Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Departamento de Computação

Programa de Pós-Graduação em Ciência da Computação

Middleware de Serviços Multi-Camadas para Redes de Sensores Sem Fio

ALUNO: JOSÉ EDUARDO RIBEIRO

ORIENTADORA: REGINA BORGES DE ARAÚJO

SÃO CARLOS - SP

Março - 2010

Middleware de Serviços Multi-Camadas para Redes de Sensores Sem Fio

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Departamento de Computação

Programa de Pós-Graduação em Ciência da Computação

Middleware de Serviços Multi-Camadas para Redes de Sensores Sem Fio

José Eduardo Ribeiro

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação, do Departamento de Computação, da Universidade Federal de São Carlos, como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.
Área de concentração: Processamento de Imagens e Sinais - PIS.

SÃO CARLOS - SP

Março – 2010

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

R484ms

Ribeiro, José Eduardo.
Middleware de serviços multi-camadas para redes de sensores sem fio / José Eduardo Ribeiro. -- São Carlos : UFSCar, 2010.
141 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2010.

1. Sistemas distribuídos. 2. Computação pervasiva. 3. Serviços da web. 4. Redes de sensores sem fio. I. Título.

CDD: 005.43 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"Middleware de Serviços Multi-Camadas para Redes de Sensores Sem Fio"

JOSÉ EDUARDO RIBEIRO

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação

Membros da Banca:

Profa. Dra. Regina Borges de Araujo
(Orientadora - DC/UFSCar)

Prof. Dr. Jander Moreira
(DC/UFSCar)

Prof. Dr. Rogério Eduardo Garcia
(DMEC/UNESP – Presidente Prudente)

São Carlos
Março/2010

DEDICO ESTE TRABALHO:

À minha querida mãe, Izilda, pelo amor, incentivo e dedicação, sempre acreditando nos meus sonhos.

Ao meu pai, José do Carmo, certamente orgulhoso por mais um importante passo na minha vida.

À minha querida esposa, Aline, por toda ajuda, e compreensão para a realização deste trabalho.

Agradecimentos

A Deus e a minha mãe rainha, pela força em atravessar os obstáculos desta vida e por iluminar meus caminhos e revigorar minha busca diária rumo aos meus objetivos e evolução espiritual.

Aos meus pais e familiares pelo apoio, e incentivo ao meu esforço e capacidade.

À Profª. Drª. Regina Borges de Araújo, pela motivação, encorajamento, críticas, amizade e orientação durante os meus estudos na pós-graduação.

A todos os meus colegas do Laboratório de Redes Sem Fio e Simulação Interativa Distribuída (antigo LRNVet), principalmente Rafaela, Leandro, Rodolfo, Igor, Max, Állan, Gislaine, Gil, Felipe e Marcelo pelas discussões que contribuíram para a realização de nossos trabalhos.

A minha irmã, Maria Cristina, e cunhado, Edinilson, a minha filha Amanda, e minhas sobrinhas Gabrieli e Julia, pelos momentos de alegria que proporcionavam quando eu estava triste.

Aos meus amigos e amigas, em especial à Eliana, Fernando e Ricardo que sempre estão ao meu lado dando apoio quando preciso.

Aos professores do Departamento de Computação da UFSCar, principalmente aos que tive a oportunidade de conhecer durante os estudos.

E principalmente à minha esposa, Aline, por todo o auxílio, carinho e incentivo e compreensão para a elaboração deste trabalho.

Resumo

Evoluções tecnológicas nos sistemas de microeletrônicos e na comunicação sem fio permitiram o desenvolvimento de dispositivos chamados nós sensores, que são pequenos, de baixo custo e de baixo consumo de energia. Os nós sensores integram módulos de sensoriamento, processamento de dados e de comunicação sem fio. A utilização dos nós sensores de forma distribuída possibilita a comunicação entre eles proporcionando a formação das Redes de Sensores Sem Fios (RSSFs). RSSFs estão sendo utilizadas cada vez mais como poderosas ferramentas de monitoramento, de aplicações de monitoramento ambiental a monitoramento de situações de emergência em ambientes sujeitos a situações de risco à vida e ao patrimônio, tais como incêndios, vazamento de gases tóxicos e explosões. Sistemas de gerenciamento da emergência que integram redes de sensores sem fio são utilizados como apoio à tomada de decisão para equipes de resposta a emergências em que o tempo-resposta nessas condições se torna fator preponderante visando o sucesso de operações práticas de salvamento. Para que o sistema de gerenciamento de emergência seja eficaz é necessária uma camada de suporte para prover integração e reusabilidade dos serviços fornecidos pelo sistema, além de uma solução para abstrair toda a complexidade de comunicação e processamento de dados no interior da RSSF. Neste trabalho, um *middleware* para RSSFs foi projetado e parcialmente implementado, para auxiliar o trabalho dos desenvolvedores de aplicações. As principais características deste *middleware* são fornecer um mecanismo expressivo e flexível para subscrições de interesse das aplicações na RSSFs e interfaces reusáveis que utilizam tecnologias padrão da Web para prover interoperabilidade entre os serviços e aplicações. Uma avaliação inicial foi realizada com um protótipo de algumas funções do *middleware*. Uma estimativa de ocupação de memória nos nós sensores foi feita com base em estudos reportados na literatura, e mostra que o *middleware* projetado é viável para a plataforma de hardware dos motes Mica 2. O mote Mica 2 é a terceira geração de nós sensores comerciais da família Mica motes, usado para capacitar baixa energia em RSSFs.

Palavras-Chave: Middleware, Serviços Web, Publish/Subscribe, RSSFs.

Abstract

Technological evolutions in the microelectronic systems and in the wireless communication allowed the development of devices called sensor nodes, which are smalles have low cost and low energy consumption. The sensor nodes integrate sensing modules, data processing and of wireless communication. The use of sensor nodes in a distributed way makes possible the communication among them providing the formation of the Wireless Sensor Networks (WSN). WSN are being used more and more as powerful monitoring tools of applications of environmental monitoring the monitoring of emergency situations in environments subject to risk situations to the life and the patrimony such as fire, leaks of poisonous gasses and explosions. Emergency management systems that integrate networks of wireless sensors have been used as support to the making of decision for response teams to the emergencies where the response time in these conditions becomes preponderant factor, aiming at the success of rescue practical operations. In order to the system of emergency management to be efficient a support layer to provide integration and reusability of the services is necessary supplied by the system besides a solution to abstract all the communication complexity and data processing inside WSN. In this work a middleware for WSN was designed and partially implemented to aid the work of the developers applications. The main characteristics of this middleware are to supply an expressive and flexible mechanism for subscriptions of interest of the application in the WSN and reusable interfaces that use technologies standard of the Web to provide interoperability between the services and applications. An initial evaluation accomplished with a prototype of some functions of the middleware. An estimate of occupation of memory in the sensor nodes was made with base in studies reported in the literature, and display that middleware projected is viable for the platform of the hardware of motes Mica 2. The mote Mica 2 is the third generation of commercial sensor nodes of the mica family motes, used to enable low energy in WSN.

Key-Words: Middleware, Web Services, Publish/Subscribe, WSN.

Lista de Figuras

Índice de Tabelas

Lista de Abreviaturas e Siglas

CDMA	Code Division Multiple Access
COM	Componente Object Model
CORBA	Commom Object Request Broker Architecture
DAARP	Dynamic Data-Aggregation Aware Routing Protocol
DAC	Digital to Analog Converter
DCE RPC	Distributed Computing Environment Remote Procedure Call
DCOM	Distributed COM
DSM	Distributed Shared Memory
FTP	File Transfer Protocol
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
IDL	Interface Definition Language
JAX-WS	Java API for XML Web Services
JVM	Java Virtual Machine
J2SE	Java Standard Edition
LOS	Line of Sight
MILAN	Middleware Linking Applications and Network
MOM	Message Oriented Middleware
OMG	Object Management Group
ORB	Object Request Broker
ORPC	Object Remote Procedure Call
PCs	Personal Communications
PDAs	Personal Digital Assistants
POJO	Plain Old Java Objects
QoS	Qualidade de Serviço
RFID	Radio Frequency IDentification
RMI	Remote Method Invocation
RSSFs	Redes de Sensores Sem Fios
RSASFs	Redes de Sensores e Atuadores Sem Fios
RPC	Remote Procedure Call
SDs	Sistemas Distribuídos
SINA	Sensor Information Networking Architecture
SMTP	Simple Mail Transfer Protocol
SOA	Service Oriented Architecture

SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SQTL	Sensor Query and Tasking Language
TDMA	Time Division Multiple Access
TinyOS	Tiny Microthreading Operating System
UDDI	Universal Description Discovery and Integration
URI	Unique Resource Identifier
WSDL	Web Services Description Language
W3C	World Wide Web Consortium
XML	eXtensible Markup Language
XSD	XML Schema Definition

Sumário

1. Introdução

O monitoramento de ambientes reais sujeitos a situações de emergências é importante para equipes de resgate e bombeiros, para saber o que está acontecendo em situações de risco e para capacitar à tomada de melhores decisões em atividades de salvamento e resgate. Com o recente desenvolvimento na área de comunicação sem fio e sensores multifuncionais com capacidade de comunicação e processamento, as Redes de Sensores Sem Fios (RSSFs), são usadas cada vez mais como poderosas ferramentas de monitoramento em ambientes sujeitos a situações críticas de risco à vida e ao patrimônio.

Uma RSSF consiste de dispositivos autônomos distribuídos que cooperativamente monitoram condições ambientais ou físicas, tais como temperatura, som, vibração, pressão, movimento etc. em diferentes localizações. RSSFs são usadas em muitas aplicações, tais como monitoramento ambiental, estrutura, equipamentos, comunicações e muitas outras aplicações que podem ser críticas como salvamento de vidas e preservação de patrimônio. Nós sensores são dispositivos com restrição de energia e o consumo de energia é geralmente associado com a quantidade de dados obtidos, já que com a comunicação se gasta mais energia. Por essa razão, protocolos, algoritmos e soluções projetados para RSSFs devem considerar o consumo de energia. RSSFs são redes centradas em dados que usualmente produzem uma grande quantidade de informações que são roteadas, frequentemente em uma forma *multi-hop*, para um nó *sink*, ou nó sorvedouro, que funciona como um *gateway* para a aplicação.

As RSSFs são consideradas ambientes de computação distribuída com restrições de processamento, armazenamento, energia e largura de banda. Os nós sensores devem consumir pouca energia e prover serviços comuns e especializados que facilitem o desenvolvimento de aplicações [HAN 05]. Com o avanço em pesquisas, sistemas de *middleware* tentam suportar uma vasta classe de aplicações e resolver alguns dos desafios que surgiram com as RSSFs tais como, gerenciamento dos interesses das aplicações para a rede, gerenciamento de dados (coleta, entrega), gerenciamento de recursos etc.

Middleware é um artefato em software que reside entre a aplicação e o sistema operacional, que fornece por meio de interfaces o reuso de serviços, facilitando o trabalho em desenvolver aplicações mais eficientes para ambiente distribuído, tal como uma RSSF [BLA 04].

Sistemas de *middleware* tradicionais não podem ser empregados diretamente na RSSF, pois geralmente consomem demasiadamente recursos como processador, memória e largura de banda. Porém em RSSF esses recursos são escassos. O principal propósito de *middleware* para RSSF é suportar o desenvolvimento, manutenção, distribuição e execução de aplicações. Isso inclui mecanismos de formulação complexa de tarefas de sensoriamento de alto nível, comunicação dessas tarefas com a RSSF, coordenação de nós sensores para distribuição das tarefas, agregação e ou fusão de dados para unir as leituras dos sensores em um resultado de alto nível e reportar os resultados das tarefas para a aplicação solicitante.

Um *middleware* para RSSF deve ser projetado também para atender outro propósito: fornecer os serviços do *middleware* acessíveis a aplicações do "mundo externo" (por exemplo, por meio da Internet). Os Serviços Web podem auxiliar na interoperabilidade, reusabilidade e a flexibilidade entre aplicações e os serviços de diferentes redes de forma padronizada e eficiente.

Neste trabalho foi projetado um *middleware* que provê mecanismos que abstraiam toda a complexidade para a aplicação em solicitar tarefas e monitorar eventos em RSSFs. O *middleware* foi projetado por meio do paradigma *publish/subscribe* baseado em conteúdo. Este esquema fornece mais dinâmica, flexibilidade e expressividade nas subscrições de interesse das aplicações nas RSSF. Além disso, o *middleware* em mais alto nível provê os serviços desenvolvidos acessíveis via Internet, por meio dos Serviços Web. Apesar de o *middleware* ter sido construído para suportar a classe de aplicações no domínio de emergência, diferentes classes de aplicações podem também utilizá-lo, devido à flexibilidade e reusabilidade introduzida por essas tecnologias.

1.1 Motivação e Objetivos do Trabalho

Este projeto está inserido em um projeto maior em desenvolvimento no Laboratório de Redes Sem Fio e Simulação Interativa Distribuída do Departamento de Computação da UFSCar, que visa à pesquisa de métodos, técnicas e soluções para a classe de aplicações de gerenciamento de emergências em ambientes físicos cientes de contexto.

A principal motivação é construir uma arquitetura de suporte de integração e reuso dos serviços desenvolvidos, além de fornecer os serviços básicos das redes de sensores às aplicações clientes.

O objetivo deste trabalho é especificar e projetar um *middleware* para RSSFs que forneça suporte aos serviços oferecidos, tal como, interpretação de contextos, por meio de interfaces *web* para as aplicações. Além disso, deve abstrair toda a complexidade em solicitar tarefas e monitorar eventos no interior das RSSFs. O projeto do *middleware* para RSSFs foi estabelecido em dois níveis: nível 1 consiste de serviços reusáveis em mais alto nível; nível 0, suportar toda a comunicação entre os nós sensores e a coordenação das subscrições de interesses das aplicações em RSSFs.

O *middleware* para RSSFs projetado neste trabalho abstrai a lacuna entre as redes de sensores e as aplicações. Para isto foram utilizados as tecnologias dos Serviços Web, o paradigma *publish/subscribe* baseado em conteúdo e o sistema operacional TinyOS[1]. O *middleware* fornece os dados agregados na rede a serviços reusáveis em mais alto nível e a diversas classes de aplicações, em especial para a classe de aplicações de gerenciamento de emergência.

1.2 Organização da Dissertação

A dissertação está organizada da seguinte maneira: o Capítulo 2 apresenta os fundamentos de RSSFs, definição, características, aplicações, desafios e funcionamento. O Capítulo 3 descreve os fundamentos de *middlewares*. Inicialmente, será realizado um estudo minucioso sobre os tipos de *middleware* para sistemas distribuídos e, posteriormente, serão listadas as características, classificações e os projetos mais relevantes de *middleware* para as RSSFs. O Capítulo 4 apresenta os paradigmas de comunicação em sistemas distribuídos. O Capítulo 5 apresenta a proposta do projeto e avaliação do *middleware*, seguido de Conclusões e Referências Bibliográficas.

[1] **TinyOS:** sistema operacional de código aberto projetado especificamente para as RSSFs.

2. Redes de Sensores Sem Fios

2.1 Considerações Iniciais

A partir do grande avanço tecnológico que ocorreu na última década, nas áreas de sensores, circuitos integrados e comunicação sem fio, surgiram as chamadas redes de sensores sem fios (RSSFs). Aplicações típicas dessas redes podem ser aplicadas em monitoramento, rastreamento, coordenação e processamento em diferentes contextos. As RSSFs podem ser utilizadas como ferramenta poderosa em aplicações de preparação e respostas a situações de emergência.

Neste capítulo são realizadas revisões de conceitos, áreas de aplicações, funcionamento, características e classificações das RSSFs. São também descritos os conceitos de redes de sensores e atuadores sem fios (RSASFs), seguidas das considerações finais.

2.2 Redes de Sensores Sem Fios

Uma rede de sensores sem fio é composta por um grande número de pequenos nós sensores que são colocados dentro ou muito próximos do fenômeno a ser analisado [AKY 02]. Cada nó sensor pode ser equipado com vários tipos de sensores como, por exemplo, sensores de temperatura, de pressão, sísmico, acústico, de radiação ou infravermelho. Sensores são construídos de hardware de relativo baixo custo e possui limitações de memória, processamento, comunicação e energia. A Figura 2.1 mostra alguns tipos de sensores.

As RSSFs possibilitam a coleta de dados de forma distribuída em ambientes onde o uso de fios ou cabeamento não seja possível ou viável. A instalação pode ser realizada, por exemplo, em prédios ou pontes, no interior de máquinas e tubulações, em ambientes residenciais, áreas de preservação permanente e de importância ambiental, palco de operações em ambientes de riscos, culturas vegetativas, áreas de monitoramento geográfico, estratégias militares ou ainda em áreas da nanotecnologia [AKY 02].

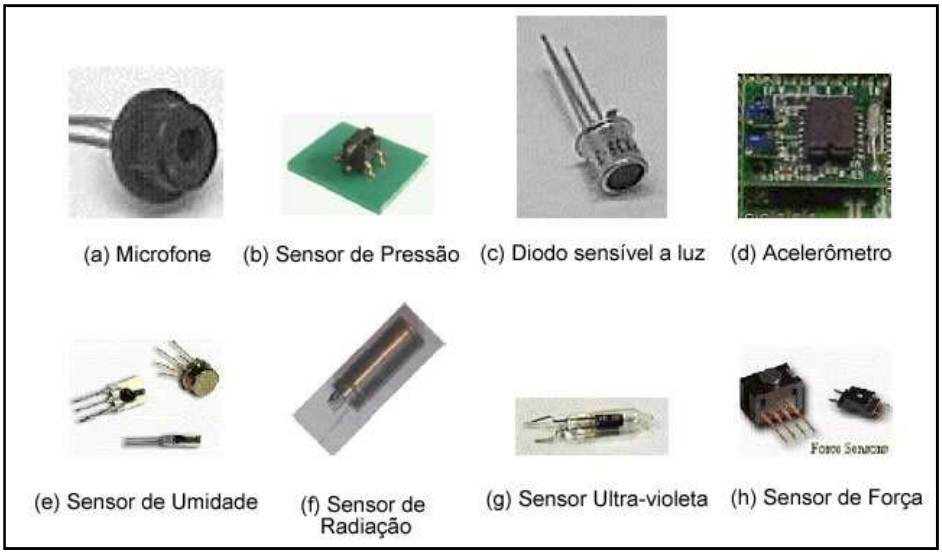

Figura 2.1 - Tipos de sensores.

