.1 Reações

As reações são implementadas seguindo o padrão de projeto *Observer*, cuja a entidade *Subject* é representa por implementações concretas da interface `IDomain` e a entidade *Observer* por implementações da interface `IReaction` (apresentada nos diagramas de classes das Figuras 5.5 e 5.6). Assim, para um agente receber notificações, é necessário possuir uma instância de uma classe concreta que implemente a interface `IReaction`, cujo principal método é `react`, e registrá-la no objeto da classe `Domain` apropriado usando o método `Subscribe`.

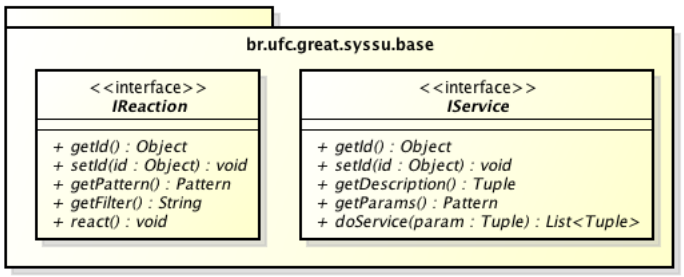

Figura 5.6: Interfaces relacionadas a implementação dos conceitos de reações e serviços.

A Listagem 5.22 mostra a implementação, usando a linguagem C#, do Agente C da Listagem 5.21 usando reações (forma assíncrona). A propriedade `Id` representa a identificação da reação e é atribuída pela *UbiBroker* de acordo com a mensagem de resposta do *UbiCentre*, como mostrado na subseção 5.1.5. Esse mesmo valor é usado pelo *UbiBroker* para desregistrar a reação. Contudo, isso é transparente para o agente, o qual usa, para isso, a referência para o objeto que representa a reação através do método `Unsubscribe` da classe `Domain`.

```
1 public class MyReaction : IReaction
2 {
3   public long Id { get; set; }
4
5   public override Pattern GetPattern()
6   {
7     var pattern = new Pattern();
8     pattern["message"] = "?string";
9     return pattern;
10  }
11
12  public override string GetFilter()
13  {
14    return
15      "function filter(tuple) {" +
16        "return tuple.getField('message').contains('fabricio')}";
17  }
18
19  public override void React(Tuple tuple)
20  {
21    Console.WriteLine(tuple["message"]);
22  }
23 }
```

Listagem 5.22: Exemplo de classe que representa uma reação.

5.2.2 Serviços

A implementação do mecanismo de serviços segue, também, ao padrão de projetos *Observer*, mas a entidade *Observer* é representada por classes concretas da interface `IService` (apresentada nos diagramas de classes das Figuras 5.5 e 5.6). O principal método dessa interface a ser implementado é o método `DoService`, o qual recebe como argumento uma tupla que representa os argumentos para a execução do serviço e pode retornar como resultado um conjunto de tuplas ou um valor `null`. Já os métodos `getDescription` e `getParams` são implementados para retornar uma tupla que descreve o serviço e um padrão de tupla que determina os parâmetros de invocação do serviço, respectivamente.

Para o serviço está disponível para uso por outros agentes, o agente precisa registrá-lo através do método `Register` da classe `Domain` e, assim como as reações, o *UbiBroker* atribui um valor de identificação para o serviço fornecido pelo *UbiCentre*. Esse valor é usado

pelo *UbiBroker* para desregistrar o serviço no *UbiCentre*. Contudo, ele é transparante para o agente, o qual utiliza as referências para as classes concretas de `IService` na execução do método `Unregister`. A Listagem 5.23 mostra uma classe em *Java* concreta que implementa a interface `IService`. Ela representa um serviço que simplesmente imprime "Hello World" com o nome de um usuário no *console* e retorna o valor `null`.

```
1 public class MyService implements IService {
2   private long id;
3   public long getId() {return id;}
4   public void setId(long id) {this.id = id;}
5
6   @override
7   public Tupple getDescription() {
8     return new Tuple().addField("name", "Say Hello");
9   }
10
11  @override
12  public Pattern getParams() {
13    return new Pattern().addField("user", "?string");
14  }
15
16  @override
17  public List<Tuple> doService(Tuple tuple) {
18    System.out.println("Hello World! " +
          tuple.getField("user").getValue() + "!");
19    return null;
20  }
21 }
```

Listagem 5.23: Exemplo de classe concreta de `IService`.

Já a listagem 5.24 mostra o código em *C#* de um outro agente que utiliza esse serviço. Ele realiza o processo de descoberta usando o método `Descovery` usando como argumento um padrão de tupla e um filtro que especifica qualquer serviço que tenha a descrição com um único campo e que esse campo seja do tipo texto e que contenha a palavra "Hello". Depois de descoberto, o agente o invoca usando como argumento o valor contido na variável `user`, independente do nome do parâmetro do serviço.

```
1 ...
2 public void SayHello(string user)
3   try
4   {
5     var description = new Pattern(); description["?"] = "?string";
```

```
6     var services = domain.Descovery(tuple,
7       "function filter(tuple) {" +
8         "return tuple.size() == 1 &&" +
9           "tuple.getField(1).contains('Hello');" +
10      "}");
11    if(services.Count > 0)
12    {
13      var parameter = new Tuple();
14      parameter[services[0].Params[0].Name] = user;
15      services[0].DoService(parameter);
16    }
17  }
18  catch{}
19}
20...
```

Listagem 5.24: Exemplo de descoberta e uso de serviços.