A utilização de RSSFs em diversas áreas justifica-se, pelo baixo custo dos nós sensores e o potencial que proporciona com monitoramento mais preciso. As redes de sensores podem ser empregadas nas seguintes áreas:

- Meio-ambiente: Rastreamento do movimento dos pássaros e pequenos animais; monitoração de condições ambientais que afetam colheitas e plantio (por exemplo: combate à geada, detecção de componentes químicos ou biológicos e irrigação); mapeamento da bio-complexidade ambiental, estudo da poluição e muitas outras [CER 01] [MAI 02] [KAH 99] [AGR 00]. Saúde: Controle de doenças contagiosas; interface para deficientes; monitoramento de pacientes; diagnóstico de distúrbios; administração de drogas em hospitais incluindo monitoração e localização de pacientes e médicos [END 06] [BUL 01] [WAR 01]. Aplicações Militares: Monitoramento de tropas; reconhecimento de terreno; detecção de alvos e ataques biológicos, químicos ou nucleares. Aplicações Comerciais: Automação de vendas e processo industriais; manutenção de inventário, monitoração de qualidade de produtos; detecção e vigilância de veículos e estabelecimentos [AGR 00] [CHO 01] [EST 99] [WAR 01].
- Monitoração de Estrutura/Equipamentos: identificação de falhas em estruturas (pontes, prédios, etc.); fadiga de máquinas e equipamentos (motores, dutos de gás, etc.); diagnósticos de máquinas [KIN 00].

Conforme referenciado em [RUI 04b], a tendência é a produção dos nós sensores em larga escala, barateando o seu custo, e o investimento no desenvolvimento tecnológico levando a novas melhorias, como aumento de processamento e armazenamento

nos nós, além de redução do tamanho dos nós sensores. Portanto, novas aplicações podem surgir aumentando a abrangência de uso das RSSFs.

A posição de cada nó sensor não precisa ser necessariamente pré-estabelecida de forma determinística, os nós sensores podem ser posicionados também de forma aleatória, ou seja, não determinística, em locais de difícil acesso, como áreas de desastres e incêndios, conforme mostra a Figura 2.2. Por outro lado, significa afirmar que algoritmos e protocolos para redes de sensores devem possuir características de auto-organização em relação aos nós. [AKY 02].

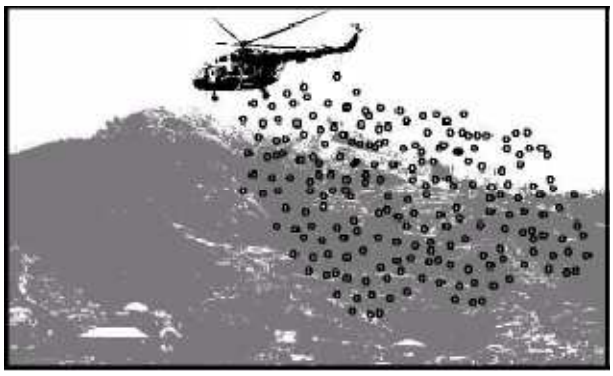

Figura 2.2 - Nós sensores espalhados em um campo de interesse [PIN 04].

Uma RSSF além de possuir nós sensores, possui, também, um ou mais nós de escoamento de dados, chamados de sorvedouros (*sink*) ou estação base. O *sink* é o elemento por meio do qual a rede se comunica com outras redes ou com um ou mais observadores [RUI 04a]. O *sink* proporciona a interface entre a aplicação e a rede, servindo de ponto de entrada para a submissão dos interesses da aplicação, além disso, é um concentrador dos dados coletados e enviados pelos nós sensores. Os *sinks* possuem maior poder computacional e menor restrição energética.

2.3 Componentes dos Nós Sensores

Nós sensores são um conjunto de dispositivos compactos e autônomos, equipados com capacidade de processamento e comunicação. Os principais componentes de um nó sensor são: unidade de comunicação sem fio, unidade de energia, unidade de sensoriamento e unidade de computação. Possuem também dependendo da aplicação

destinada, unidades adicionais tais como: unidade de localização, gerador de energia e movimentação. A Figura 2.3 mostra tais unidades presentes nos nós sensores.

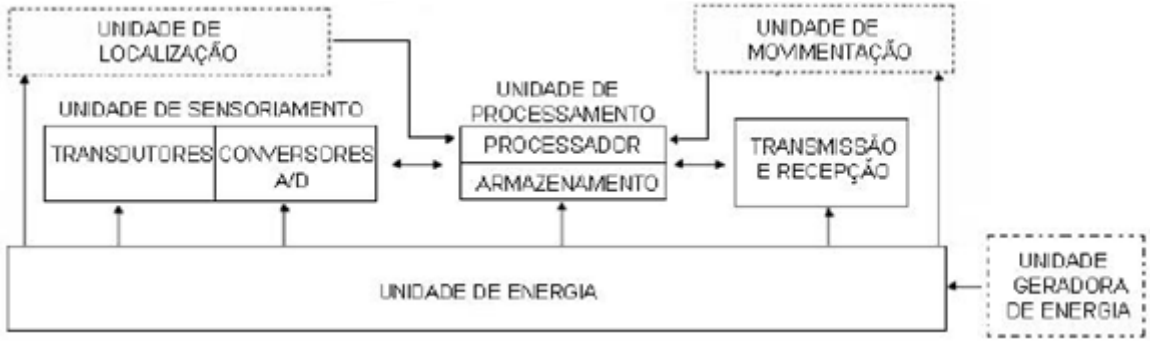

Figura 2.3 - Componentes do nó sensor sem fio. (Adaptado de [AKY 02])

Os transdutores e os conversores A/D são as subunidades que compõem a unidade de sensoriamento. Os dados coletados são passados para a unidade de processamento. Esta, por sua vez, é associada a uma pequena unidade de armazenamento de dados e pela subunidade processador, responsável pelos procedimentos que fazem os sensores colaborarem entre si para realizar a tarefa de sensoriamento. A unidade de energia é composta por células de energia e possivelmente uma unidade de geração de energia associada. A unidade de transmissão e recepção é responsável pela comunicação na camada física da rede.

Unidades adicionais podem ser instaladas a um nó sensor sem fio, tais como a unidade de localização que pode ser útil quando se quer saber a localização do nó com maior precisão.

Em alguns casos uma RSSF também pode ser composta de dispositivos atuadores que permitem ao sistema controlar parâmetros do ambiente monitorado. Esforços na área de dispositivos eletrônicos estão sendo feitos para solucionar as restrições de recursos presentes nos hardwares dos nós sensores sem fio. Possivelmente, em um futuro próximo esses dispositivos possuirão capacidades maiores de processamento, memória e principalmente energia.

2.4 Funcionamento de RSSF

O funcionamento da RSSF consiste, inicialmente, do lançamento dos nós sensores de forma aleatória ou colocados individualmente na área de interesse. Antes de iniciar as atividades de sensoriamento, os nós sensores são ativados e estabelecem os

caminhos que serão usados para o tráfego de dados [LOU 03]. A Figura 2.4 mostra o estabelecimento de uma rede de sensores.

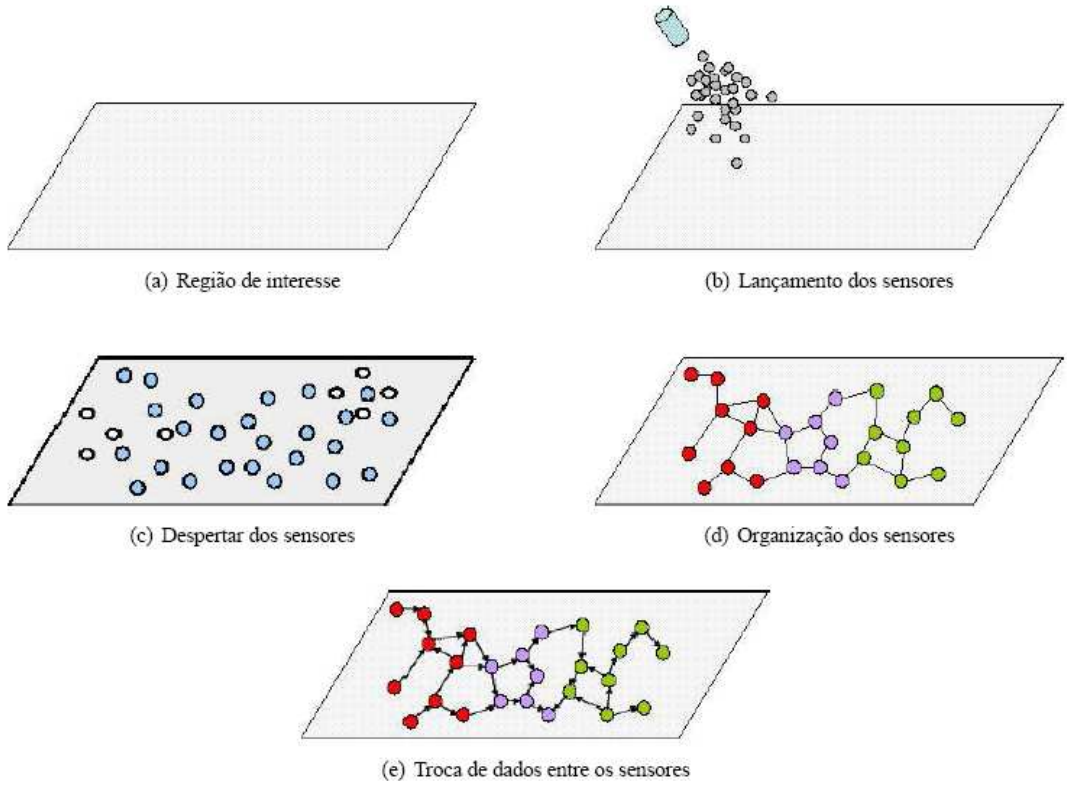

Figura 2.4 - Estabelecimento de Redes de Sensores [LOU 03].

O nó *sink* difunde as tarefas de sensoriamento na rede e aguarda as respostas. Quando um evento de interesse é detectado, os nós coletam os dados e, após algum processamento, enviam-os ao nó *sink* pelo caminho definido pelo protocolo de roteamento, como mostra a Figura 2.5. A dinâmica da rede, provocada pelo movimento ou pela falha dos nós, exige uma manutenção periódica da rota a ser usada.

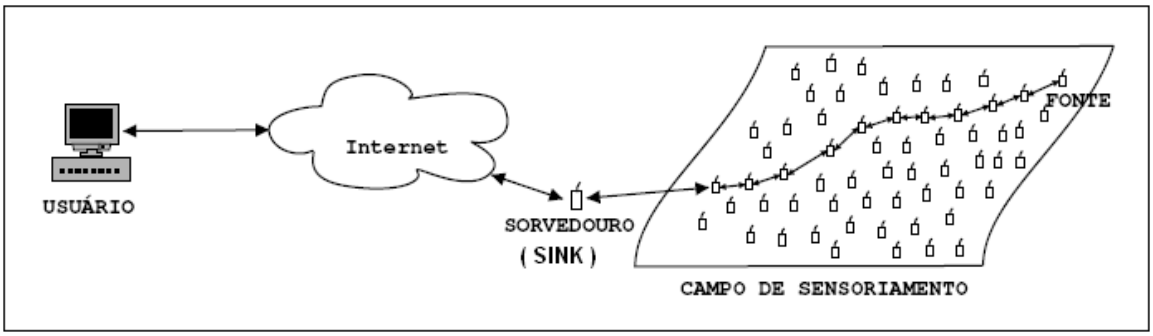

Figura 2.5 - Arquitetura de Redes de Sensores [PIN 04].

2.5 Características de RSSF

As RSSFs possuem um conjunto de características que as diferenciam de outros tipos de sistemas distribuídos. Isto se deve principalmente às áreas em que são aplicados, o que faz com que questões específicas tenham que ser resolvidas. Essas questões incluem: consumo de energia, tolerância a falhas, limitações de hardware, expansibilidade, agregação de dados, ambiente de operação, topologia da rede, meio de transmissão e Qualidade de Serviços.

2.5.1 Consumo de Energia

O consumo energético é um dos fatores mais relevantes em uma RSSF, visto que o nó sensor é um dispositivo microeletrônico com limitada fonte de energia. Além disso, na maioria das aplicações, pode ser impossível ou inviável substituir a fonte energética. Dessa forma, as capacidades de sensoriamento, processamento e comunicação do nó sensor são limitadas pela disponibilidade de energia [AKY 02].

O tempo de vida de um nó é altamente dependente do tempo de vida de sua bateria. O mau funcionamento de alguns nós pode acarretar mudanças na topologia da rede e, em consquência, a necessidade de reorganização de rotas para entrega de dados.

A otimização do consumo de energia em RSSFs é extremamente complexa, pois não se limita apenas a diminuir o consumo de um único nó sensor, mas também a prolongar o tempo de vida da rede como um todo. Para alcançar este objetivo, é necessário que um judicioso controle de dissipação de energia seja incorporado em todos os estágios do projeto e operação de uma RSSF.

O sistema deve ainda executar avaliações dinâmicas entre o consumo de energia, nível de desempenho e fidelidade operacional [RAG 02].

2.5.2 Tolerância a Falhas

As RSSF podem ser empregadas em ambiente possivelmente hostil, como aplicações críticas, o que elevam as chances de falhas, portanto, deve-se considerar a confiabilidade dos dados gerados.

Os nós sensores podem falhar ou tornarem-se inoperantes por falta de energia, danos físicos, ou ainda, interferência. As falhas dos nós sensores não devem afetar a tarefa global da rede de sensores, que é a de fornecer informações sobre o ambiente monitorado. A tolerância a falhas, ou confiabilidade da rede, é a habilidade de manter as funcionalidades da rede sem interrupção mesmo mediante falhas dos nós [HOB 00] [SHE 01].

Protocolos e algoritmos devem ser implementados para alcançar os níveis de tolerância a falhas requisitadas pelas aplicações das redes de sensores [AKY 02], ou seja, se uma rede de sensores for utilizada em um ambiente no qual há pouca interferência, o protocolo pode ser um pouco mais simples. Uma rede de sensores em um ambiente doméstico e controlado deve ter um nível de tolerância a falha muito menor do que uma rede de sensores que será utilizada na entrada de um vulcão prestes a entrar em erupção.

2.5.3 Limitações de Hardware

O consumo de energia é uma das principais limitações em nós sensores. Como os nós sensores são normalmente inacessíveis, o tempo de vida da rede depende do tempo de vida das fontes de energia dos nós. A energia é também um recurso escasso devido à limitação de tamanho do hardware do nó sensor [MIN 01]. É possível estender o tempo de vida da rede de sensores por meio da extração de energia do ambiente. Um exemplo são as células solares, que transformam a energia térmica em energia elétrica.

Embora um maior poder computacional esteja sendo disponibilizado em processadores cada vez menores, os módulos de processamento e de memória dos nós sensores ainda são recursos escassos. Por exemplo, a unidade de processamento de um nó Mica 2 mote é um micro-controlador de 8 MHZ com 128 KB de memória *flash* para instruções, 4 KB de memória RAM e 4 KB de EEPROM [LEW 10].

2.5.4 Expansibilidade

A suscetibilidade a falhas e a necessidade de cobertura de extensas áreas podem exigir considerável número de nós sensores. O número de nós sensores empregados podem atingir a ordem de centenas ou milhares e, dependendo da aplicação, chegar a milhões. Novos protocolos e soluções de RSSF devem ser capazes de funcionar com este elevado número de nós. Dessa característica de rede, deriva um importante parâmetro de sistemas distribuídos: a densidade de nós. A densidade $\mu(R)$ pode ser obtida pela expressão $(N\pi R^2)/A$, onde N é o número de nós sensores com raio de transmissão R que são lançados em uma região de área A [PIN 04].

2.5.5 Agregação de Dados

Os nós sensores normalmente podem gerar dados redundantes. Pacotes similares de múltiplos nós podem ser agregados para que o número de transmissões seja reduzido. A agregação de dados, ou fusão, é a combinação de dados de fontes diferentes por meio do uso de funções como supressão (eliminação de pacotes duplicados), mínimo, máximo, média, etc. [EST 99]. Algumas dessas funções podem ser utilizadas parcial ou integralmente em cada nó sensor, permitindo que os nós reduzam os dados na rede [ELS 01] [EST 00] [GIR 01]. Ao reconhecer que o processamento consome menor energia que a comunicação, surge a possibilidade em economizar energia através da agregação de dados. Essa técnica tem sido usada para alcançar a eficiência no consumo de energia e otimização de tráfego em alguns protocolos de roteamento [ELS 01] [GIR 01].

2.5.6 Ambiente de Operação

Os nós de uma rede de sensores geralmente irão trabalhar sensoriando locais inóspitos, sejam muito próximo ou dentro deles. Alguns locais onde as redes de sensores podem ser utilizadas [AKY 02]:

- Dentro de um furacão;
- Em campos contaminados biologica ou quimicamente;
- Em campos de batalha nas linhas inimigas;
- No interior de maquinários;
- No fundo do oceano;
- Acoplados a animais;
- Acoplados a veículos.

Os nós ainda podem funcionar em ambientes com calor ou frio extremo, sob alta pressão, em ambientes ruidosos etc.

2.5.7 Topologia da Rede

Centenas a milhares de nós sensores são utilizados em uma rede e podem ser lançados ou colocados um a um no local que se deseja monitorar. Devido ao fato do grande número de nós sensores e das frequentes falhas o gerenciamento da topologia das RSSFs é uma tarefa desafiadora. Mudança na topologia da rede é uma característica comum em RSSFs. No entanto, as mudanças de topologia das RSSFs não são atribuídas ao movimento dos nós, como acontece em redes *ad hoc* tradicionais ou nos sistemas celulares. Os nós sensores permanecem, em sua maioria, estacionários após a deposição. As mudanças na topologia ocorrem quando os nós deixam de operar por falta de energia, são destruídos ou os rádios são desligados para economizar energia [MIN 02]. Além disso, deve ser prevista a possibilidade de adição de novos nós na rede, após a sua disposição inicial, para a ampliação da área de cobertura.

2.5.8 Meio de Transmissão

Uma RSSF pode explorar três possibilidades de comunicação sem fio: ótica, infravermelho e rádio frequência. A comunicação ótica consome menos energia por bit

transmitido e é sensível às condições atmosféricas [RUI 04b]. A comunicação por infravermelho é robusta contra interferências de dispositivos elétricos. Nenhum desses meios de comunicação necessita de uma área física para instalação de antena, porém exigem linhas de visão (*LOS – Line of Sight*) para comunicação, ou seja, alinhamento entre transmissor e receptor. Estes meios de comunicação não são viáveis para aplicações de monitoramento, pois a comunicação direcional nestes ambientes nem sempre é possível. A comunicação por rádio frequência é baseada em ondas eletromagnéticas, e suas vantagens são as facilidades de uso e aceitação comercial, tornando este tipo de comunicação viável para plataformas de sensores [SHI 01]. A maioria dos projetos de hardware para nós sensores são baseados em circuitos de radio frequência. O nó sensor μAMPS, descrito em [SHI 01], usa um transceptor Bluetooth de 2.4 GHz. O dispositivo sensor descrito em [LEW 10], usa um canal simples de radio frequência operando a 916 MHz.

2.5.9 Qualidade de Serviço (QoS)

Qualidade de serviço (QoS) é um requisito importante de RSSF, pois a comunicação nas RSSFs apresenta limitações como largura de banda reduzida, altas taxas de erro e poder computacional limitado, sendo desejável uma melhor utilização dos recursos de rede por meio de mecanismos que provejam qualidade de serviço (QoS), esse requisito apresenta ainda muitos desafios não superados [CHE 04]. Esses desafios provêm das diferentes características que as RSSF apresentam se comparadas às redes convencionais, a saber: limitações severas de recursos de processamento, armazenamento e comunicação, tráfego não uniforme, alta redundância de dados, rede dinâmica, gasto de energia uniforme, escalabilidade, múltiplos *sinks*, múltiplos tipos de tráfego, etc.

2.6 Classificação das RSSFs

Uma RSSF é um tipo de sistema dependente da aplicação, portanto, todos os fatores citados anteriormente são influenciados pelos requisitos da aplicação. De acordo com os objetivos da aplicação, os parâmetros de configuração e manutenção podem variar. De

forma semelhante, a classificação das RSSF também depende de seu objetivo e de sua área de aplicação. A Tabela 1, mostra uma classificação das RSSFs baseado na configuração de rede.

Tabela 1 - Classificação das RSSFs baseada em configurações da rede. (Adaptado de [RUI 04b])

Configuração		
Composição	**Homogênea**	Rede composta de nós de mesma capacidade de hardware.
	Heterogênea	Rede composta de nós com diferentes capacidades de hardware.
Densidade	**Balanceada**	Quando a rede apresenta alta concentração de nós sensores por unidade de área considerada ideal.
	Densa	Quando a rede apresenta alta concentração de nós sensores por unidade de área.
	Esparsa	Quando a rede apresenta baixa concentração de nós sensores por unidade de área.
Mobilidade	**Estacionária**	Rede em que os nós sensores são fixos durante toda a vida da rede.
	Móvel	Rede em que os nós sensores podem ter sua posição espacial variável durante o tempo de vida de rede. Isto é, estes nós podem ter a posição inicial onde foram depositados modificados por sua capacidade de locomoção, por forças da natureza ou pela entidade na qual estão acoplados.
Distribuição	**Irregular**	Rede que apresenta distribuição não uniforme na área monitorada.
	Regular	Rede que apresenta distribuição uniforme na área monitorada.
Organização	**Hierárquica**	Rede em que os nós são agrupados (*cluster*s), onde cada grupo contém um líder (*cluster-head*). Em redes que possuem esta organização, os grupos formam hierarquias entre si, de forma que quando um dado necessita ser remetido à estação base, ele é remetido primeiro ao líder que irá enviar diretamente, ou através de outros líderes, à estação base. Fases de seleção dos nós líderes são necessárias. Esta organização tem primordialmente a intenção de reduzir o tráfego da rede objetivando economia de energia.
	Plana	Rede em que os nós não são posicionados em grupos.

As RSSFs ainda podem ser classificadas quanto ao modo de sensoriamento e coleta dos dados em [RUI 04b]:

- **Periódica -** Os nós sensores coletam dados sobre os fenômenos em intervalos regulares. Por exemplo, relatar a temperatura a cada dez minutos.

- **Dirigida a Eventos -** Os nós sensores coletam dados quando ocorrem eventos de interesse. Por exemplo, indicar quando temperatura for maior que 60 graus.
- **Baseada em Consultas -** Os nós sensores coletam dados quando solicitado pelo observador. Por exemplo, relatar a temperatura atual.

Qualquer projeto ou solução proposta para estas redes deve considerar características e restrições dos nós sensores, requisitos das aplicações a serem desenvolvidas, e características do ambiente onde tais redes serão aplicadas.

2.7 Arquitetura de Comunicação para RSSFs

Esta subseção descreve a pilha de protocolos comumente admitida para RSSFs, ilustrada na Figura 2.6, discutindo as características e funcionalidades para que se tenha conhecimento dos objetivos de cada camada da arquitetura de comunicação.

Uma aplicação usando a rede não interage diretamente com o *hardware* da rede. Ao invés disso, a aplicação interage com o software do protocolo. A noção de protocolos em camadas fornece um conceito básico para o entendimento de como um conjunto complexo de protocolos pode trabalhar junto com o *hardware* para prover um sistema de comunicação eficiente. A pilha de protocolos da RSSF caracteriza-se pela exploração do esforço colaborativo e da disponibilidade de energia dos elementos da rede para o estabelecimento de rotas, agregação de dados e a comunicação empregando o meio sem fio. A pilha de protocolos da RSSF é dividida nas camadas física, de enlace de dados, de rede, de transporte e de aplicação [AKY 02].

Figura 2.6 – Pilha de Protocolos da RSSF. (Adaptado de [AKY 02])

2.7.1 Camada Física

A camada física é responsável por seleção de frequência, geração da frequência portadora, detecção de sinal, modulação e encriptação de dados. A geração de frequência e detecção de sinal estão relacionadas mais ao projeto de hardware do transceptor.

Como descrito na subseção 2.5.8, uma RSSF pode explorar três possibilidades de comunicação sem fio: ótica, infravermelho e rádio frequência. As vantagens da comunicação em rádio frequência são a facilidade de uso e a aceitação comercial, que tornam este tipo de comunicação viável para plataformas de nós sensores [RUI 04b].

A comunicação sem fio de longa distância pode ser muito custosa em termos de energia e complexidade de implementação. Vários aspectos afetam o consumo de energia do rádio, incluindo tipo de modulação, taxa de dados, e energia de transmissão. Em geral os rádios podem operar em quatro modos distintos, transmitindo, recebendo, *idle* e *sleep* [RUI 04b].

Os modelos TR1000 e CC1000 são dois exemplos de rádio usados em nós sensores. O modelo TR é um transceptor de rádio híbrido que suporta transmissão de dados em taxas superiores a 115.2 kbps, com alcance de 30 a 90 metros. Com estas taxas ele consome aproximadamente 14.4 mW na recepção, 36 mW durante a transmissão e 15 µW no modo *sleep*. O rádio Chipcon CC1000 é um transceptor de baixo consumo de energia que obtém transferência de dados de até 76.8 kbps. No modo de baixo consumo, a corrente consumida é de 0.2 µA. A tensão de operação varia de 2.1 a 3.6 V [RUI 04b].

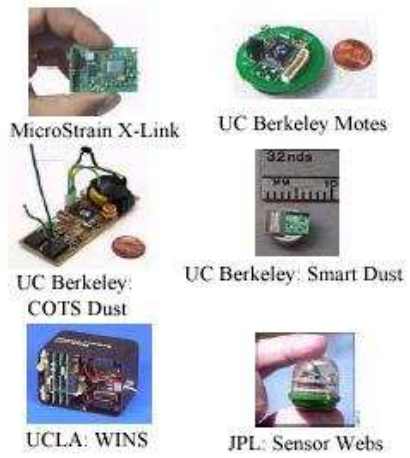

Figura 2.7 – Exemplos de projetos de nós sensores [RUI 04b].

A Figura 2.7 mostra os projetos de nós sensores para redes de sensores sem fio. Suas características comuns são: pequenos em tamanho, baixo consumo de energia, possuem

sensores para luminosidade, temperatura, umidade, pressão, aceleração, etc.; possuem módulos de rádio e processamento.

2.7.2 Camada de Enlace

O objetivo da camada de enlace é assegurar conexões confiáveis em uma rede de comunicação, por meio das tarefas de detecção dos quadros de dados, multiplexação dos fluxos de dados, acesso ao meio e controle de erro. Embora muitas dessas funções ainda não tenham sido completamente definidas, o ponto mais crítico e de fundamental importância para diversas aplicações de RSSF é o protocolo de acesso ao meio (MAC).

Um protocolo de controle de acesso ao meio para redes de sensores sem fio deve atingir dois objetivos [AKY 02]. O primeiro é a criação de uma infra-estrutura de rede. Como milhares de nós sensores estão densamente espalhados por um campo de sensores, os esquemas MAC devem estabelecer enlaces de comunicação para transferência de dados. Isso forma a infra-estrutura básica necessária para a comunicação sem fio nó a nó e oferece a habilidade de auto-organização para a rede de sensores. O segundo objetivo é compartilhar eficientemente os recursos de comunicação entre os nós sensores.

Como em toda rede que compartilha o meio de transmissão, o protocolo MAC é fundamental para o sucesso da operação da rede. A função mais importante do protocolo é evitar colisões. Existem diversos protocolos MAC disponíveis para redes sem fio. No entanto, esses protocolos possuem restrições quando aplicados às RSSFs [AKY 02]. Por exemplo, o acesso múltiplo por divisão do tempo (*Time Division Multiple Access* - TDMA) exige uma elevada taxa de transmissão e coordenação entre os nós, enquanto o acesso múltiplo por divisão de código (*Code Division Multiple Access* - CDMA) e os protocolos baseados em contenção da especificação IEEE 802.11 requerem que os nós escutem o canal para receber um possível tráfego, o que provoca um elevado dispêndio de energia. Tais aspectos implicam em modificações ou desenvolvimento de novos protocolos de acesso ao meio, específicos para RSSFs.

2.7.3 Camada de Rede

Os protocolos da camada de rede são responsáveis pelo roteamento dos dados entre os nós sensores e o *sink*. O roteamento pode ser definido como o processo pelo qual a rede consegue identificar o destinatário da mensagem e encontrar um caminho entre a origem e o destino desta mensagem [RUI 04b].

Ao contrário dos protocolos tradicionais que buscam diminuir o retardo fim-a-fim ou aumentar a vazão, em RSSFs a maior parte das pesquisas visam estabelecer rotas que permitam estender o tempo de vida da rede por meio da racionalização do consumo de energia, muitas vezes, com sacrifício de outras métricas de desempenho, como latência [YOU 02].

A fusão/agregação de dados tem sido apontada como uma opção que permite otimizar a operação das RSSFs. A idéia é pré-processar os dados dentro da rede reduzindo a ocorrência de redundâncias e o número de transmissões para economizar energia. Essa técnica modifica o foco da abordagem tradicional, centrada em endereço, para uma abordagem nova, centrada em dados, que permite a consolidação ou sumarização de dados redundantes [RUI 04b].

2.7.4 Camada de Transporte

Essa camada é necessária especialmente quando o sistema precisa ser acessado pela Internet ou por outras redes externas [AKY 02]. Diferentemente de protocolos como TCP, os esquemas de comunicação fim-a-fim nas redes de sensores não são baseados em endereçamento global. Esses esquemas devem considerar que uma nomeação de dados baseada em atributos é utilizada para indicar os destinatários dos pacotes. Além disso, fatores como consumo de energia e expansibilidade, e características como roteamento centrado em dados (*data-centric*) exigem um tratamento diferente na camada de transporte de uma rede de sensores [POT 00].

2.7.5 Camada de Aplicação

Existem diversas aplicações definidas e propostas para RSSFs, cada qual com diferentes requisitos. Na literatura, vários trabalhos [TIL 02] [HEI 03] destacam a importância da participação da aplicação no processo de comunicação em RSSFs. Otimizações específicas da aplicação podem reduzir o número de transmissões e, por conseguinte, o consumo total de energia na rede. No entanto, protocolos para camada de aplicação de RSSFs ainda constituem uma região inexplorada.

A camada de aplicação contém uma série de protocolos de alto nível que são comumente necessários. Ela faz a interface entre o protocolo de comunicação e o aplicativo que pediu ou recebeu a informação através da rede.

2.8 Redes de Sensores e Atuadores Sem Fio

Esta seção apresenta as características das redes de sensores e atuadores sem fios (RSASFs), cuja adição de novos elementos chamados de atuadores fornece vantagens adicionais às RSSFs. Em RSASFs observa-se que muitos dos conceitos das RSSFs permanecem válidos.

As RSASFs possuem a capacidade adicional de reação automática e em tempo real aos eventos capturados, mas possuem as mesmas propriedades das RSSFs conforme estudado nas seções anteriores. Algoritmos e protocolos desenvolvidos para as RSASFs devem atender aos mesmos requisitos das RSSFs, tais como: escalabilidade e eficiência energética; além disso, devem considerar a heterogeneidade dos nós e os requisitos de aplicações de tempo real [AKY 04].

Estas redes podem integrar as mesmas áreas de atuação das RSSFs. Um exemplo de RSASF é o monitoramento de fogo em que nós sensores detectam e transmitem as informações que compõem o fogo (presença de oxigênio, presença de fumaça, temperatura alta) e nós atuadores “atuam” no ambiente de forma ativa, extinguindo o fogo, por exemplo, com nós “*sprinklers*” de água.

2.8.1 Componentes do Nó Atuador

Nós atuadores e nós sensores diferem-se em vários aspectos. Nós atuadores, atuam no ambiente e nós sensores monitoram. Além disso, os atuadores possuem um custo mais alto, melhores capacidades de comunicação sem fio, processamento, e uma bateria de maior duração. Já os nós sensores são de baixo custo, baixas capacidade(s) de comunicação sem fio, computação, e dispositivo(s) de baixa energia com sensoriamento limitado. Outra diferença refere-se à quantidade, o número de nós sensores implantados no ambiente, geralmente, são da ordem de centenas ou milhares, no entanto não é necessária uma densa distribuição de nós atuadores, pois os atuadores possuem recursos melhores e podem atuar em grandes áreas [AKY 04].

As unidades dos nós atuadores são: energia, comunicação, processamento, decisão (controlador), atuação e um conversor digital analógico (DAC). A Figura 2.8 mostra esses componentes. Os eventos são recebidos pela unidade de comunicação, então cada evento é analisado na unidade de decisão gerando comandos que executam ações como saídas. Esses comandos são então convertidos para sinais analógicos via DAC e são transformados em ações via unidade de atuação.

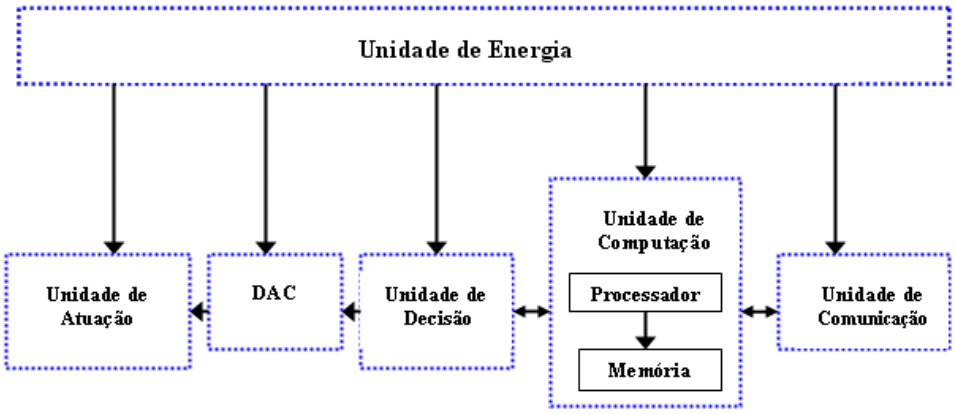

Figura 2.8 - Componentes do nó Atuador. (Adaptado de [AKY 04])

2.8.2 Funcionamento Básico de RSASF

O funcionamento da Rede de Sensores e Atuadores Sem Fio assemelha-se muito com a Rede de Sensores Sem Fio, os sensores coletam informações do mundo físico. A

diferença está quando os nós atuadores estão presentes, os atuadores processam informações para tomar decisões e realizar ações apropriadas sobre o ambiente. A Figura 2.9 mostra os nós sensores e os nós atuadores que estão espalhados sobre uma área de sensoriamento e atuação. O *sink* monitora toda a rede e realiza a comunicação da RSASF com a aplicação, ou pode disponibilizar as informações coletadas a outras redes. Este cenário permite aos usuários realizarem o sensoriamento e atuação no ambiente a distância.