5.3 *GREat Tour*

O *GREat Tour* é uma aplicação para guiar visitações desenvolvido para auxiliar os visitantes presentes no prédio do Grupo de Redes de Computadores e Engenharia de Software (GREat[5]) no campus do Pici da Universidade Federal do Ceará. Basicamente, ele fornece informações dos laboratórios contidos no prédio e dos pesquisadores e desenvolvedores presentes em cada um. Tais informações consistem de especialidades e pesquisas sendo desenvolvidas que são visualizadas pelos usuários através de texto, fotos e vídeos em telefones celulares e smartphones com capacidade de executar *MIDlets JME*. Essas informações são fornecidas por um conjunto de serviços executados em servidores presentes nos laboratórios através de uma rede sem fio *IEEE 802.11*, assim, se constituíndo em um sistema móvel que abrange todo o edifício.

Além disso, ele é enquadrado como sendo sensível ao contexto, pois pode fornecer as informações dos laboratórios de acordo com a posição *indoor* do usuário no prédio. Dessa forma, o usuário não necessita determinar explicitamente na interface gráfica qual laboratório ele se encontra para visualizar as informações. Esse posicionamento é determinado pela utilização de sensores de proximidade[6] transportados pelos usuários e presentes nos laboratórios. Adicionalmente, o sistema é capaz de determinar quais os pesquisadores

[5] `www.great.ufc.br`
[6] *Wireless Sensor Network Professional Kit – 2.4 GHz, Crossbow Technology, Inc*

estão no laboratório ou não através de *logons* na rede *Windows* instalada no prédio. A Figura 5.7 mostra um cenário de uso do *GREat Tour*.

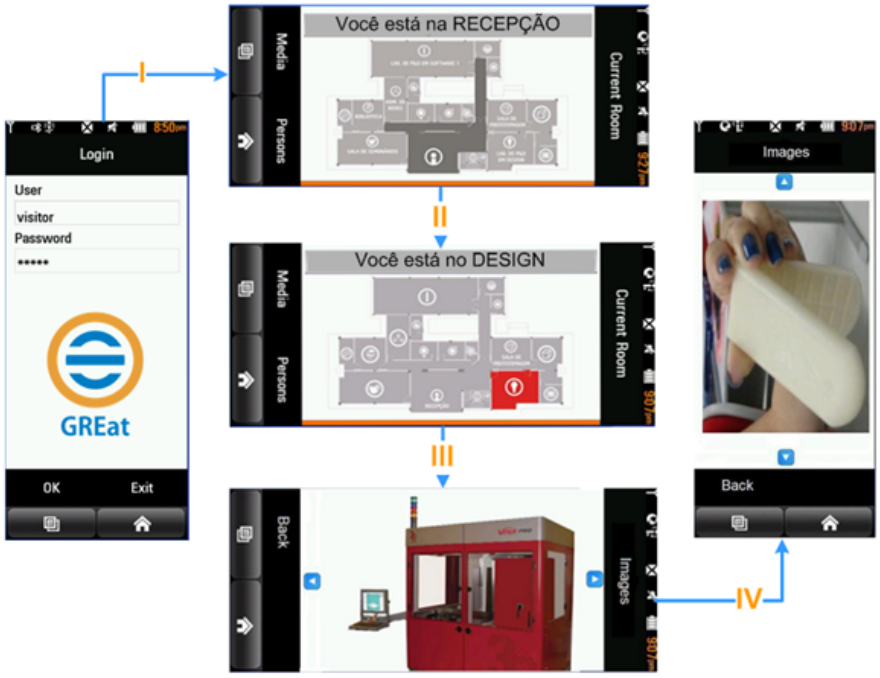

Figura 5.7: Exemplos de telas e de um cenário do *GREat Tour*.

O *GREat Tour* não é um sistemas ubíquo propriamente dito, ele é enquadrado como um sistema móvel e sensível ao contexto. Ele foi desenvolvido como estudo de caso para um projeto de linha de produtos de software para sistemas móveis e sensíveis ao contexto (MARINHO et al., 2010b, 2010a). Contudo, ele apresenta as características da descoberta de serviços automática, da descentralização, da onipresença dos serviços, do comportamento adaptável e da interoperabilidade espontânea. Características, que junto a interoperabilidade, estão relacionadas com o sistema de suporte proposto nesta dissertação.

Na configuração original do *GREat Tour*, a aplicação móvel presente nos dispositivos móveis dos usuários foi desenvolvida e é executada utilizando a plataforma *Java JME CLDC*. Os serviços presentes no ambiente ubíquo foram desenvolvidos utilizando *Java JEE* e *Microsoft .NET*, sendo que o último é executado através de um *wrapper Java* para poder se comunicar apropriadamente com os clientes. Como mecanismo de coordenação é usado a versão mais básica do *LighTS* utilizando o *middleware* de *RPC RME* (PEREIRA et al., 2006). Assim, o *GREat Tour* pode ser considerado como um sistema de uma única plataforma, a *Java*, apresentando heterogeneidade somente em relação aos dispositivos.

Com a necessidade de desenvolver uma nova versão da aplicação para executar em

outra plataforma, no caso o *Google Android*, foi constatado que a forma como o sistema havia sido construído não possuía a interoperabilidade adequada. Uma solução possível seria desenvolver uma nova versão que atendesse às duas plataformas, porém isso apenas postergaria o problema para o momento em que surgisse a necessidade de portar a aplicação para outra plataforma de desenvolvimento. Além disso, foram encontrados alguns problemas no desenvolvimento da versão anterior, onde podemos citar:

- Um dos serviços registra e notifica o *logon* dos usuários na rede *Windows*, no qual o sistema é implantado. Contudo, ele é implementado em *.NET* (versões em outras plataformas não foram viáveis devido a plataforma *Windows Server*), assim, para integrar esse serviço com o *LighTS/RME* foi necessário utilizar um *wrapper* Java.
- O *LighTS/RME* dificulta a adição de novas tuplas com mecanismos de associação diferenciado e, assim, de novos serviços no sistema.
- O *RME* se mostrava instável em alguns casos. Para esse problema, foi implementado uma versão que utilizava *web services*, mas foram encontrados problemas na serialização em *SOAP* dos objetos das classes da biblioteca do *LighTS*.

Dessa forma, uma nova versão do *GREat Tour*, juntamente com os serviços utilizados, foi desenvolvida utilizando o *SysSU* com o propósito de solucionar esses problemas e os novos requisitos. No desenvolvimento da nova versão, o *UbiCentre* já estava implementado e não foi necessário nenhuma manutenção adaptativa para seu uso. O mecanismo de *RPC* escolhido foi o *JSON-RPC* devido as sua performance. A refatoração da aplicação em *JME* não exigiu muito esforço uma vez que ela já era baseada no conceito de tuplas e de eventos baseado em tuplas e a versão em *Android* reutilizou muito do código da versão em *JME*. Contudo, a principal modificação foi a remoção do *wrapper Java* do serviço em *.NET* mencionado. No caso, o serviço se comunica diretamente com o *UbiCentre* através de um *UbiBroker .NET/JSON-RPC*.

A Figura 5.8 mostra a arquitetura da nova versão do sistema, cujo principais agentes presentes são:

1. O agente que determina a localização *indoor* do usuário. Essa localização é a nível de laboratório e é determinada pelos sensores presentes em cada um. Esses sensores, ao detectar a presença de um outro sensor transportado pelo usuário, envia uma mensagem com o nome do usuário e o laboratório para um servidor que os controla que, por sua vez, insere uma tupla de padrão `{"user":"?string",`

`"location":"?string"}` no domínio chamado `users.location`. A Listagem 5.25 apresenta o trecho de código que realiza essa operação. No caso, o agente retira a tupla de localização anterior relacionada ao usuário. Além disso, esse agente disponibiliza um serviço que fornece o histórico das localizações dos usuários.

2. Os agentes alocados em cada laboratório são responsáveis por fornecer as informações sobre o laboratório e os pesquisadores. Para isso, eles disponibilizam dois serviços, através do *UbiCentre*, que podem ser invocados pelos agentes. Um deles fornece quais pesquisadores fazem parte de um determinado laboratório e o outro retorna um conjunto de *URLs* para *download* das informações por parte da aplicação cliente. Os serviços tem como descrição as tuplas `{"name":"researchers", "lab":"[nome do laboratório]"}` e `{"name":"information", "lab":"[nome do laboratório]"}`, repectivamente, e são registrados no domínio `services`. Para invocá-los, são usados os parâmetros na forma `{"room":"?string"}` e `{"room":"?string", "researcher":"?string"}`, respectivamente.

3. O agente que determina quais pesquisadores efetuaram *logon* na rede e, assim, estão presentes. Esse agente insere uma tupla de padrão `{"researcher":"?string", "location":"?string"}` no domínio `researchers`.

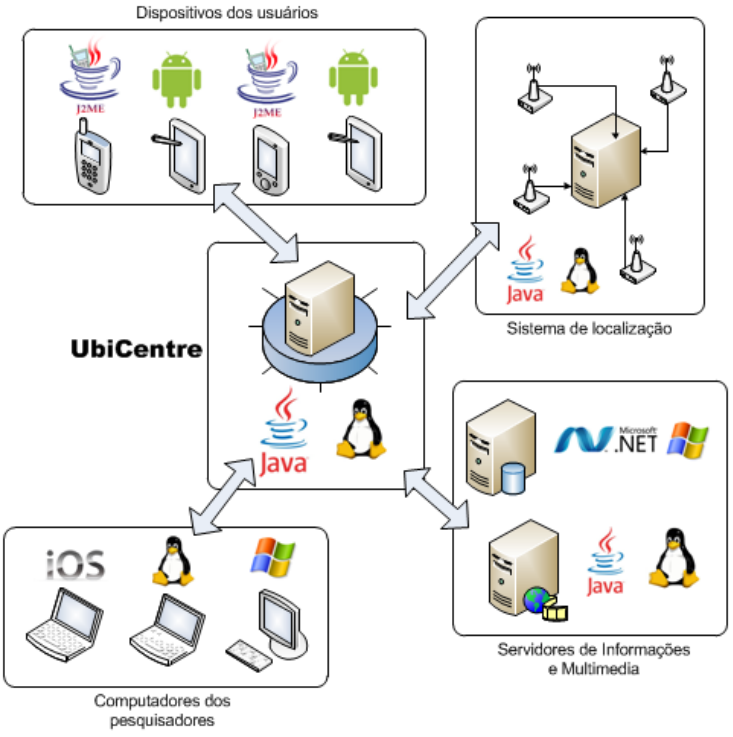

Figura 5.8: Arquitetura da nova versão do *GREat Tour*.

```
1 public void setUserLocation(String user, String loc, UbiBroker ub) {
2   IDomain tp = ub.getDomain("environment.users");
3   tp.take(new Pattern()
4     .addField("user", user).addField("location", "?string"), null);
5   tp.put(new Tuple()
6     .addField("user", user).addField("location", loc));
7 }
```

Listagem 5.25: Exemplo de utilização do *UbiBroker* no *GREat Tour*.

A aplicação presente nos *smartphones* dos usuários não consegue saber a sua localização por si só, para saber essa informação, ela realiza a operação `read` com o padrão `{"user":"[nome do usuário]", "location":"?string"}` no domínio `users`. Com a localização obtida, a aplicação invoca os serviços responsáveis pelas informações do laboratório e dos pesquisadores com essa localização. Além disso, ela verifica quais pesquisadores estão presentes no laboratório para destacá-los na visualização das informações.

Os dois primeiros passos podem ser simplificados com o uso de um agregador registrado no domínio `GLOBAL` que intercepta as operações `read` com o padrão `{"user":"[nome do usuário]"}`. O agregador realiza as duas ações e retorna como resultado as informações necessárias sobre o laboratório e os pesquisadores. Dessa forma, é economizado processamento e transmissão de dados por parte da aplicação cliente.