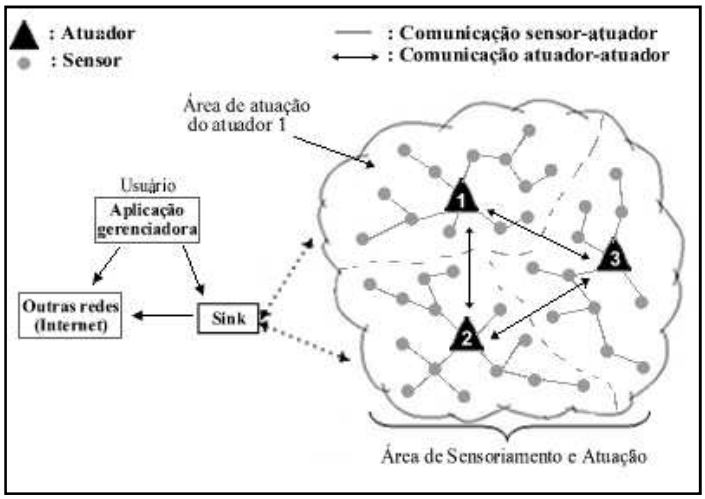

Figura 2.9 - Esquema da arquitetura física das RSASFs [VIL 07].

Sensores, ao detectarem um fenômeno, podem operar de duas maneiras distintas: (a) Interpretação centralizada: rotear os dados de volta ao *sink*; (b) Interpretação descentralizada: transmitir os dados ao atuador que, ao processar estes dados, inicia uma ação apropriada. Em [AKY 04] foram definidas duas arquiteturas de RSASF para os dois modos de operação em que, dependendo da aplicação, uma delas se torna mais vantajosa.

A primeira arquitetura, chamada de Semi-Automática, é bastante similar a arquitetura tradicional das RSSFs, os nós sensores ao detectarem um evento, transmitem suas leituras ao *sink*, que é responsável por interpretá-las, e caso seja necessário, enviar comandos de atuação aos atuadores.

A segunda é a arquitetura Automática, as leituras detectadas pelos nós sensores são enviadas aos atuadores, que são responsáveis por interpretá-las, e caso seja necessário, atuar no ambiente. A primeira arquitetura demonstra um modo de interpretação centralizada, o *sink* seria a entidade central responsável por coordenar todos os atuadores, enquanto a segunda demonstra um modo de interpretação distribuída, onde cada atuador seria responsável por interpretar os dados de sua região de atuação.

2.9 Considerações Finais

Neste capítulo foram apresentados aspectos, requisitos, aplicações e características importantes sobre as RSSFs e RSASFs. Um fator importante presente nessas redes é o consumo de energia. Devido à escassez deste recurso, há um esforço em conjunto de pesquisadores, que tentam amenizar esse problema desenvolvendo soluções. Neste capítulo foi representado que as RSASFs fornecem vantagens adicionais a RSSF, possibilitando a reação automática, e em tempo real, aos eventos coletados pelos nós sensores, por meio de nós atuadores.

No próximo capítulo é apresentado o conceito de *middleware*, em especial para as RSSFs, e os trabalhos existentes nessa área.

3. Sistemas Operacionais e Middlewares para RSSFs

3.1 Considerações Iniciais

Um sistema distribuído consiste em uma coleção de componentes entre vários dispositivos conectados via uma rede. Os componentes precisam interagir entre si, a fim de trocar dados ou acessar os serviços um do outro. Essa interação poderia ser construída diretamente no topo das primitivas do sistema operacional, porém seria muito complexo para os desenvolvedores de aplicações. Um *middleware* localizado entre os componentes do sistema distribuído e componentes do sistema operacional pode prover o suporte para os desenvolvedores de aplicações [MAS 02].

Neste capítulo são realizadas revisão de conceitos, requisitos e descrição de alguns dos principais sistemas operacionais para as RSSFs. São também descritos conceitos e classificação de sistemas de *middleware*, seguida de conceitos, princípios de projeto, desafios e classificação de *middleware* para as RSSFs, além de comparações entre os sistemas operacionais e *middleware* para as RSSFs, seguida de considerações finais.

3.2 Sistemas Operacionais para RSSFs

Os sistemas operacionais (SO) típicos para computação embarcada, como VxWorks, QNX, OS-9, WinCE e µClinus fornecem um ambiente de programação semelhante ao existente em computadores de mesa. Muitos destes SOs fornecem e, por consequência, exigem suporte de *hardware* à proteção de memória. Apesar de serem bastante adequados para o uso em telefones celulares e aplicações embarcadas complexas, os requisitos de processamento e memória fazem com que eles sejam inadequados para o uso em RSSFs [WAN 08].

A necessidade de conectividade, abstração de *hardware* e gerência de recursos limitados, torna o suporte em nível do sistema operacional imperativo para aplicações em RSSFs. Baseado na tecnologia, aplicações e pesquisas atuais, é possível listar uma série de requisitos de sistemas operacionais para as RSSFs [WAN 08]:

- **Fornecer a funcionalidade básica de um sistema operacional:** um sistema operacional para os nós sensores deve prover serviços tradicionais como: abstração de *hardware*, gerência de processos, serviços de temporização e gerência de memória;
- **Fornecer mecanismos para gerência do consumo de energia nos nós sensores:** um sistema operacional para os nós sensores deve fornecer mecanismos de gerência de energia às aplicações, bem como utilizar os recursos de *hardware* de maneira a consumir o mínimo de energia necessário a execução adequada das aplicações;
- **Prover mecanismos para reprogramação em campo:** um sistema operacional para os nós sensores deve prover algum mecanismo de reprogramação total ou parcial de aplicações já instaladas. Dado que os nós de uma RSSF podem estar localizados em regiões de difícil acesso, e que os requisitos e parâmetros das aplicações de sensoriamento podem mudar com o tempo, a reprogramação em campo, via rede de comunicação, é um fator importante em RSSFs.
- **Abstrair o hardware de sensoriamento heterogêneo de maneira uniforme:** uma aplicação desenvolvida para determinada plataforma, dificilmente é portável para outra diferente, a menos que os sistemas de suporte de tempo de execução destas plataformas abstraiam e encapsulem adequadamente a plataforma de sensoriamento.
- **Fornecer uma pilha de protocolos de comunicação configurável:** o sistema operacional deve prover um mecanismo de configuração da pilha de comunicações a ser utilizada na rede, dadas as necessidades específicas de comunicação de diferentes aplicações, e o requisito de que o hardware de comunicação seja amplamente configurável.
- **Operar com recursos limitados:** um sistema operacional para os nós sensores deve entregar os serviços necessários às aplicações, sem consumir uma parcela significativa dos recursos computacionais disponíveis.

Entre os sistemas operacionais desenvolvidos especificamente para as RSSFs, cabe ressaltar SOS [HAN 05], Contiki [DUN 04] e TinyOS [HIL 00].

3.2.1 SOS

O SOS é um sistema operacional para redes de sensores reconfigurável dinamicamente. O sistema é organizado como um conjunto de módulos binários, que implementam tarefas ou funções específicas, comparáveis em funcionalidade com os componentes do TinyOS [WAN 08].

Uma aplicação do sistema é composta por uma série de módulos que interagem entre si e apresentam interfaces de métodos e passagem de mensagens. A passagem de mensagem do SOS funciona de maneira assíncrona e é coordenado por um escalonador que retira mensagens de uma fila ordenada por prioridade e passa a mensagem ao tratador adequado do módulo de destino [WAN 08].

O SOS inclui um alocador dinâmico de memória e um *garbage collector*. Como no TinyOS o escalonador coloca o processador em modo de baixo consumo quando não há tarefas para escalonar [WAN 08].

O modelo de reconfigurabilidade dinâmica do SOS faz com que o sistema tenha requisitos de memória e sobrecusto consideravelmente maior do que os outros sistemas operacionais citados. Além disso, o SOS não tem um programador de tarefas em tempo real e não garante a execução de seus módulos em tempo real [ROC 09a].

3.2.2 Contiki

O Contiki é um sistema operacional que foi desenvolvido para ambientes confinados e de difícil acesso, que provê uma execução dinâmica de programas e serviços. O sistema suporta *multithreading* preemptivo que é implementado por bibliotecas utilizadas pelas aplicações [ROC 09a].

O Contiki também fornece comunicação IP e implementa a uIP, uma pilha TCP/IP controlada por um micro controlador de 8 a 16 bits e provê os protocolos necessários para a comunicação pela Internet [ROC 09a].

Contiki é escrito na linguagem de programação C e consiste de um *kernel* dirigido a eventos, sobre os quais aplicações podem ser dinamicamente carregadas e descarregadas em tempo de execução. Processos em Contiki usa leves *protothreads* que

fornece um estilo de programação em *thread* sobre o topo do *kernel* dirigido a eventos. Além de *protothreads*, Contiki também suporta *multithreading* de forma opcional por processo e a comunicação interprocesso acontece por passagem de mensagem [SIC 09].

3.2.3 TinyOS

O TinyOS (*Tiny Microthreading Operating System*) é um sistema operacional de código aberto, baseado em eventos e projetado especificamente para as RSSFs. A arquitetura do TinyOS é baseada em componentes, possui *drivers* de sensor e ferramentas para aquisição de dados. Além de um sistema operacional, o TinyOS pode ser definido também como uma plataforma de desenvolvimento para uma RSSF, pois ele contém uma gama de programas que visam facilitar o trabalho do programador, tal como o compilador NesC [HIL 00].

O TinyOS apresenta um modelo de concorrência baseado em tarefas que executam até sua completude, e que só podem ser preemptadas por interrupções. Um modelo tradicional, baseado em *threads* com pilhas próprias exige que cada *thread* reserve espaço para o seu contexto de execução na memória. Ao não permitir concorrência entre tarefas, o TinyOS reduz boa parte deste sobrecusto, mas perde as características de um modelo *multithreading* tradicional. A restrição de concorrência limita também a capacidade do sistema em tratar tempo real [WAN 08].

As bibliotecas e as aplicações em TinyOS são escritos na linguagem NesC, uma linguagem de alto nível, baseada em componentes e tem uma sintaxe semelhante à linguagem C. O principal objetivo é permitir aos desenvolvedores de aplicação construir os componentes que possam facilmente ser compostos em sistemas completos e simultâneos [SOU 05].

A comunicação entre os nós da rede é baseada no paradigma *active message*. De acordo com este paradigma, cada mensagem contém o ID de uma rotina de tratamento a ser invocada no nó alvo e um *payload* contendo os dados a serem passado como argumentos [BUO 01]. Esta comunicação baseada em eventos e orientada a mensagem faz do TinyOS uma boa fundação para a construção de uma infra-estrutura de comunicação *publish/subscribe*.

3.3 Comparação Entre os Sistemas Operacionais para RSSFs

A seguir, é apresentada uma tabela comparativa e auto-explicativa das principais funcionalidades dos sistemas operacionais discutidos neste trabalho.

Tabela 2 - Comparação dos sistemas operacionais para RSSF. (Adaptado de [ROC 09a])

Funcionalidades	TinyOS	SOS	Contiki
Módulo de Energia	✓	✓	
Programação dinâmica	✓	✓	✓
Prioridade – agendamento			✓
Modelo de execução	Componentes	Módulo	Tarefas
Multithreading			✓
Portabilidade		✓	✓
Facilidade em usar		✓	
TCP/IP			✓
Linguagem	NesC	C	C

O TinyOS é o sistema operacional mais utilizado pela comunidade científica em RSSFs. Na segunda versão, foi aprimorado o *Boot Sequence*, que permite maior agilidade na inicialização do SO, ocupando menos memória RAM e ROM. O TinyOS também tornou-se mais confiável para aplicações finais, mais portável e já é compatível com as plataformas eyesIFX2, intelmote2, mica2, micaZ e telosb.

3.4 Sistemas de *Middleware*

Segundo [BLA 04], *middleware* é um artefato em software que reside entre a aplicação e o sistema operacional, que fornece por meio de interfaces o reuso de serviços, facilitando o desenvolvimento de aplicações mais eficientes para ambiente distribuído tal como uma RSSF. O principal objetivo de um *middleware* é possibilitar a comunicação entre

componentes distribuídos, escondendo das aplicações a complexidade do ambiente de rede subjacente e livrando-as da manipulação explícita de protocolos e serviços de infra-estrutura.

Em [MAS 02] é apresentado um modelo de referência que classifica os sistemas de *middleware* entre fixos e *ad-hoc* ou móveis. Essa classificação leva em consideração três aspectos, o tipo de carga computacional, o paradigma de comunicação e a representação de contexto.

Middleware para sistemas distribuídos tradicionais ou fixos possuem algumas limitações que faz o uso deles impraticável nas RSSFs. Esses sistemas demandam recursos computacionais (carga computacional pesada), escondem informações de contexto das aplicações tanto quanto possível (transparência), e suportam comunicação síncrona entre componentes, com exceção do *middleware* orientado a mensagem (MOM). A comunicação síncrona não é adequada para ambientes com muitas desconexões. Outro problema é a carga computacional dos *middlewares* tradicionais, para os poucos recursos dos dispositivos de uma RSSF. A transparência apresentada nem sempre é adequada para as aplicações nas RSSFs, que necessitam de alguma informação sobre o contexto de execução para uma melhor adaptabilidade [MAS 02].

Ao projetar um *middleware* para as RSSFs deve ser levado em consideração as características dos dispositivos que as constitui, assim como suas limitações.

As RSSFs são uma categoria de redes sem fio *ad-hoc*, e devem ser projetadas obedecendo a alguns dos requisitos de *middleware* para redes ad-hoc ou móveis. Os principais requisitos incluem carga computacional leve para suportar os dispositivos, prover comunicação assíncrona para contornar os problemas causados pelas frequentes desconexões e tornar as aplicações cientes de contexto. Dessa forma, novos tipos de *middleware* têm sido projetados e desenvolvidos para atender diversos tipos de aplicações em RSSFs [MAS 02].

3.5 *Middleware* para Rede de Sensores Sem Fios

Segundo [ROM 02], o principal propósito de *middleware* para RSSF é suportar o desenvolvimento, manutenção, distribuição, e execução de aplicações de sensoriamento. Isso inclui mecanismos para formulações complexas de tarefas de sensoriamento de alto nível, a comunicação dessas tarefas com a RSSF, coordenação de nós sensores para distribuição das

tarefas, agregação e ou fusão de dados para unir as leituras dos sensores em um resultado de alto nível e reportar os resultados das tarefas de volta para o emissor.

Em [MIA 08] descreve três formas que podem ajudar os desenvolvedores de aplicações a utilizar um *middleware* para RSSF. Primeiro, o *middleware* pode fornecer abstrações de sistemas apropriados, de forma que o programador da aplicação pode focar na lógica da aplicação sem se preocupar sobre detalhes de implementação de baixo nível. Segundo, o *middleware* pode prover o reuso de código de serviços tais como atualização, serviços de dados, assim como filtragem de dados, dessa forma programadores de aplicações podem distribuir e executar aplicações sem se preocupar com complexas e tediosas funções. Terceiro, o *middleware* pode ajudar os programadores no gerenciamento da infra-estrutura da rede e na adaptação, fornecendo recursos eficientes de serviços, como gerenciamento de energia, além de suportar integração de sistemas, monitoramento e segurança.

Em [YU 04] foram propostos princípios a serem adotados em um projeto de *middleware* para RSSFs.

- O *middleware* deve fornecer mecanismos centrados em dados para o processamento e a consulta de dados no interior da rede.
- Algoritmos localizados devem ser usados para alcançar um desejável objetivo enquanto fornecem boa escalabilidade e robustez ao sistema.
- *Middlewares* tradicionais são projetados para suportar uma ampla variedade de aplicações na rede. Mas devido aos recursos limitados disponíveis, *middleware* para RSSFs não podem ser generalizados dessa forma.
- A disponibilidade de recursos dos nós sensores é baixa, o *middleware* então deve ser leve em termos de requisitos de comunicação e computação.
- Devido aos recursos limitados, é muito provável que os requisitos de desempenho de todas as aplicações em execução não possam ser simultaneamente satisfeitos. Portanto, é necessário que o *middleware* negocie de forma inteligente a QoS de várias aplicações umas contra as outras.

Ambos primeiros princípios de projeto motivam fortemente as arquiteturas baseadas em *cluster*. De fato, tais arquiteturas foram amplamente investigadas em redes *ad hoc* desde que eles “promovem mais eficientes uso dos recursos em controlar redes dinâmicas grandes”. Comparadas com redes móveis, que possui um alto custo de manutenção dos clusters, as RSSFs usualmente consistem de nós sensores estacionários com menos dinamismo [YU 04].

3.5.1 Desafios no Projeto de *Middleware* para RSSFs

O projeto e desenvolvimento de *middleware* impõem vários desafios devido as característica das RSSFs, por exemplo, restrições de recursos, disponibilidade e diversidade de hardware de sensor. Mohammad [MOL 06] elaborou e descreveu vários desafios associados com *middleware* para RSSFs. Alguns dos desafios citados abaixo por Mohammad são os mesmos citados em princípios de projetos.

a) **Suporte na abstração**: as RSSFs consistem de um grande número de sensores heterogêneos. Os sensores são desenvolvidos por empresas diferentes e talvez tenha diferentes plataformas de hardware. Esconder as plataformas de hardware subjacente para oferecer uma visão holística da rede é um dos principais desafios dos *middlewares* para RSSFs.

b) **Fusão/Agregação de dados**: sensores são usados para coletar dados do ambiente. Coletar dados de vários sensores, unir, agregar e apresentar os dados em mais alto nível é outro importante desafio.

c) **Restrições de recursos**: *middleware* para RSSFs tem que ser leve para funcionar sobre hardware limitado em recursos.

d) **Topologia dinâmica**: *middleware* para RSSFs deve ser capaz de lidar com a topologia dinâmica da rede, devido à mobilidade, falha de nós, e falha na comunicação entre os nós.

e) **Conhecimento da aplicação**: *middleware* para RSSFs deve integrar conhecimento da aplicação nos serviços disponíveis. Otimizações de rede podem ser alcançadas com o conhecimento do nível da aplicação. Por exemplo, características da aplicação podem influir tanto na infra-estrutura da rede, quanto nos protocolos utilizados. O conhecimento da aplicação pode ser aproveitado pela rede para que ela possa alcançar uma maior eficiência em termos de consumo de energia prolongando, assim, o tempo de vida da rede.

f) **Paradigma de programação**: os paradigmas de programação para RSSFs são diferentes de estilos de programação tradicional, devido às restrições de recursos, topologia dinâmica da rede, e dificuldades envolvidas na coleta e processamento de dados do sensor.

g) **Adaptabilidade**: *middleware* para RSSFs deve suportar algoritmos que tenham desempenho adaptativo.

h) **Escalabilidade**: *middleware* para RSSFs talvez seja escalável em termos de números de nós, números de usuários, etc. para operar por longos períodos de tempo.

i) **Segurança**: redes de sensores têm que tratar questões de segurança no processamento e comunicação de dados. Mas devido às limitações de recursos e ao baixo poder computacional, a maioria dos algoritmos existentes e modelos de segurança não são adequados para as redes de sensores.

j) **Suporte de QoS**: *middleware* para RSSFs deve também resolver várias questões de QoS, tempo de resposta, disponibilidade, largura de banda, alocação, etc.