Uma das vantagens de adotar o sistema de suporte proposto nessa solução é que, além da facilidade de inclusão de uma nova versão da aplicação em outras plataformas, o sistema fica aberto para a inclusão de novas aplicações e serviços desenvolvidos nas plataformas já adotadas (i.e., *Java JME* e *Google Android*) ou em outras (e.g., *Microsoft .NET Compact Framework* ou *Apple iOS*) devido ao desacoplamento e a interoperabilidade fornecida. Além disso, ela permite uma maior expressividade em relação a sensibilidade ao contexto, pois é possível definir novas tuplas e pesquisas alterando somente o código da aplicação ou do serviço.

5.4 *UbiPrinter*

Outro estudo de caso implementado foi uma aplicação de impressão denominado *UbiPrinter*. Esse sistema possui a principal funcionalidade de determinar automaticamente qual a impressora mais adequada para impressão de um documento do usuário. Para tanto, o usuário instala um *driver* especial de impressão em seu computador ou um apli-

cativo no seu *smartphone Android* ou *Windows Mobile*. Esse *driver* e essa aplicação representam um agente que determina a impressora ideal para imprimir documentos do usuário de acordo com as características do documento e das impressoras disponíveis e, principalmente, da localização do usuário.

Foram usadas na implantação 6 impressoras variando em lasers, jato de tintas, coloridas, preto e brancas dispostas em diferentes salas e andares. Em cada uma foi vinculado um agente que fornece o serviço de impressão relacionada àquela impressora. Sua descrição é fornecida por tuplas `{"service":"printer", "location":"[sala da impressora]", "tipo":"[laser ou jato de tinta]", "color":[true ou false], "ppm":[páginas por minuto]}`. De acordo com os requisitos de impressão do usuário, a aplicação realiza uma descoberta de serviços de acordo com essa descrição.

Contudo, uma das funcionalidades do sistema é utilizar a impressora mais próxima do usuário. Para isso, é utilizado os mesmos serviços presentes no *GREat Tour* que determinam a posição do usuário e quais estão logados na rede. A Listagem 5.26 apresenta o filtro utilizado na consulta de serviços que representam impressoras. Esse exemplo mostra a sensibilidade ao contexto oferecida por esse mecanismo. O filtro determina que somente os serviços que com o campo `location` igual ao do usuário `'fabricio'` (portador do dispositivo) sejam retornadas.

```
1 function filter(tuple) {
2   var query = new Query(new Pattern()
3     .addField("user", "fabricio")
4     .addField("location", "?string"),
5     null);
6   var userLocation = domain.readone(query);
7   return userLocation != null &&
8       tuple.getField("locations").getValue()
9         .indexOf(userLocation.getField("location").getValue()) != -1;
10 }
```

Listagem 5.26: Exemplo de filtro sensível ao contexto.

Esse estudo de caso foi desenvolvido para validar, principalmente, o mecanismo de descoberta de serviços. No caso, foram alcançados os seguintes benefícios:

- Os agentes interessados podem realizar um busca usando somente o padrão `{"service":"printer"}` ou qualquer subconjunto dos campos da descrição utilizada;
- Novas impressoras podem ser adicionadas sem afetar qualquer outro agente presente

no sistema;

- A execução do serviço é totalmente desacoplada, o agente cliente não possui conhecimento do serviço que está executando a impressão; e
- Podem ser adicionados novos cliente em outras plataformas, desde que exista o *UbiBroker* apropriado, sem afetar os outros agentes.

Contudo, a principal característica dessa implementação foi a uso do mesmo *UbiCentre* utilizado pelo *GREat Tour* e o compartilhando de um serviço: o de localização do usuário. Assim, as duas aplicações usadas como estudo de caso fazem parte de um único ambiente ubíquo constituído por vários serviços e aplicações que podem ser combinados das mais variadas formas para fornecer novas funcionalidades para os usuários independente das plataformas de desenvolvimento e de execução escolhidas.

5.5 Conclusão

Este capítulo teve o objetivo de descrever de forma sucinta as características das implementações do *UbiCentre* e de *UbiBrokers* realizadas. Nessas implementações, foram escolhidas as tecnologias de *RPC JSON-RPC* e *web services SOAP*, mas somente o uso da primeira foi descrito, mostrando o formato das mensagens que o *UbiCentre* é capaz de receber. Essa descrição serve de guia para o desenvolvimento dos *UbiBrokers* necessários para que os agentes se comuniquem com o *UbiCentre* e, assim, realizem a descoberta de serviços e a coordenação considerando as características dos sistemas ubíquos, como discutido no capítulo 2.

Além disso, este capítulo apresentou uma validação para a solução proposta nesse trabalho de dissertação. Essa validação foi realizada através do desenvolvimento de duas aplicações que, apesar de serem simples, apresentam a volatilidade e a heterogeneidade comum a sistemas ubíquos. Essas aplicações utilizam serviços implementados em *Java SE* e *.NET Framework* e foram desenvolvidas nas plataformas *Java ME*, *Android* e *Windows Mobile* sendo executadas em dispositivos móveis que precisam descobrir e invocar os serviços e interagir entre si de forma desacoplada e transparente. Além disso, em alguns desses cenários são necessários mecanismos sensíveis ao contexto para a realização mais apropriada e otimizada de determinadas funcionalidades oferecidas pelas aplicações aos usuários.

Pelo exposto, a solução se adequou apropriadamente para todos os cenários dos estudos

de casos de forma simples. Essa simplicidade se refere, empiricamente, a complexidade de implementação do código dos agentes que utilizam o *UbiBroker* que, no caso, pelas listagens apresentadas, exigem poucas linhas de código e as classes e métodos são de fácil compreensão.