3.5.2 Classificação dos *Middleware* para RSSFs

Nos últimos anos, várias abordagens de *middleware* para RSSFs foram propostas. Em [MAR 05a] foi proposta uma classificação de *middleware* para as RSSFs, categorizando os projetos considerando os tipos de níveis de abstração. A Tabela 3 ilustra alguns dos projetos mais relevantes de *middleware* para RSSFs.

- Clássica: escondem a complexidade de comunicação da rede e a transferência de dados;
- Centrado em Dados: fornece a abstração da rede como um Banco de Dados;
- Máquina Virtual: a rede é uma coleção de códigos interpretáveis na qual executa programas ou scripts;
- Adaptativo: o foco principal é sobre adaptabilidade.

Tabela 3 - Classificação de sistemas de *middleware*. (Adaptado de [MAR 05a])

Clássica	Centrado em Dados	Máquina Virtual	Adaptativa
Impala	Cougar	Agilla	MiLAN
Mires	Sina	Maté	TinyCubus

3.5.2.1 Abordagem Clássica

O *middleware* Impala foi especialmente projetado como parte do projeto ZebraNet, um projeto para monitorar a vida de animais selvagens. Sensores são alocados em um grupo de animais no *hábitat* (deles) para realizar o estudo migratório por um longo período. A arquitetura do *middleware* provê modularidade, adaptatividade e mecanismos capazes de reparar o mau funcionamento de aplicações na RSSF [JUA 02]. A Figura 3.1 mostra a arquitetura do Impala. Conforme a arquitetura, na camada superior está contida todos os protocolos da aplicação, já a camada subjacente possui serviços, representados por agentes.

O Adaptador de Aplicação modifica o protocolo de aplicação de acordo com o contexto de execução para alcançar melhorias de desempenho, eficiência em energia e robustez. O Atualizador de Aplicação realiza atualização de software automaticamente, ele recebe e propaga software atualizado por meio do transceptor *wireless* e instala-os sobre os nós sensores. O Filtro de Eventos captura e despacha eventos para as unidades do sistema adjacente e inicia o processamento. Impala tem cinco tipos de eventos. Evento de Tempo sinaliza que um determinado prazo de tempo terminou. Evento de Pacote sinaliza que chegou um pacote da rede. Evento de Envio Realizado sinaliza que um pacote foi enviado ou ocorreu falha no envio. Evento de Dado sinaliza que um dado do sensor está pronto para ser lido. Evento de Dispositivo sinaliza que uma falha no dispositivo foi detectada. Quando múltiplos eventos chegam ao mesmo tempo, eles são pré-processados sequencialmente. Isso elimina a complexidade da programação de sincronizar entre diferentes tratadores de eventos. As Aplicações, o Adaptador e o Atualizador de Aplicações são todos programados em um conjunto de manipuladores de eventos que são invocados pelo Filtro de Eventos quando os eventos associados são recebidos. As aplicações devem implementar quatro manipuladores de eventos: Manipulador de Tempo, Manipulador de Pacote, Manipulador de Envio Realizado e Manipulador de Dado. As aplicações são requisitadas para implementar três outras rotinas: Consulta Aplicação, Termina Aplicação, e Inicializa Aplicação [LIU 03].

Impala propõe uma camada de *middleware* assíncrono baseado em eventos e usa programas modulares compilados em instruções binárias. Em Impala a adaptação é implementada através do modelo de programação baseada em eventos, que ocorre em respostas a variados eventos. Alguns eventos como, Evento de Tempo, sinaliza que o tempo ultrapassou desde a última verificação de *status*, talvez o Adaptador de Aplicação então escolha perguntar sobre o *status* da aplicação ou do sistema para determinar se alguma

adaptação deve ser executada [LIU 03]. A chave para eficiência em energia no Impala é que, as aplicações nos nós sensores sejam tão modulares quanto possíveis, permitindo pequena atualização quando necessário, dessa forma demanda pouca energia para a transmissão do evento [MAS 07]. Um código de aplicação pode ser instalado e atualizado em sensores em tempo de execução. Impala é um modelo diferenciado por usar uma abordagem autonômica, entretanto, não tem suporte para fusão de dados e não suporta heterogeneidade em termos de plataforma de hardware [HEN 06] [MOL 06] [HAD 06].

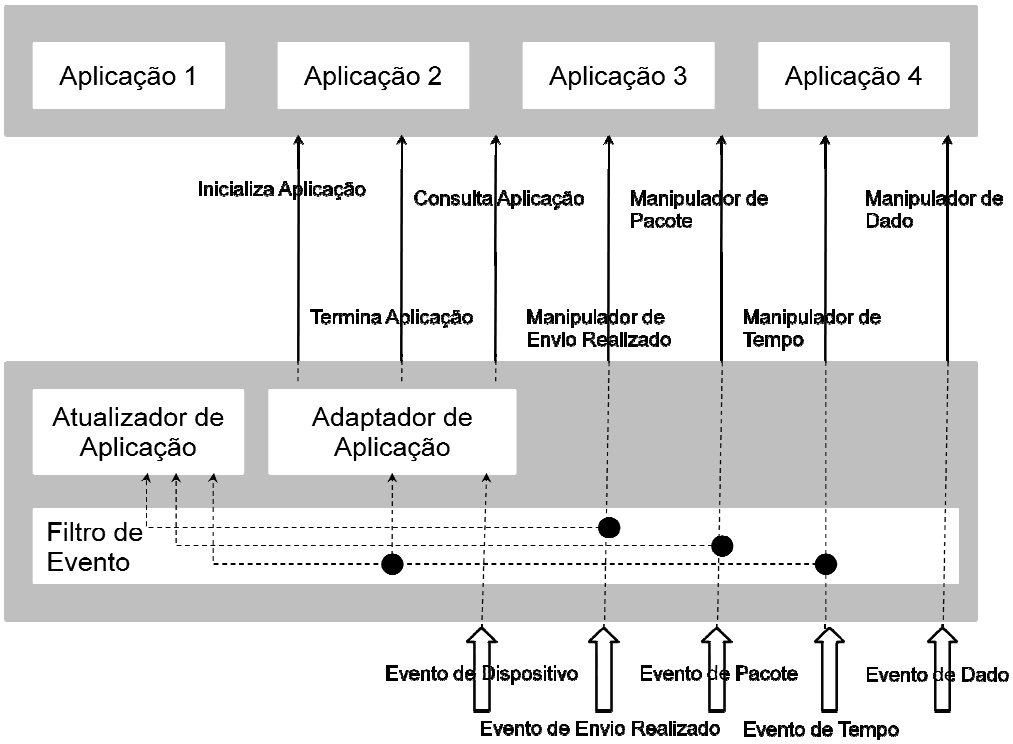

Figura 3.1 – Arquitetura do *middleware* Impala. (Adaptado de [LIU 03])

O Mires é um *middleware* para RSSF orientado a mensagem, baseado em serviços, que implementa o paradigma *publish/subscribe* que desacopla os produtores e consumidores de dados. O Mires estabelece uma estrutura hierárquica de roteamento para a RSSF, que viabiliza a comunicação com as aplicações clientes por meio do nó *sink* na RSSF. O *sink* é o gateway entre a RSSF e as aplicações clientes [SOU 05]. A Figura 3.2 mostra a arquitetura do Mires. O primeiro bloco contém os componentes de hardware dos nós sensores. Os componentes estão conectados e controlados pelo sistema operacional. O Mires está posicionado acima do sistema operacional, encapsulando seus detalhes internos e fornecendo serviços à aplicação do nó. O componente principal da arquitetura é o serviço de comunicação. Este serviço é responsável por: anunciar os tópicos fornecidos pela aplicação do nó, manter a lista dos tópicos assinados pela aplicação terminal, publicar as mensagens que

contêm os dados relacionados aos tópicos anunciados e intermediar a comunicação entre os demais serviços do *middleware* [GUI 06].

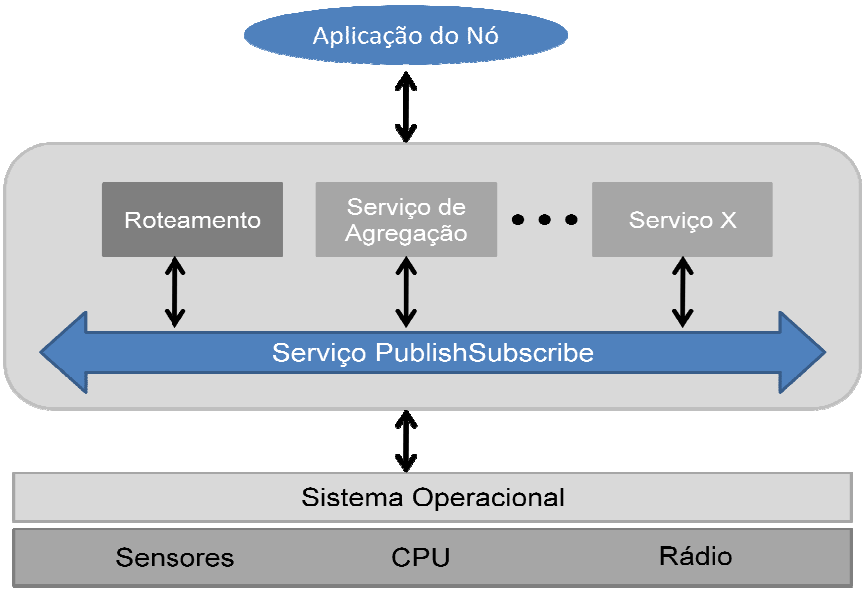

Figura 3.2 – Arquitetura do *middleware* Mires (Adaptado de [SOU 05])

O funcionamento do Mires segue os seguintes passos: inicialmente é realizada a fase de anúncio, os nós da rede de sensores criam mensagens de anúncio dos tópicos disponíveis, como temperatura e umidade que são obtidos dos nós locais. Em seguida, o componente *PublishSubscribe* encapsula a informação em uma mensagem que são roteadas até o nó *sink* usando um algoritmo de roteamento *multi-hop*. Após a fase de anúncio, uma aplicação do usuário conectado ao nó *sink*, assinala os tópicos de interesses disponibilizados na fase anterior e defini a política de agregação e o critério de parada para cada tópico interessado. As informações de configurações (os tópicos assinados pela aplicação do usuário, políticas de agregação e critérios de parada) contidas na mensagem de assinatura são enviadas para o nó na rede via *broadcast*. Após os nós na rede receberem a mensagem de assinatura, passam a publicar os dados coletados ao serviço *PublishSubscribe*, que através da hierarquia de roteamento passa a enviar as mensagens de volta a aplicação cliente através do nó *sink* [GUI 06].

A Figura 3.3 mostra as conexões entre os componentes do serviço de comunicação *PublishSubscribe* e outros elementos do Mires. O componente *PublishSubscribe* fornece as interfaces *Advertise* e *Publish* para a aplicação do nó sensor, as interfaces *PublishState* e *Notifier* aos serviços adicionais. Estas duas últimas interfaces permitem a adição de novos serviços ao Mires [GUI 06].

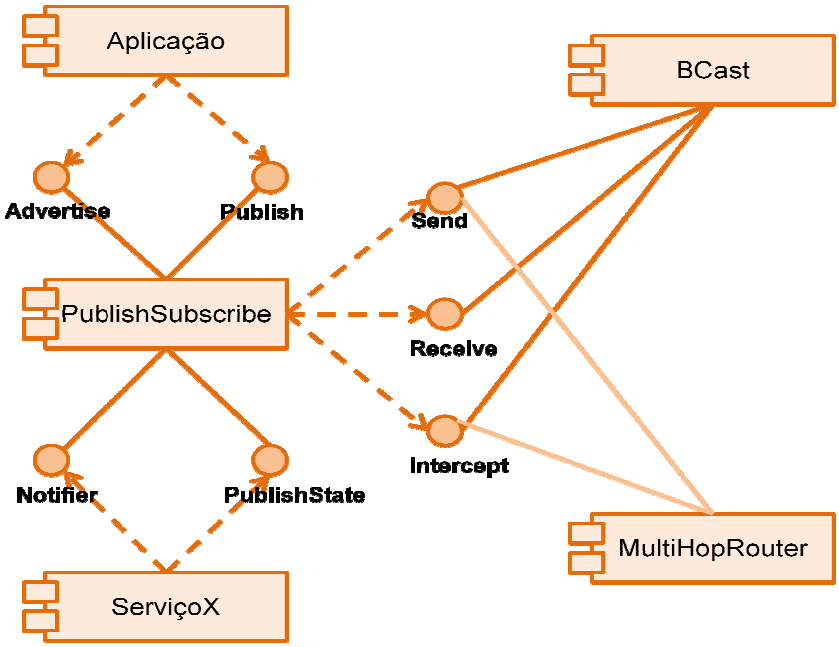

Figura 3.3 – Diagrama de componentes do Mires. (Adaptado de [GUI 06])

O componente *Publish/Subscribe* usa três interfaces: *send*, *receive* e *intercept*, do Sistema Operacional TinyOS e são realizadas por meio dos componentes de comunicação *Bcast* e *MultiHopHouter*, que também são do TinyOS. O componente *MultiHopHouter* é responsável por estabelecer a hierarquia de roteamento para o nó sorvedouro, já o componente *Bcast* transmite a informação de configuração via *broadcast* para a RSSF [GUI 06].

3.5.2.2 Abordagem Centrada em Dados

O *middleware* Cougar provê a rede de sensores como um sistema de banco de dados distribuído, cada nó sensor armazena dados percebidos e são conectados por meio de uma rede sem fio *multi-hop* [BON 01]. Cougar realiza consultas usando uma linguagem como SQL. Usuários e aplicações não sabem como os dados são gerados no interior da rede de sensores e como os dados são processados na resposta a consultas. A arquitetura fornece gerenciamento de catálogo, otimização de consulta e técnicas de processamento de consulta que abstrai o usuário dos detalhes de contatar relevantes nós sensores, processar e retornar dados do sensor. Para capacitar consultas declarativas para a rede de sensores, Cougar provê uma camada de consulta consistindo de um *Proxy* de Consulta sobre cada nó sensor. O *Proxy* de Consulta fica entre a camada de rede e a camada da aplicação, e fornece serviços de alto

nível por meio de consultas que são injetadas na rede de sensores por meio do nó *gateway* [YAO 02].

Um otimizador de consulta localizado sobre o nó *gateway* gera um plano de processamento de consulta distribuído após receber a solicitação de uma aplicação. O plano de consulta é criado de acordo com a informação de catálogo e a especificação da consulta. Tal plano de consulta especifica tanto o fluxo de dados entre os sensores, quanto o plano de computação exato para cada sensor. O plano é então disseminado para todos os sensores relevantes. Estruturas de controle são criadas para sincronizar o comportamento do sensor e a consulta é inicializada. Em tempo de execução os registros de dados fluem de volta para o nó *gateway* enquanto a computação acontece dentro da rede.

O otimizador de consulta aperfeiçoa uma consulta verificando a carga de trabalho atual e tenta fundir a nova consulta com outras similares já existentes. Assumindo que a consulta Q é apenas uma consulta que está rodando no interior da rede, o otimizador de consulta gera um novo plano de consulta QP. O plano de consulta QP especifica o nó líder dessa consulta. Designa um nó que desempenha uma computação da temperatura média, por exemplo. O líder pode ser um nó fixo com maior recurso computacional e de energia, ou selecionado randomicamente, por algum algoritmo de eleição de líder distribuído. Dois planos de computação são produzidos, um para o nó líder, e um segundo plano para os restantes dos nós na região da consulta.

A Figura 3.4 mostra o plano de consulta para um nó não líder que participa de uma consulta. Nós não líder têm um operador *scan* para ler os valores do sensor periodicamente e enviá-los para o nó líder. Adicionalmente, o plano de consulta deles contém um operador de agregação para agregar os dados de outros sensores.

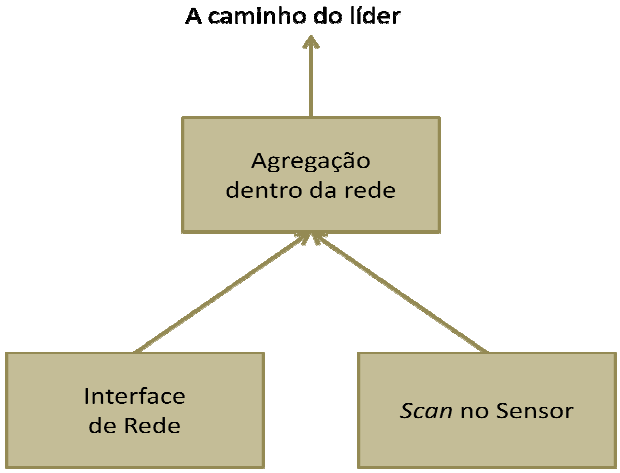

Figura 3.4 – Plano de consulta para um nó não líder. (Adaptado de [YAO 02])

A Figura 3.5 mostra o plano de consulta para o nó líder, que contém um plano de consulta, no exemplo abaixo, um operador AVG para computar o valor médio de temperatura de leituras recebidas de todos os sensores, e um operador SELECT que checa se os resultados estão acima de um limiar.