Uma análise quantitativa não se faz necessária, pois não foram encontrados trabalhos relacionados com as mesmas características que o *SysSU* para realizar comparações relacionadas a quesitos como performance, velocidade de transmissão ou menor consumo de energia. Por exemplo, poder-se-ia realizar uma comparação entre a versão antiga do *GREat Tour* que usa a combinação *LighTS/RME* com a versão que utiliza o *UbiBroker JME/JSON-RPC*. Porém, o resultado não invalidaria o propósito do *SysSU* que é fornecer mecanismos de serviços e de coordenação desacoplados, interoperáveis e sensíveis ao contexto.

6 Conclusão

Na Computação Ubíqua, as dificuldades surgem no projeto das arquiteturas, na modelagem da colaboração entre os componentes de software heterogêneos, na engenharia do sistema, e na comercialização, implantação e validação do sistema como um todo. Novos paradigmas e desafios para o desenvolvimento de software surgem, agora necessitando ser projetado para lidar intrinsecamente com certos requisitos e limitações. Dessa forma, para suprir as características de sistemas ubíquos e obedecendo as suas restrições são necessárias infraestruturas de software apropriadas, como *frameworks*, *middlewares*, sistemas de suporte, entre outros, que auxiliem o seu desenvolvimento, pois essas infraestruturas servem de suporte para o desenvolvedor, fornecendo soluções prontas e reutilizáveis para os principais problemas encontrados.

A grande parte dos desafios encontrados no desenvovimento de tais sistemas está relacionada as suas características peculiares, visto que muitos problemas já são bem solucionados nas áreas de Sistemas Distribuídos e Computação Móvel. Essas características determinam um conjunto de requisitos de software que possibiltam a construção das infraestruturas de software desejadas e, assim, facilitando o desenvolvimento de sistemas ubíquos. O capítulo 2 apresentou com detalhes essas característicos e esses requisitos bem como os seus relacionamentos. O principal ponto levantado nesse capítulo foi a dinamicidade e a heterogeneidade apresentadas pelos sistemas ubíquos. Essas características influenciam consideravelmente os requisitos de sistemas presentes no desenvolvimento de sistemas ubíquos como a coordenação e a descrição/descoberta de serviços.

Contudo, soluções são possíveis para esses problemas, como a adoção de infraestruturas de serviços e de coordenação que sejam desacopladas, sensíveis ao contexto e interoperáveis. Porém, nenhuma infraestrutura com essas características foi encontrada. Devido a isso, este trabalho de dissertação propôs o *SysSU*, uma infraestrutura de software na forma de um sistema de suporte.

6.1 Resultados Alcançados

A principal contribuição deste trabalho foi fornecer em uma única solução uma infraestrutura de serviços e uma infraestrutura de coordenação que são desacopladas, interoperáveis e sensíveis ao contexto. O desacoplamento foi obtido pela adoção dos modelos *Linda* e *publish/subscribe* como base. A interoperabilidade foi alcançada pela definição das primitivas de comunicação utilizando notações de alto nível não dependendo de nenhuma plataforma de desenvolvimento e execução. Já a sensibilidade ao contexto foi fornecida por meios expressivos de armazenar e pesquisar por informações contextuais necessárias. Os capítulos 4 e 5 detalharam como isso foi alcançado.

Além disso, foi fornecida uma implementação do *SysSU* utilizando plataformas de desenvolvimento de software largamente utilizadas. Essa implementação foi desenvolvida tomando como base a descrição da arquitetura e das primitivas de comunicação e foi utilizada no estudo de caso. Ela está disponível para utilização e contribuições em um repositório de código do tipo *Subversion*[1] no endereço `https://syssu.googlecode.com/svn/trunk/`.

6.2 Trabalhos Futuros

Além de algumas melhorias, como uma verificação formal do modelo de interação proposto com a finalidade de mostrar que ele apresenta ou não certas propriedades, como *deadlock*, *livelock* e *starvation*, algumas propostas são apresentadas como trabalhos futuros:

1. Uma notação para a especificação de aplicações que sigam o estilo arquitetônico do *SysSU* para que, de tal forma, possa ser aplicada técnicas para a análise e verificação de especificações;

2. Uma ferramenta para análise, testes e simulações prévias de sistemas e das interações entres agentes;

3. Um framework orientado a tarefas para o desenvolvimento de sistemas ubíquos que utiliza os mecanismos de serviços e de coordenação como base, abstraindo os seus conceitos através de um novo modelo de componentes;

[1] `http://subversion.tigris.org/`

4. Implementaçao de mecanismos de dependabilidade, principalmente em relação a um melhor tratamento de exceções que possam ocorrer na interação entre os agentes; e

5. Fornecimento de mecanismo de segurança nas operações realizadas nos espaços de tuplas como, por exemplo, o uso de permissões.

Por fim, esse sistema de suporte possui, também, o propósito de servir como base para a construção de soluções mais completas para auxiliar no desenvolvimento de sistemas ubíquos. Ele simplesmente fornece os requisitos primários de qualquer sistema distribuído, o qual é a interação entre os componentes de software que compõem a arquitetura do sistema, com o diferencial de considerar as características dos sistemas ubíquos. Já essas novas soluções reutilizam essas funcionalidade primárias através de especializações da arquitetura de referência e da combinação das primitivas para fornecer novas funcionalidades destinadas a resolver requisitos mais complexos.

Uma dessas novas soluções a serem implementadas é uma infraestrutura de contexto desacoplada e interoperável. Ela terá como base a sensibilidade ao contexto fornecida pelo *SysSU* aos requisitos de coordenação e descoberta de serviços. Outras soluções possíveis são mecanimos de autonomicidade que podem utilizar a infraestrutura de interação e de contexto disponibilizada pelo *SysSU* para implementar políticas de dependabilidade em sistemas úbiquos, como tratamento de exceções sensíveis ao contexto.

Bibliografia

AGUILERA, M. K. et al. Matching events in a content-based subscription system. In: *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 1999. (PODC '99), p. 53–61.