Quando a consulta começa, os planos de consultas são disseminados para o *Proxy* de Consulta de todos os sensores relevantes. O *Proxy* de Consulta registra a consulta, cria uma árvore de operação local, ativa os sensores relevantes, e retornam os registros de acordo com a especificação do plano de consulta. O nó líder gera um registro apenas se a temperatura média está acima do limiar definido pelo usuário.

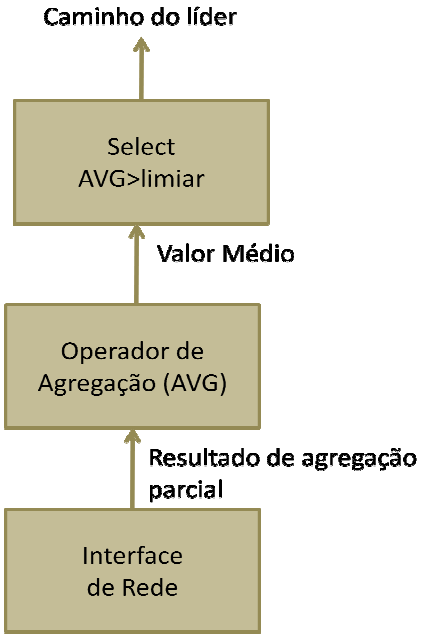

Figura 3.5 – Plano de consulta para o líder. (Adaptado de [YAO 02])

SINA [SRI 00] é um *middleware* para RSSF que executa consultas, monitoramento, e permite enviar tarefas a pedidos de aplicações para a rede de sensores. A arquitetura possui um *kernel* que fornece um conjunto de primitivas de comunicação e configuração, que capacitam interação entre os objetos de sensores e solução escalável de forma organizada, robusta e eficiente em energia. Para suportar operações escaláveis e eficientes em energia, sensores são autonomicamente agrupados em cluster. No topo do *kernel* tem um substrato programável que segue o paradigma *spreadsheet* [KAL 97] e fornece mecanismos para criar associações entre sensores. Usuários acessam informações em uma rede de sensores usando consultas declarativas e efetuam tarefas usando *scripts* programáveis.

SINA provê suporte a consultas e monitoramento na rede de sensores, por meio de uma estrutura de dados que está contido dentro dos nós sensores baseado no

paradigma *spreadsheet*. Um *spreadsheet* fornece uma abstração que permite que as informações sejam organizadas e acessadas de acordo com as necessidades especificas das aplicações. SINA incorpora um mecanismo de *cluster* hierárquico e esquemas de nome baseado em atributos baseado sobre *Broadcast* Associativo [BAY 99]. *Broadcast* Associativo é um mecanismo que facilita a interação de processos com base em atributos de nomes. Uma mensagem enviada por um nó contém um Seletor que é uma expressão proposicional sobre atributos. Quando um nó recebe uma mensagem, ele checa se o Seletor corresponde ao seu perfil local. Se sim, o nó então recebe a mensagem, caso ao contrário, a mensagem é simplesmente descartada.

No paradigma *spreadsheet* utilizado por SINA, cada nó sensor em uma rede mantém um planilha lógica, contendo um conjunto de células, onde cada célula é exclusivamente nomeada e representa um atributo do nó. Dados armazenados em uma célula podem ter valor único, ou valor múltiplo. As redes de sensores podem ser vista como uma coleção de planilhas lógicas como mostra a Figura 3.6.

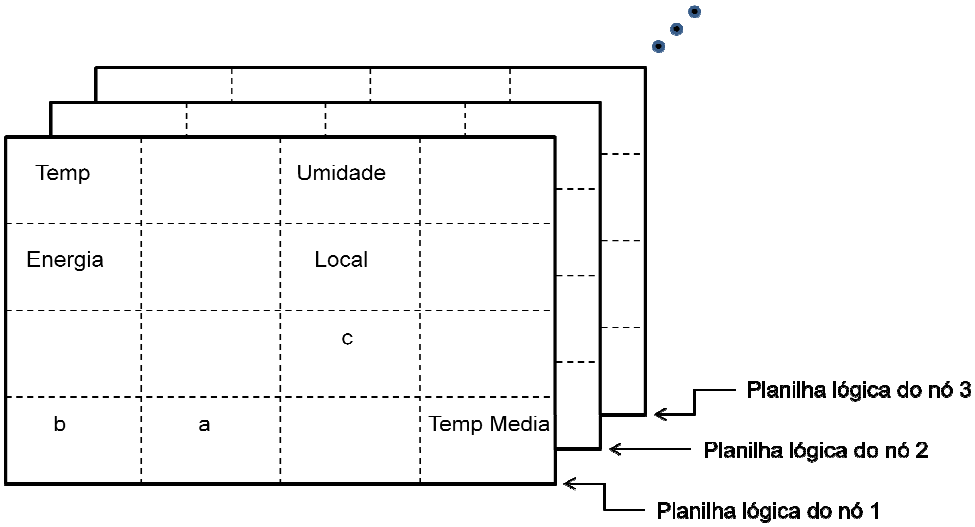

Figura 3.6 – Visão lógica da estrutura *spreadsheet*. (Adaptado de [SRI 00])

SINA contém algumas diferenças em comparação com algumas aplicações populares. Em Excel, por exemplo, uma célula é um endereço de acordo com as coordenadas lógicas x-y, porém em SINA não existe correspondência baseado em *grid*. Relacionamento entre células são definidas com base em atributos de nomes e estabelecidas via Broadcast Associativo. Por exemplo, o nome (localização=N-E, temperatura > 90) seleciona todos os nós sensores localizado dentro do quadrante Norte-Leste com uma leitura de temperatura maior do que 90 graus.

Em SINA, inicialmente a planilha lógica sobre cada nó está vazio. Um nó cria uma nova célula quando ele recebe uma requisição de outro nó, particularmente do usuário ou o *cluster head* dele. Uma requisição consiste de uma frase como SQL, encapsulada no interior de uma estrutura como KQML chamada SQTL (*Sensor Query and Tasking Language*). O valor de uma célula pode ser obtido de várias formas: referindo-se diretamente de uma ou mais células, invocando uma função definido pelo sistema, construindo uma expressão de função definida pelo sistema ou operações aritméticas, e agregando dados de outros *datasheets* invocando uma função de agregação.

A Figura 3.7 mostra como uma célula é definida como agregação de outras células localizadas em *datasheets* diferentes. SINA utiliza SQTL que foi originalmente projetado para ser uma ferramenta simples para efetuar tarefas na rede de sensores. Um *script* SQL injetado na rede de sensores é encapsulado em um *wrapper* SQTL. Assim, ambos SQL e SQTL podem ser alternativamente desenvolvidos sobre SINA com o objetivo de obter informações dos nós sensores.

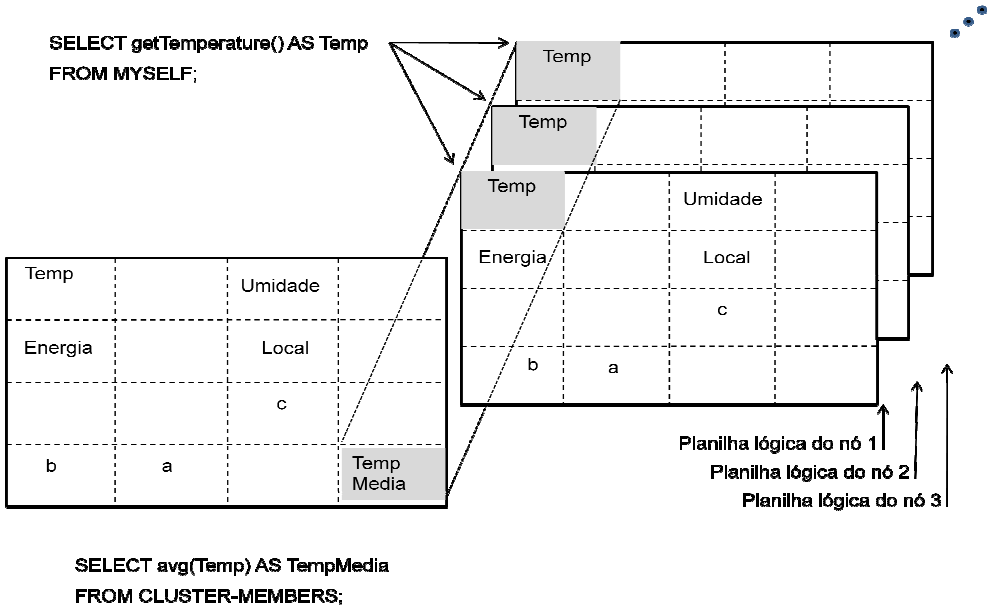

Figura 3.7 – Visão lógica de uma célula definida como agregação de outras células. (Adaptado de [SRI 00])

3.5.2.3 Abordagem de Máquina Virtual

Agilla [FOK 05] é um *middleware* que aumenta a flexibilidade da RSSF enquanto simplifica o desenvolvimento de aplicações. Uma rede Agilla é posicionada com

nenhuma aplicação pré-instalada. Usuários injetam agentes móveis na rede desempenhando tarefas especificas das aplicações. Os agentes móveis são autônomos, permitindo múltiplas aplicações compartilharem uma rede. *Tuple Spaces* (Espaço de Tuplas) são usadas para comunicação inter agentes e descoberta de contexto.

O modelo de Agilla é ilustrado na Figura 3.8. Cada nó suporta múltiplos agentes e mantém uma lista de espaço de tuplas e vizinhos. O espaço de tuplas é local e é compartilhado por agentes residindo sobre o nó. Instruções especiais permitem que agentes acessem remotamente o espaço de tuplas de outro nó. A lista de vizinho contém o endereço de todos nós de profundidade de um *hop*. Agentes podem migrar carregando o código e estado deles, mas não o próprio espaço de tuplas.

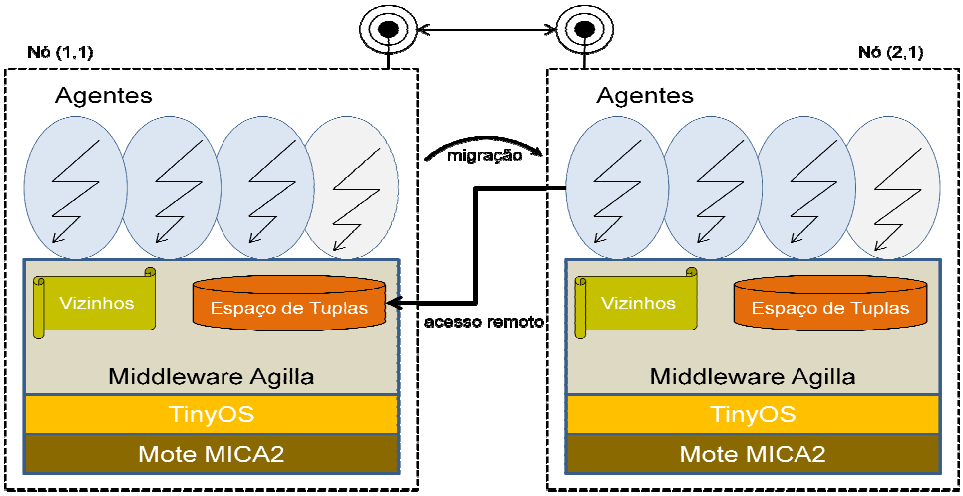

Figura 3.8 – O modelo de Agilla. (Adaptado de [FOK 05])

A arquitetura de Agilla é dividida em três camadas conforme mostra a Figura 3.9. A camada do meio contém o núcleo dos componentes de Agilla, enquanto abaixo está o TinyOS, na camada mais superior contém os agentes móveis e fixos. O Gerenciador de Agentes mantém cada contexto do agente. Ele é responsável por alocar memória para um agente quando chega e desalocar quando acaba a execução do agente. Esse componente é também responsável por determinar quando um agente está pronto para rodar, e notifica o motor de Agilla quando isso ocorre. Já o Gerenciador de Contexto determina a localização bem como a de seus vizinhos. Ele descobre os vizinhos e armazena a localização dos vizinhos em uma lista de conhecidos que é acessível para um agente via instruções especiais.

O Gerenciador de Instrução é fornecido para prover a alocação dinâmica de memória que o TinyOS não suporta. O componente Gerenciador de Espaço de Tuplas implementa todas as operações de espaço de tuplas não bloqueantes e reações, além disso, gerencia o conteúdo do espaço de tuplas local e registros de reação. O espaço de tuplas

gerencia dinamicamente alocação de memória para cada tupla. O Gerenciador de Espaço de Tuplas associa os registros de reações para cada agente armazenando eles em um registro. Se uma tupla é inserida, ele checa o registro para uma reação. Se a nova tupla associa a um modelo de reação o Gerenciador de Espaço de Tuplas notifica o Gerenciador de Agentes, que atualiza o contador no programa do agente para executar o código de reação. Por fim, o Motor de Agilla serve como um *kernel* de máquina virtual que controla a execução concorrente de todos os agentes sobre um nó. Ele implementa uma simples política de escalonamento *round-robin*, onde cada nó pode executar um número fixo de instruções antes de mudar o contexto.

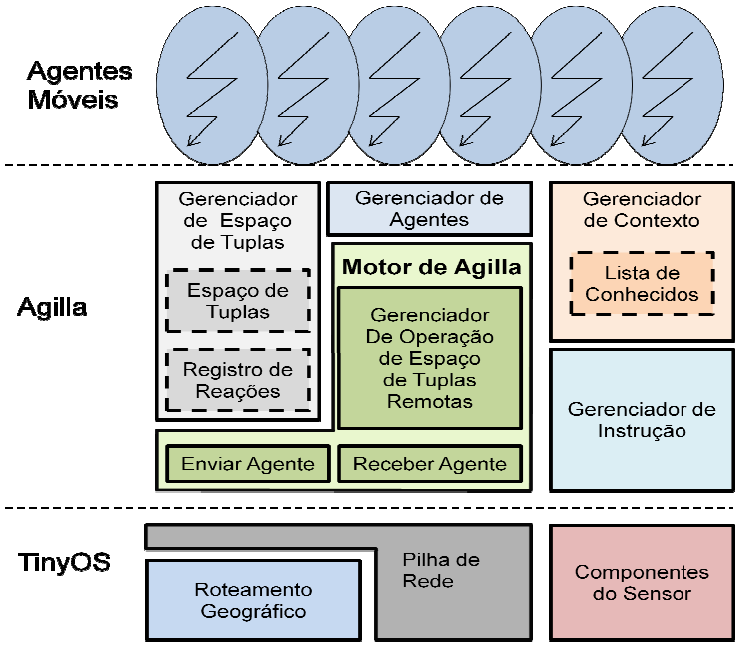

Figura 3.9 – Arquitetura do *middleware* Agilla. (Adaptado de [FOK 05])

Outra solução famosa nessa categoria é Maté [LEV 02] uma pequena máquina virtual centrada em comunicação para as redes de sensores. A interface de alto nível permite que programas complexos sejam instalados na rede de sensores com um tamanho de bytes bem reduzido, melhorando o custo de energia ao transmitir novos programas. O código é dividido em pequenas cápsulas de 24 instruções, e grandes programas podem ser decompostos em múltiplas cápsulas que automaticamente se replicam por meio da rede. Pacotes de envio e de recepção de cápsulas capacitam o roteamento *ad-hoc* e algoritmos de agregação de dados. Representações de programas de alto nível simplificam a programação e permite que grandes redes sejam frequentemente reprogramadas em uma forma eficiente em energia.

Maté é um interpretador que executa sobre TinyOS. As cápsulas geradas contêm identificadores e informação de versão. O modelo de comunicação da arquitetura

Maté permite que um programa envie uma mensagem com uma única instrução, e a mensagem enviada é roteada automaticamente para o seu destino. As instruções de Maté escondem o assíncronismo da programação de TinyOS, quando uma instrução *send* é solicitada, Maté chama um comando no componente de roteamento *ad-hoc* para enviar um pacote. Maté então suspende o contexto até que um evento completo de mensagem enviada seja recebido, resumindo a execução. Assim, Maté não necessita gerenciar *buffers* de mensagens. De forma similar, quando uma instrução *sense* é solicitada, Maté requisita dados do sensor com o componente do TinyOS e suspende o contexto até o componente retornar os dados de um evento.

A arquitetura de Maté tem três contextos de execução conforme mostra a Figura 3.10 e corresponde a três eventos: tempo de *clock*, recepção de mensagens, e requisição de envio de mensagens. Cada contexto tem duas pilhas, uma pilha de operando e uma pilha de endereço de retorno. O primeiro é usado para todas as instruções de tratamento de dados, enquanto o segundo é usado para chamadas de sub-rotinas.

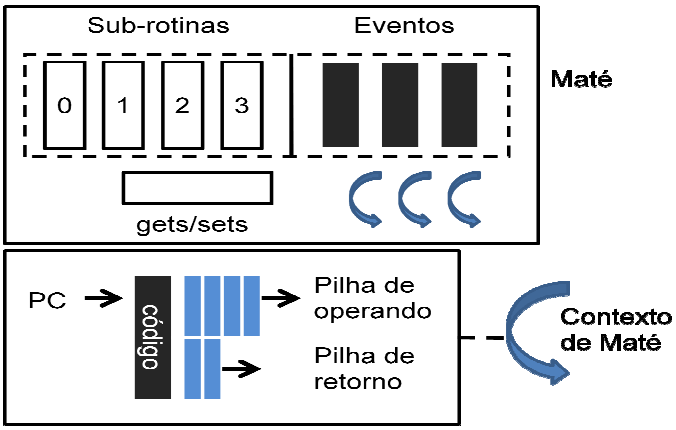

Figura 3.10 – Arquitetura de Maté. (Adaptado de [LEV 02])

3.5.2.4 Abordagem Adaptativa

MiLAN (*Middleware Linking Applications and Networks*) [HEI 04] é um *middleware* que pode ser utilizado nas RSSFs, e usa informações de QoS dos sensores. MiLAN permite que as aplicações para redes de sensores especifiquem as suas necessidades de qualidades de serviços e ajusta as características da rede para aumentar o tempo de vida da rede e aplicação enquanto satisfaz as necessidades de QoS. MiLAN recebe informações:

- de aplicações individuais sobre os requisitos de QoS ao longo do tempo, e como encontrar os requisitos de QoS deles usando diferentes combinações de sensores;
- do sistema geral sobre a importância relativa das diferentes aplicações; e
- da rede sobre a disponibilidade de sensores e de recursos, tais como, nível de energia do sensor e largura de banda do canal.