ALONSO, G. et al. *Web Services: Concepts, Architectures and Applications*. Berlin: Springer, 2004. ISBN 3540440089.

ANDREWS, G. *Concurrent Programming: Principles and Practice*. Boston, MA, USA: The Benjamin/Cummings Publishing Company, 1991. ISBN 0849338336.

ASSOCIATES, S. C. *TCP Linda*. 2011. Disponível em: <http://www.lindaspaces.com/products/linda.html>.

BALDAUF, M.; DUSTDAR, S.; ROSENBERG, F. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, v. 2, n. 4, p. 263–277, 2007.

BALZAROTTI, D.; COSTA, P.; PICCO, G. The lights tuple space framework and its customization for context-aware applications. *Web Intelligence and Agent Systems*, IOS Press, v. 5, n. 2, p. 215–231, 2007.

BARESI, L.; NITTO, E. D.; GHEZZI, C. Towards open-world software: Issues and challenges. *Computer*, IEEE Computer Society, Los Alamitos, CA, USA, v. 39, p. 36–43, 2006.

BAZIRE, M.; BREZILLON, P. Understanding context before using it. In: DEY, A. et al. (Ed.). *Modeling and Using Context*. [S.l.]: Springer Berlin - Heidelberg, 2005, (Lecture Notes in Computer Science, v. 3554). p. 29–40.

BELLAVISTA, P.; CORRADI, A. *The Handbook of Mobile Middleware*. Boston, MA, USA: Auerbach Publications, 2006. ISBN 0849338336.

CABRI, G. et al. Uncoupling coordination: Tuple-based models for mobility. In: BELLAVISTA, P.; CONRRADI, A. (Ed.). *The Handbook of Mobile Middleware*. New Jersey, US: Prentice Hall, 2006. cap. 10, p. 229–256.

CABRI, G.; LEONARDI, L.; ZAMBONELLI, F. Xml dataspaces for mobile agent coordination. In: *ACM symposium on Applied computing*. New York, NY, USA: ACM, 2000. (SAC '00), p. 181–188. ISBN 1-58113-240-9.

CABRI, G.; LEONARDI, L.; ZAMBONELLI, F. Mobile agent coordination for distributed network management. *Network System Management*, Plenum Press, New York, NY, USA, v. 9, n. 4, p. 435–456, 2001.

CARRIERO, N.; GELERNTER, D. Linda in context. *Communications of ACM*, ACM, New York, NY, USA, v. 32, n. 4, p. 444–458, 1989.

CARZANIGA, A. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. Tese (PhD thesis) — Politecnico di Milano, Milan, Italy, dez. 1998.

CARZANIGA, A.; ROSENBLUM, D. S.; WOLF, A. L. Achieving scalability and expressiveness in an internet-scale event notification service. In: *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2000. (PODC '00), p. 219–227.

CHARLES, A.; MENEZES, R.; TOLKSDORF, R. On the implementation of swarmlinda. In: *42nd annual Southeast regional conference*. New York, NY, USA: ACM, 2004. (ACM-SE 42), p. 297–298.

CHEN, H.; FININ, T.; JOSHI, A. An ontology for context-aware pervasive computing environments. *The Knowledge Engineering Review*, Cambridge University Press, New York, NY, USA, v. 18, p. 197–207, September 2003.

CHETAN, S.; RANGANATHAN, A.; CAMPBELL, R. Towards fault tolerance pervasive computing. *IEEE Technology and Society Magazine*, v. 24, n. 1, p. 38–44, March 2005.

CLEMENTS, P. et al. *Documenting Software Architectures: Views and Beyond*. Boston, MA: Addison-Wesley, 2003. ISBN 978-0-201-70372-6.

COSTA, C. A. da; YAMIN, A. C.; GEYER, C. F. R. Toward a general software infrastructure for ubiquitous computing. *IEEE Pervasive Computing*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 7, n. 1, p. 64–73, 2008.

COULOURIS, G.; KINDBERG, T.; DOLLIMORE, J. *Distributed Systems: Concepts and Design*. 4th. ed. [S.l.]: Addison Wesley, 2005. ISBN 0321263545.

COUTAZ, J. et al. Context is key. *Communications of the ACM*, ACM, New York, NY, USA, v. 48, n. 3, p. 49–53, 2005.

CUGOLA, G.; JACOBSEN, H.-A. Using publish/subscribe middleware for mobile systems. *SIGMOBILE Mob. Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 6, p. 25–33, October 2002.

DAVIES, N. et al. Limbo: a tuple space based platform for adaptive mobile applications. In: *IFIP/IEEE international conference on Open distributed processing and distributed platforms*. London, UK, UK: Chapman & Hall, Ltd., 1997. p. 291–302.

DEVARAJU, A.; HOH, S.; HARTLEY, M. A context gathering framework for context-aware mobile solutions. In: . New York, New York, USA: ACM Press, 2007. v. 7, p. 39–46.

DEY, A. K. Understanding and using context. *Personal Ubiquitous Computing*, Springer-Verlag, London, UK, v. 5, n. 1, p. 4–7, 2001.

EDWARDS, W. Discovery systems in ubiquitous computing. *IEEE Pervasive Computing*, IEEE, v. 5, n. 2, p. 70–77, 2006.

ENDRES, C.; BUTZ, A.; MACWILLIAMS, A. A survey of software infrastructures and frameworks for ubiquitous computing. *Mobile Information Systems*, IOS Press, Amsterdam, The Netherlands, The Netherlands, v. 1, n. 1, p. 41–80, 2005.

ERFIANTO, B.; MAHMOOD, A. K.; RAHMAN, A. S. A. Modeling context and exchange format for context-aware computing. In: *Research and Development, 2007. SCOReD 2007. 5th Student Conference on*. [S.l.: s.n.], 2007. p. 1–5.