A combinação dessas informações, fornece a MiLAN a possibilidade de continuamente adaptar as configurações da rede (por exemplo, especificar quais nós sensores devem enviar dados, quais nós sensores devem ser roteados em uma rede *multi-hop*, quais nós sensores devem executar funções especiais na rede) para realizar as necessidades das aplicações. A Figura 3.11 mostra um diagrama de alto nível de um sistema que emprega MiLAN.

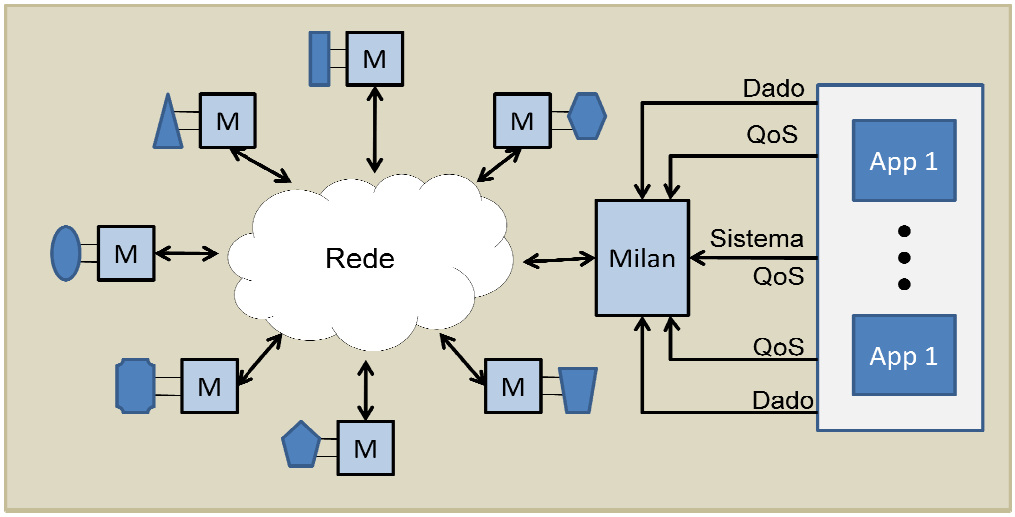

Figura 3.11 – Visão geral de MiLAN. (Adaptado de [HEI 04])

De forma geral MiLAN tem como objetivo: receber uma descrição dos requisitos da aplicação; monitorar as condições da rede; e otimizar as configurações de rede e sensor para maximizar o tempo de vida da rede e aplicação. Para alcançar esses objetivos, aplicações representam os requisitos para MiLAN por meio de grafos especializados que incorporam as mudanças baseado em estado em necessidades das aplicações. Baseado sobre as informações, MiLAN toma decisões sobre como controlar a rede assim como os sensores para balancear os requisitos de QoS da aplicação.

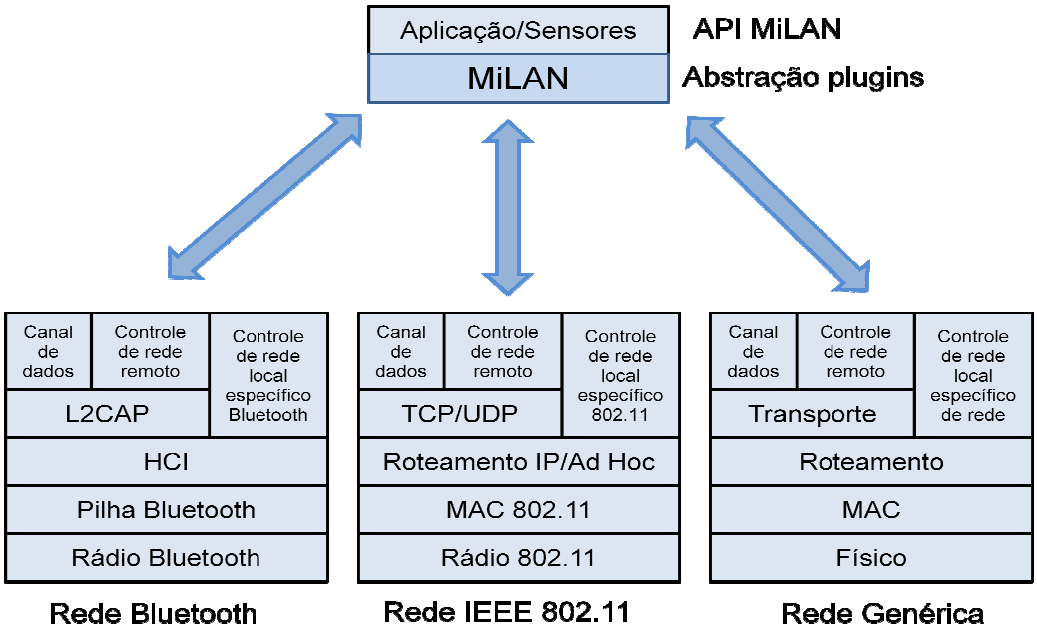

Figura 3.12 – Pilha de protocolos do MiLAN. (Adaptado de [HEI 04])

Diferente de sistemas de *middleware* tradicionais que situam entre e a aplicação e o sistema operacional, MiLAN tem uma arquitetura que estende a pilha de protocolos como mostra a Figura 3.12. MiLAN está situado no topo de múltiplas redes físicas. Uma camada de abstração é fornecida para permitir que *plugins* específicos para uma rede convertam comandos de MiLAN para comandos específicos do protocolo. Assim, MiLAN pode continuamente adaptar às características específicas de qualquer rede que esteja sendo usada para comunicação, para realizar as melhores necessidades das aplicações ao longo do tempo. Para determinar como melhor servir a aplicação, MiLAN deve conhecer:

- as variáveis de interesse da aplicação;
- a QOS requerida para cada variável;
- o nível de QOS que dados de cada sensor ou conjunto de sensores podem fornecer para cada variável.

Todas estas informações podem mudar baseado sobre o estado atual das aplicações. Durante a inicialização da aplicação, as informações são transportadas da aplicação para MiLAN por meio de grafos *State-based Variables Requirements* e do sensor para Milan por meio de grafos *Sensor QoS*. Dessa forma é possível especificar as variáveis e a QoS requerido para os possíveis estados da aplicação, e grafos de QoS que contém os sensores e o nível de QoS para cada variável. Para uma dada aplicação, a QoS para cada variável pode ser satisfeita usando dados de um ou mais sensores. A aplicação especifica estas informações para MiLAN por meio do grafo de QoS do Sensor. Dado todas as informações

desses grafos assim como o estado atual da aplicação, MiLAN pode determinar quais conjuntos de sensores satisfazem todos os requisitos de QoS para cada variável.

TinyCubus [MAR 05b] é um *framework* desenvolvido para suportar a heterogeneidade dos requisitos das aplicações. Os requisitos das aplicações são esperados que mudem, dessa forma, desenvolvimento, distribuição e otimização de aplicações para redes de sensores sem fio é uma tarefa extremamente difícil. O TinyCubus foi projetado para criar um *framework* reconfigurável genérico para redes de sensores. Como mostra a Figura 3.13, TinyCubus é implementado no topo do TinyOS. Para TinyOS, TinyCubus é apenas uma aplicação rodando no sistema. Todas outras aplicações registram seus requisitos com TinyCubus e são executados pelo *framework*.

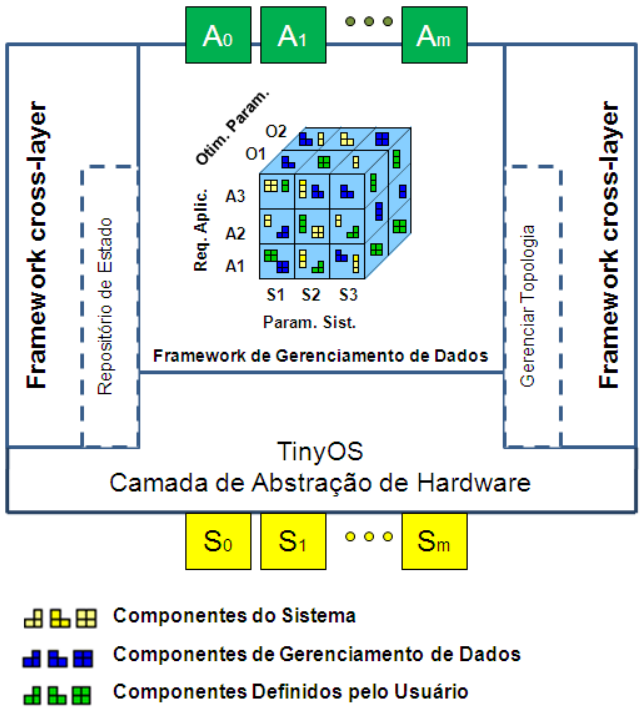

Figura 3.13 – Componentes arquiteturais em TinyCubus. (Adaptado de [HEI 04])

O TinyCubus consiste de um *framework* de gerenciamento de dados, um *framework cross-layer* e um motor de configuração.

O *framework* de gerenciamento de dados fornece um conjunto de componentes padrões de gerenciamento de dados e de sistema. Ele combina parâmetros de otimizações, tais como energia, latência de comunicação, e largura de banda; requisitos das aplicações, tais como confiabilidade; e parâmetros do sistema, tais como mobilidade. O *framework* de gerenciamento de dados seleciona o mais adequado conjunto de componentes baseado sobre os parâmetros atuais do sistema, requisitos da aplicação e parâmetros de otimizações. A

adaptação é executada durante o tempo de vida do sistema como uma parte crucial do processo de otimização.

O *framework cross-layer* suporta o compartilhamento de dados e outras formas de interação entre componentes para alcançar otimizações *cross-layer*. Ele usa uma específica linguagem que permite a descrição de tipos de dados e informações requeridas e fornecidas por cada componente. Esses tipos de dados são armazenados no repositório de estado. Para lidar com *callbacks* e carregar código dinamicamente, TinyCubus estende as funcionalidades fornecidas pelo TinyOS para resolução de interfaces e componentes por meio de *callbacks*. O conceito de *callbacks* do TinyOS não é sofisticado o bastante para o propósito de TinyCubus, pois os seus componentes são definidos de forma estática. Com o TinyCubus os componentes são selecionados dinamicamente e podem mudar em tempo de execução.

O motor de configuração permite que código seja distribuído de forma confiável e eficiente levando em conta a topologia e as funcionalidades dos nós sensores. Em alguns casos de parametrização, como fornecido pelo *framework cross-layer*, não é o bastante. Instalar novos componentes, ou trocar certas funções é necessário, por exemplo, quando novas funcionalidades tais como novas funções de processamento ou agregação para dados percebidos são requeridas pela aplicação, o motor de configuração resolve esses problemas, gerenciando a topologia e com a utilização de um algoritmo baseado em regras.

3.6 Comparação Entre os Projetos de Middleware para RSSFs

De acordo com a classificação realizada na seção de *middleware* para RSSFs, as abordagens podem ser comparadas a partir dos desafios que apresentam às RSSFs. A Tabela 4, compara e resume os principais projetos relatados neste trabalho.

Tabela 4 - Comparação de alguns dos *middlewares para RSSF*. (Adaptado de [MOL 06])

Desafios	Impala	Mires	SINA	Agilla	Maté	MiLAN	TinyCubus
Abstração	✓	✓	✓	✓	✓		✓
Fusão/Agregação de Dados	✓	✓	✓	✓			
Restrições de Recursos	✓	✓	✓	✓	✓	✓	✓

Topologia Dinâmica	✓		✓	✓			✓
Conhecimento da Aplicação						✓	✓
Paradigma de Programação	✓	✓	✓	✓	✓	✓	✓
Adaptabilidade	✓			✓	✓	✓	✓
Escalabilidade	✓	✓	✓	✓		✓	
Segurança					✓		
QoS						✓	
Integração				✓			

Como pode ser visto na Tabela 4, segurança e QoS são as duas características mais ignoradas pelos projetos. Todos os *middlewares* descritos tentam resolver os desafios de restrições de recursos, mas a abordagem deles para resolver esse desafio não é escalável e adaptável. Conhecimento da aplicação é também ignorado pela maioria dos *middlewares*.

Outra característica desejável e não mencionada é a integração da RSSF com outras existentes infra-estruturas de redes, por exemplo, a Internet. A abordagem orientada a serviços para implementar integração a RSSF é baseado sobre tecnologias arquiteturais de padrões aberto tal como os Serviços Web. A maioria dos serviços de integração está ainda em um estágio preliminar.

O projeto de uma arquitetura de *middleware* é altamente influenciado pelos requisitos das aplicações, assim como a infra-estrutura de rede. Assim, o tipo de *middleware* a ser utilizado é dito como ótimo de acordo com as características de aplicações específicas, e das características do ambiente.

Dois estágios relativos às pesquisas e à tecnologia são considerados em [BAR 05] sobre a evolução de soluções em RSSFs, que acabaram impulsionando o interesse sobre arquiteturas de softwares, mais especificamente *middleware*. No primeiro estágio, a maioria das soluções propostas desenvolvidas em várias partes do mundo evidenciava a possibilidade do uso das RSSFs em diversas aplicações, ou seja, a aplicabilidade do desenvolvimento focava em construção de plataformas para aplicações específicas, se preocupando com o consumo de energia. Os nós sensores seriam operados por baterias não-recarregáveis e que o consumo excessivo, principalmente para aplicações de difícil manutenção, seria o maior vilão

contra a sobrevida da rede. A partir dessa premissa o *software* e principalmente o *hardware* deveriam ser adaptados.

Atualmente a busca por plataformas mais flexíveis leva a discussões a respeito das arquiteturas tanto do *hardware* quanto a do *software*. A possibilidade que os avanços promovidos pela microeletrônica, podem levar a eliminação da necessidade do uso de baterias já que, ao ultrapassar a barreira dos 100uW (*MicroWatts)* teoricamente os nós sensores poderiam ser alimentados pela energia extraída do ambiente operacional. Quanto ao *software*, as discussões caminham na direção da padronização e da flexibilidade como forma de garantir e aumentar a portabilidade das plataformas junto à diversidade de aplicações, assim como ocorreu com as redes de computadores [BAR 05]. Grandes são os esforços no desenvolvimento de arquiteturas de software para RSSFs. Espera-se que com o suporte de *middleware* e a inclusão de outras áreas do conhecimento, soluções padronizadas, reusáveis e flexíveis possam ajudar a resolver os desafios impostos pelas RSSFs.

3.7 Considerações Finais

Neste capítulo foram realizados revisões e comparações sobre os principais sistemas operacionais para RSSFs.

Foi descrito neste capítulo que sistemas de *middleware* para redes tradicionais não podem ser aplicadas diretamente nas RSSFs devido às características particulares desse tipo de rede.

Os princípios de projeto, desafios, e classificações de *middleware* para as RSSFs com base no nível de abstração das soluções propostas foram descritos.

A maioria dos *middleware* desenvolvidos para as RSSFs cobre parcialmente alguns dos desafios impostos, devido aos requisitos particulares das aplicações. Foi realizada uma comparação das soluções dos *middlewares* levando em conta os desafios citados, e uma reflexão sobre a evolução das pesquisas e tecnologias realizadas na área.

No próximo capítulo será apresentado o conceito e comparação de alguns paradigmas de comunicação em sistemas distribuídos.

4. Paradigmas de Comunicação de Suporte a Sistemas Distribuídos

4.1 Considerações Iniciais

O desacoplamento entre produtores e consumidores da informação, aumenta a escalabilidade, removendo todas as dependências explícitas entre os participantes da interação. Os paradigmas de comunicação possuem particularidades em relação ao desacoplamento, à escolha de um determinado paradigma depende do tipo de aplicação em um sistema distribuído. Este capítulo descreve os principais paradigmas de suporte a sistemas distribuídos.

Neste capítulo serão realizadas revisões de conceitos sobre os paradigmas: passagem de mensagem, notificação, fila de mensagem, *publish/subscribe*, espaço compartilhado e invocação remota. Uma discussão sobre os principais paradigmas de comunicação é também realizada, seguida de considerações finais.

4.2 Paradigmas de Comunicação de Suporte a SDs

Passagem de mensagem, notificação, fila de mensagem, invocação remota e espaço compartilhado, constituem paradigmas de comunicação alternativos ao esquema *publish/subscribe*. Eles possuem diferentes níveis de abstração e uma forma de compará-los, com o *publish/subscribe* é enfatizar a capacidade de desacoplamento que cada paradigma provê na comunicação entre os participantes.

Nas próximas subseções são realizadas revisão de conceitos, com ênfase no desacoplamento que cada paradigma de comunicação fornece, a saber: espaço, tempo e fluxo.

4.2.1 Passagem de Mensagem

Passagem de mensagem pode ser visto como um antecessor de interações distribuídas. Este paradigma representa uma forma de baixo nível de comunicação distribuída, na qual participantes comunicam-se simplesmente enviando e recebendo mensagens.

Passagem de Mensagem é assíncrono para o produtor, enquanto o consumo da mensagem é geralmente síncrono. O produtor e consumidor são unidos no tempo e no espaço, eles devem estar ativos no mesmo tempo e o destinatário de uma mensagem é conhecido pelo produtor [EUG 03].

4.2.2 Notificação

A fim de alcançar o desacoplamento da sincronização, uma invocação remota síncrona é por vezes dividida em duas invocações assíncronas: um cliente envia para o servidor, acompanhado pela invocação de argumentos e uma chamada de referência para o cliente; o servidor retorna para o cliente o retorno da resposta.