ERL, T. *Service-Oriented Architecture : Concepts, Technology, and Design*. [S.l.]: Prentice Hall PTR, 2005. ISBN 0131858580.

EUGSTER, P. T. et al. The many faces of publish/subscribe. *ACM Computing Surveys*, v. 35, n. 2, p. 114–131, jun. 2003.

EUGSTER, P. T.; GARBINATO, B.; HOLZER, A. Location-based publish/subscribe. In: *Proceedings of the Fourth IEEE International Symposium on Network Computing and Applications*. Washington, DC, USA: IEEE Computer Society, 2005. p. 279–282.

FREEMAN, E.; ARNOLD, K.; HUPFER, S. *JavaSpaces Principles, Patterns, and Practice*. 1st. ed. Essex, UK, UK: Addison-Wesley Longman Ltd., 1999.

GARLAN, D. et al. Project aura: Toward distraction-free pervasive computing. *IEEE Pervasive Computing*, v. 1, n. 2, p. 22–31, April–June 2002.

GRIMM, R. et al. System support for pervasive applications. *ACM Transactions on Computer Systems*, v. 22, n. 4, p. 421–486, November 2004.

HANDOREAN, R.; ROMAN, G.-c. Service provision in ad hoc networks. *Coordination Models and Languages*, Springer Berlin / Heidelberg, v. 2315, p. 299–307, 2007.

HANSMANN, U. et al. *Pervasive Computing : The Mobile World*. [S.l.]: Springer, 2003. ISBN 3540002189.

HUEBSCHER, M. C.; MCCANN, J. A. A survey of autonomic computing–degrees, models, and applications. *ACM Computing Surveys*, ACM, New York, NY, USA, v. 40, n. 3, p. 1–28, 2008.

JAROUCHEH, Z.; LIU, X.; SMITH, S. A perspective on middleware-oriented context-aware pervasive systems. *Computer Software and Applications Conference, Annual International*, IEEE Computer Society, Los Alamitos, CA, USA, v. 2, p. 249–254, 2009.

JULIEN, C.; ROMAN, G.-C. Egospaces: Facilitating rapid development of context-aware mobile applications. *IEEE Transactions on Software Engineering*, Citeseer, v. 32, n. 5, p. 281–298, 2006.

KIELMANN, T. Object-oriented distributed programming with objective linda. In: *In First International Workshop on High Speed Networks and Open Distributed Platforms*. [S.l.: s.n.], 1995.

KIM, E.; HELAL, S.; COOK, D. Human activity recognition and pattern discovery. *IEEE Pervasive Computing*, v. 9, n. 1, p. 48–53, 2010.

KINDBERG, T.; FOX, A. System software for ubiquitous computing. *IEEE Pervasive Computing*, v. 1, n. 1, p. 70–81, jan. 2002.

KON, F. et al. A flexible, interoperable framework for active spaces. In: *OOPSLA'workshop on Pervasive Computing*. Minneapolis, MN: [s.n.], 2000.

KON, F. et al. 2k: A dynamic, component-based operating system for rapidly changing environments. In: *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*. [S.l.: s.n.], 1998. p. 388–390.

KRUMM, J. *Ubiquitous Computing Fundamentals*. 1st. ed. [S.l.]: Chapman & Hall/CRC, 2009. ISBN 9781420093605.

LEHMAN, T. J. et al. Hitting the distributed computing sweet spot with tspaces. *Computing Network*, Elsevier North-Holland, Inc., New York, NY, USA, v. 35, n. 4, p. 457–472, 2001.

LIU, C.; PENG, Y.; CHEN, J. Web services description ontology-based service discovery model. In: *IEEE/WIC/ACM International Conference on Web Intelligence*. Washington, DC, USA: IEEE Computer Society, 2006. p. 633–636. ISBN 0-7695-2747-7.

LTD, G. T. *Space-Based Architecture and The End of Tier-based Computing*. 2011. Disponível em: <http://www.gigaspaces.com>.

MAIA, M. E. F. *AESPmob – Modelo Autônomo e Evolutivo para Provisâo de Serviços Essenciais em Redes Móveis*. Tese (Dissertçao de Mestrado) — Universidade Federal do Ceará, Fortaleza, Ceará, Brasil, 2009.

MAIA, M. E. F.; ROCHA, L. S.; ANDRADE, R. M. C. Requirements and challenges for building service-oriented pervasive middleware. In: *Proceedings of the 2009 international conference on Pervasive services*. New York, NY, USA: ACM, 2009. p. 93–102.

MAMEI, M.; ZAMBONELLI, F. Programming pervasive and mobile computing applications: The tota approach. *ACM Transactions on Software Engineering and Methodology*, ACM Press, New York, NY, USA, v. 18, n. 4, p. 1–56, 2009.

MARINHO, F. G. et al. An architecture proposal for nested software product lines in the domain of mobile and context-aware applications. *Software Components, Architectures and Reuse, Brazilian Symposium on*, IEEE Computer Society, v. 0, p. 51–60, 2010.

MARINHO, F. G. et al. A software product line for the mobile and context-aware applications domain. In: BOSCH, J.; LEE, J. (Ed.). *Software Product Lines: Going Beyond*. [S.l.: s.n.], 2010, (Lecture Notes in Computer Science, v. 6287). p. 346–360.

MüHL, G.; FIEGE, L.; PIETZUCH, P. *Distributed Event-Based Systems*. 1st. ed. London, UK: Springer, 2006. ISBN 9783540326519.

MORAN, T. P.; DOURISH, P. Introduction to this special issue on context-aware computing. *Human-Computer Interaction*, L. Erlbaum Associates Inc., Hillsdale, NJ, USA, v. 16, n. 2, p. 87–95, 2001.

MURPHY, A. L.; PICCO, G. P.; ROMAN, G.-C. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology*, ACM Press, San Antonio, TX, USA, v. 15, n. 3, p. 279–328, 2006.

NICOLA, R. D.; FERRARI, G.; PLUGLIESE, R. Klaim: A kernel language for agents interactions and mobility. *IEEE Transactions on Software Engineering*, Citeseer, v. 24, n. 5, p. 315–330, 1998.

NIEMELä, E.; LATVAKOSKI, J. Survey of requirements and solutions for ubiquitous software. In: *3rd international conference on Mobile and ubiquitous multimedia*. New York, NY, USA: ACM Press, 2004. p. 71–78.

NIEUWDORP, E. The pervasive discourse: an analysis. *Comput. Entertain.*, ACM, New York, NY, USA, v. 5, April 2007. ISSN 1544-3574.

NOBLE, B. D.; PRICE, M.; SATYANARAYANAN, M. A programming interface for application-aware adaptation in mobile computing. In: *2nd Symposium on Mobile and Location-Independent Computing*. [S.l.]: Computing Systems, 1995. p. 57–66.

NOBLE, B. D.; SATYANARAYANAN, M. Experience with adaptive mobile applications in odyssey. *Mobile Networks and Applications*, Kluwer Academic Publishers, Hingham, v. 4, n. 4, p. 245–254, 1999. ISSN 1383-469X.

OMG. *CORBA*. 2011. Disponível em: <http://www.corba.org/>.

OMICINI, A.; ZAMBONELLI, F. Tuple centres for the coordination of internet agents. In: *ACM symposium on Applied computing*. New York, NY, USA: ACM, 1999. (SAC '99), p. 183–190.

ORACLE. *Java Message Service (JMS)*. 2011. Disponível em: <http://www.oracle.com/technetwork/java/jms-136181.html>.

PARZYJEGLA, H. et al. Design and implementation of the rebeca publish/subscribe middleware. In: SACHS, K.; PETROV, I.; GUERRERO, P. (Ed.). *From Active Data Management to Event-Based Systems and More*. [S.l.]: Springer Berlin / Heidelberg, 2010, (Lecture Notes in Computer Science, v. 6462). p. 124–140.

PEREIRA, F. M. Q. et al. Arcademis: a framework for object-oriented communication middleware development. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 36, p. 495–512, April 2006.

QIAN, K.; LIU, J.; TAO, L. Asynchronous callback in web services. *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, International Conference on and Self-Assembling Wireless Networks, International Workshop on*, IEEE Computer Society, Los Alamitos, CA, USA, p. 375–380, 2006.

RANGANATHAN, A.; AL-MUHTADI, J.; CAMPBELL, R. Reasoning about uncertain contexts in pervasive computing environments. *IEEE Pervasive Computing*, v. 3, n. 2, p. 62–70, abr. 2004.

RANGANATHAN, A.; CAMPBELL, R. H. An infrastructure for context-awareness based on first order logic. *Personal and Ubiquitous Computing*, v. 7, n. 6, p. 353–364, dez. 2003.

ROMAN, M. et al. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, v. 1, n. 4, p. 74–83, out. 2002.

SAHA, D.; MUKHERJEE, A. Pervasive computing: A paradigm for the 21st century. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 36, p. 25–31, March 2003. ISSN 0018-9162.

SALBER, D.; DEY, A. K.; ABOWD, G. D. The context toolkit: aiding the development of context-enabled applications. In: *SIGCHI conference on Human factors in computing systems*. New York, NY, USA: ACM Press, 1999. p. 434–441.

SATYANARAYANAN, M. Pervasive computing: vision and challenges. *IEEE Personal Communications*, IEEE Communications Society, v. 8, n. 4, p. 10–17, 2001.

SATYANARAYANAN, M. et al. Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, IEEE Computer Society, Washington, v. 39, n. 4, p. 447–459, abr. 1990.

SCHELFTHOUT, K.; HOLVOET, T. Coordination middleware for decentralized applications in dynamic networks. In: *2nd international doctoral symposium on Middleware*. San Antonio, TX, USA: ACM Press, 2005. p. 1–5.

SCHOLTZ, J.; CONSOLVO, S. Toward a framework for evaluating ubiquitous computing applications. *IEEE Pervasive Computing*, v. 3, n. 2, p. 82–88, April 2004.

SPíNOLA, R. O.; MASSOLLAR, J.; TRAVASSOS, G. H. Checklist to characterize ubiquitous software projects. In: *XXI Simpósio Brasileiro de Engenharia de Software*. Joao Pessoa, PB, Brasil: [s.n.], 2007.

STANFORD, V. Toward a framework for evaluating ubiquitous computing applications. *IEEE Pervasive Computing*, v. 3, n. 1, p. 99–103, jan. 2004.

STERRITT, R. et al. A concise introduction to autonomic computing. *Advanced Engineering Informatics*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 19, n. 3, p. 181–187, 2005.

TANENBAUM, A. S.; STEEN, M. V. *Distributed Systems: Principles and Paradigmas*. 2th. ed. [S.l.]: Person Education Inc., 2007. ISBN 9788576051428.

THANH, D. v.; JORSTAD, I. A service-oriented architecture framework for mobile services. In: *Advanced Industrial Conference on Telecommunications/Service Assurance with Partial and Intermittent Resources Conference/E-Learning on Telecommunications Workshop*. Washington, DC, USA: IEEE Computer Society, 2005. p. 65–70.

WANT, R.; PERING, T. System challenges for ubiquitous & pervasive computing. In: *27th international conference on Software engineering*. New York, NY, USA: ACM Press, 2005. p. 9–14.

WEISER, M. The computer for the twenty-first century. *Scientific American*, v. 265, n. 3, p. 94–104, 1991.

ZHU, F.; MUTKA, M.; NI, L. Service discovery in pervasive computing environments. *IEEE Pervasive Computing*, IEEE, v. 4, n. 4, p. 81–90, 2005.