Esse esquema pode ser estendido para retornar várias replicas, pois o servidor tem que fazer vários *callbacks* para o cliente. Tais interações baseado em notificação são amplamente usados para assegurar a consistência de *caches Web*, após o *download* do conteúdo *Web, proxies Web* recebem uma garantia de serem notificados se alguma mudança ocorrer no Servidor Web. Isto implementa uma limitada forma de interação *publish/subscribe*, na qual os *proxies Web* atuam como assinantes e o servidor Web como publicadores [EUG 03].

Esse tipo de interação, em que assinantes registram seus interesses diretamente com produtores, na qual gerencia subscrições e enviam eventos, corresponde ao chamado padrão de projeto do observador. Isto é geralmente implementado, usando invocações assíncronas, no intuito de realizar o desacoplamento síncrono. Embora produtores notifiquem os assinantes de forma assíncrona, ambos permanecem acoplados no tempo e no espaço. Além disso, o gerenciamento da comunicação é deixado ao critério do produtor e pode ficar com uma carga de trabalho grande com o sistema crescendo em tamanho [EUG 03].

4.2.3 Fila de Mensagem

Fila de mensagem e *publish/subscribe* são estreitamente interligados: sistemas de fila de mensagem normalmente integram alguma forma de interação, como o, *publish/subscribe*. Tais abordagens, centrada em mensagem são frequentemente referidas como *Middleware* Orientado a Mensagem (MOM) [EUG 03].

Em nível de interação, fila de mensagem lembra o espaço de tuplas: filas podem ser vista como espaço global, na qual são adicionados com mensagens de produtores. Em sistemas de fila de mensagem, mensagens são concorrentemente consumidas na forma 1 de N, semântica similar daquelas oferecidas por espaços de tuplas através de operações *in()*. Estes modelos de interação são frequentemente referenciados como fila Ponto-a-Ponto (PTP). No qual, o elemento retornado por um consumidor, não é definido por estruturas de elementos, mas pela ordem nas quais os elementos são armazenados na fila (geralmente FIFO ou ordenação baseado em prioridade) [EUG 03].

De forma similar ao espaço de tuplas, produtores e consumidores são desacoplados no tempo e espaço. Como consumidores sincronicamente consomem mensagens, fila de mensagem não fornece desacoplamento de sincronização. Alguns sistemas de fila de mensagem oferecem suporte limitado para entrega de mensagens assíncronas, mas os mecanismos assíncronos não escalam bem para grandes populações de consumidores, por causa das interações necessárias para manter transacional, o tempo e a garantia de ordenação [EUG 03].

4.2.4 Invocação Remota

Uma das formas de interação distribuída amplamente usada é a invocação remota, uma extensão do conceito de invocação de operação para um contexto distribuído. Esse tipo de interação tem sido proposto na forma de RPC (Chamada de Procedimento Remoto). Essa interação acabou sendo aplicada para contextos de orientação a objetos na forma de invocação de método remoto, por exemplo, em Java RMI, CORBA, Microsoft DCOM.

Interações remotas “aparecem” da mesma forma como interações locais. O modelo RPC e suas derivações tornam a programação distribuída bastante fácil, isto explica sua popularidade na computação distribuída. Entretanto, distribuição não pode ser feita completamente transparente para a aplicação, pois pode dar origem para potenciais falhas como, falhas na comunicação, que têm de ser tratada explicitamente.

RPC difere-se, por exemplo, do paradigma *publish/subscribe* em termos de acoplamento: a natureza síncrona do RPC apresenta um forte acoplamento no tempo; a sincronização (no lado do consumidor); e também um acoplamento espacial, já que uma invocação de objeto possui uma referência remota para o invocador, ou seja, as partes necessitam conhecer um ao outro [EUG 03]. Nas próximas subseções são descritas algumas tecnologias derivadas do RPC.

4.2.4.1 CORBA (Commom Object Broker Architecture)

A arquitetura CORBA (*Commom Object Request Broker Architecture*) foi definida pela OMG em 1989. A OMG (*Object Management Group*) é uma organização internacional suportada por centenas de membros, que abrange, desde usuários até projetistas de sistemas. O principal objetivo é alcançar sistemas baseados em objetos em ambientes distribuídos heterogêneos com características de reusabilidade, portabilidade e interoperabilidade [MON 97].

Objetos clientes requisitam serviços por meio de um ORB (*Object Request Broker*). O ORB é um componente intermediário responsável por todos os mecanismos requeridos para encontrar o objeto, e preparar a implementação do objeto para receber e executar uma requisição. O cliente vê a requisição de forma independente de onde o objeto está localizado, da linguagem de programação na qual ele foi implementado, ou qualquer outro aspecto que não está refletido na interface do objeto. Para a definição de interfaces, CORBA especifica uma linguagem denominada OMG IDL, ou seja, especifica um contrato entre os objetos. A OMG IDL é uma linguagem puramente declarativa baseada em C++. Isso garante que os componentes em CORBA sejam auto-documentáveis, permitindo que diferentes objetos, escritos em diferentes linguagens, tais como C, C++, JAVA, Smaltalk e COBOL, possam inter-operar por meio das redes e de sistemas operacionais [CAL 04].

O ORB, por si só, não executa todas as tarefas necessárias para os objetos inter-operarem. Ele só fornece os mecanismos básicos. Outros serviços necessários são oferecidos por objetos com interface IDL, o qual a OMG vem padronizando para os objetos de aplicação poder utilizar:

- Serviço de nomes.
- Serviço de controle de concorrência.
- Serviço de eventos.
- Serviço de tempo.

4.2.4.2 JAVA RMI (Java Remoted Method invocation)

RMI é uma tecnologia de programação de objetos distribuídos que é parte integrante da plataforma Java. RMI permite invocações de métodos entre JVM (*Java Virtual Machine*) localizado em nós diferentes de uma rede. Java RMI é um pacote Java que permite criar objetos cujos métodos podem ser invocados remotamente a partir de aplicações clientes, localizadas em JVM remotas. Uma aplicação cliente em RMI tipicamente consulta o serviço de nomes do sistema, chamado RMI Registry, a fim de obter uma referência de rede, isto é, uma referência para um objeto localizado em outra JVM. Essa referência pode ser então utilizada para invocar métodos do objeto remoto, com uma sintaxe equivalente àquela de chamadas locais [CAL 04].

Aplicações servidoras tipicamente são responsáveis por criar objetos remotos e, eventualmente, registrá-los no serviço de nomes do sistema. Em Java RMI a classe de um objeto remoto deve estender uma classe pré-definida, chamada java.rmi.server.UnicastRemoteObject. Além disso, essa classe deve implementar uma interface remota, isto é, uma interface que estende java.rmi.Remote. Uma interface remota define um contrato entre cliente e servidor, especificando métodos que o cliente pode invocar no servidor. Por fim, métodos de uma interface remota devem relacionar em sua cláusula *throws* exceções do tipo java.rmi.RemoteException. Em aplicações clientes, tais exceções são ativadas pela implementação de Java RMI para indicar uma falha de comunicação com a JVM servidora.

A classe java.rmi.Naming oferece métodos estáticos para registrar (*bind*) e pesquisar (*lookup*) objetos remotos associados ao RMI Registry. Ao registrar um objeto,

deve-se informar um nome textual para o mesmo. A escolha deste nome deve ser acordada com as aplicações clientes, já que estas precisam informá-lo ao pesquisar por um objeto remoto associado ao Registry de um determinado nó.

Internamente, a implementação de Java RMI se baseia em dois objetos principais, chamados de *stubs* e *skeletons*. Esses dois objetos são usados para implementar a abstração proporcionada por uma referência remota. Em uma aplicação cliente, uma referência de rede é, na realidade, uma referência para um objeto local, denominado *stub*. Esse objeto implementa a mesma interface remota do objeto servidor, sendo responsável por encapsular todos os detalhes de comunicação com o mesmo. Uma invocação remota é então "capturada" pelo *stub* do cliente, que converte seus parâmetros para uma representação seriada, a qual é enviada para um objeto da aplicação servidora chamado de *skeleton*. Esse objeto é responsável por recuperar os dados da chamada e invocar o método servidor. Após esse método ser executado, seu resultado é enviado de volta ao processo cliente, via objetos *skeleton* e *stub*. As classes de *stubs* e *skeletons* são geradas automaticamente a partir da classe do objeto servidor. Essa geração é realizada pela ferramenta *rmic*, a qual faz parte de ambientes de desenvolvimento de aplicações em Java [VAL 03].

4.2.4.3 DCOM (Distributed Component Object Model)

Em 1993, a Microsoft lançou uma nova tecnologia de encapsulamento de objetos designada COM (*Component Object Model*). O COM é disponibilizado pela Microsoft como um modelo de programação orientado a objetos com suporte a integração de várias aplicações diferentes, projetado para facilitar a interoperabilidade do software [AMO 04].

Um objeto no modelo COM é uma entidade funcional que obedece ao princípio de encapsulamento de orientação a objeto. Os clientes não manipulam os objetos diretamente. Ao invés disso, o objeto exporta para os seus clientes vários conjuntos de ponteiros de funções conhecidas como interfaces [AMO 04].

O DCOM é a extensão distribuída do COM que constrói uma camada de chamada de procedimentos de objetos remotos ORPC (*Object Remote Procedure Call*) no topo do DCE RPC (*Distributed Computing Environment Remote Procedure Call*) para suportar objetos remotos. O COM define como os componentes de software e os seus clientes

interagem. Essa interação é definida tal que o cliente e o componente possam se conectar sem a necessidade de qualquer recurso intermediário [AMO 04].

O DCOM define um padrão binário por plataforma sem a preocupação de definir ligações com linguagens de alto nível. Dessa forma, os clientes e desenvolvedores podem integrar componentes gerados com ferramentas de diversos fabricantes e usá-los em diferentes implementações *runtime* do DCOM. Com a distribuição do DCOM em todos os ambientes de desenvolvimentos da Microsoft (Visual C++, Visual Basic e Visual J++), ocasionou-se um aumento do número de linguagens e ferramentas que suportam esta tecnologia [AMO 04].

De forma geral essas tecnologias (CORBA, RMI, DCOM) resolvem muitos dos problemas de integração, porém ainda há barreiras que impedem seu pleno uso na WEB. Uma das principais barreiras é que tais tecnologias exigem que as aplicações adotem o uso de um modelo de programação específico, por exemplo, baseado em objetos. Por outro lado, diferente dessas tecnologias, o uso de Serviços Web (*Web Services*) não requer que, aplicações sejam desenvolvidas segundo um determinado modelo de programação, apenas requer o uso de um conjunto mínimo de protocolos de comunicação para permitir a integração de aplicações através da Web [MAS 02].

Tecnologias como CORBA, DCOM e RMI falham em prover mecanismos capazes de permitir que sistemas “conversem” entre si, pois essas tecnologias permitem apenas a “conversa natural” entre si, assim o processo de conversão de objetos entre eles se torna muito onerosa. Com o surgimento do XML (*eXtensible Markup Language*) tornou-se possível a utilização de um formato padrão para troca de informações independente de plataforma e linguagem. O resultado das pesquisas no desenvolvimento de protocolos baseados em XML foi à especificação XML-RPC (baseado em XML e HTTP) e SOAP (baseado em XML e diversos protocolos de comunicação). As evoluções desses formatos de representação da informação e dos protocolos de comunicação serviram como ponto de partida para a especificação de Serviços Web [GRA 05].

4.2.4.4 Serviços Web (Web Services)

Novas tecnologias e *frameworks* de desenvolvimento surgiram permitindo uma maior integração entre os diversos aplicativos e serviços disponíveis na Internet. Esse novo

modelo deve tratar tarefas complexas, como o gerenciamento de transações, por meio da disponibilidade de serviços distribuídos que utilizem interfaces de acesso simples e bem definidas. Esses serviços ou aplicativos distribuídos são conhecidos como Serviços Web [GRA 05].

Os Serviços Web são tecnologias de *middleware* baseada em XML que oferece um alto grau de interoperabilidade e flexibilidade para as aplicações, devido principalmente à ubiquidade dos protocolos [CRU 08]. São inúmeros os benefícios que os serviços Web apresentam: independência de plataforma de hardware e software, baixo acoplamento e reusabilidade de serviços. Essas características conferem aos Serviços Web o potencial de interoperabilidade que nenhuma outra tecnologia até hoje proposta conseguiu atingir. Alguns dos principais aspectos que definem um Serviço Web são: construídos sobre XML; orientado a mensagem; em sua essência são distribuídos; possuem baixo índice de acoplamento; são baseados em componentes; e podem ser localizados dinamicamente [GRA 05].

Os Serviços Web são identificados por um URI (*Unique Resource Identifier*) e são descritos e definidos usando XML. A padronização tem sido a chave para o sucesso e aceitação dessa tecnologia, pois são usados apenas protocolos e tecnologias abertas, padronizados e de ampla aceitação, como HTTP, XML e SOAP [GOM 04].

Tecnicamente, Serviços Web são serviços distribuídos que processam mensagens SOAP codificadas em XML, enviadas através de HTTP e que são descritas através de *Web Service Description Language* (WSDL).

Arquitetura dos Serviços Web

Os Serviços Web baseiam-se na arquitetura SOA, na qual se apresenta como uma arquitetura flexível orientada a serviços. Um SOA é um modelo de projeto com um conceito consolidado de encapsulamento da lógica de programação dentro de serviços que atuam entre si por meio de um protocolo de comunicação comum. Quando Serviços Web são usados para estabelecer esse suporte de comunicação, eles basicamente representam uma implementação baseada em Web de um SOA [ERL 04].

Os componentes da arquitetura SOA são coleções de serviços que se comunicam por meio de troca de mensagens. Três papéis são definidos na arquitetura SOA [CRU 08]:

- Fornecedor de Serviços: responsável por descrever as informações de ligação dos serviços usados para sua chamada, essas informações são representadas em XML escrito

na linguagem padrão WSDL (*Web Services Description Language*). O fornecedor também é responsável pela publicação de Serviços Web no registro de serviços.

- Solicitante de Serviços: responsável por encontrar uma descrição de um Serviço Web publicado em um ou mais registro de serviço e usa tal descrição para ligar-se ao respectivo provedor e invocar o serviço.
- Registro de Serviços: responsável por manter a lista de todos os serviços e suas descrições, como nome, fornecedor e categoria. Além disso, é responsável por anunciar descrições de Serviços Web publicado por fornecedores de serviços e por permitir que solicitantes de serviços inspecionem as descrições de serviços existentes no registro. O registro de serviços funciona como um intermediário entre fornecedores e solicitantes de serviços. O padrão adotado na arquitetura de Serviços Web para registro é o UDDI (*Universal Description, Discovery and Integration*).

A interação entre os três papéis envolve três operações básicas: a publicação da informação sobre um determinado serviço, a descoberta dos serviços disponíveis e a ligação entre esses serviços. A Figura 4.1 mostra como os papéis e interações são relacionadas.

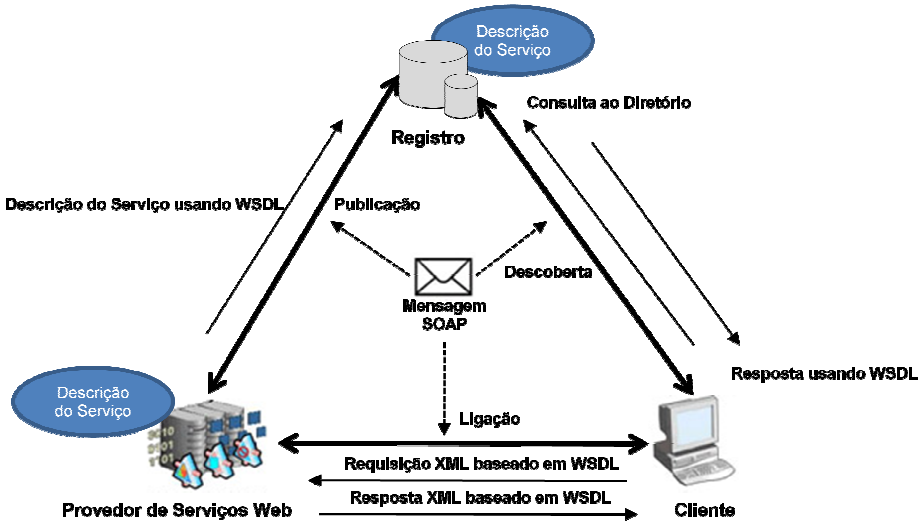

Figura 4.1 - Serviços Web: papéis e suas interações. (Adaptado de [CRU 08])

Um dos maiores benefícios da arquitetura SOA é o desacoplamento das requisições do cliente ao Serviço Web. Outras vantagens presente nessa arquitetura, dentre as quais se destacam o suporte para diferentes tipos de clientes, reusabilidade, escalabilidade e disponibilidade. Essas vantagens contribuem para a integração de processos de negócios além de capacitar e reutilizar serviços e recursos.

Classificação dos Componentes dos Serviços Web

A tecnologia de Serviços Web se enquadra na arquitetura SOA, e pode ser classificada de acordo com três conjuntos de especificações [CRU 08]:

1. Descrição de Serviços: utilizada para definir as operações, mensagens e os tipos de dados de um serviço e mantém as informações sobre como acessar os serviços;
2. Publicação e descoberta de serviços: contém os protocolos que possibilitam a localização das descrições dos serviços;
3. Protocolos de comunicação: utilizado para definir, estabelecer e manter a comunicação entre as aplicações. Contém também a descrição dos formatos das mensagens utilizadas no estabelecimento da comunicação entre as aplicações.

De acordo com essa classificação, os três conjuntos de especificação têm em comum o uso da linguagem XML, tecnologia padrão, aberta e extensível que permite a integração e troca de dados entre componentes distintos, garantindo a interoperabilidade necessária para a arquitetura.

Descrição dos Serviços

A WSDL provê um modelo e um formato para descrever Serviços Web, é uma linguagem baseada em XML que descreve de forma padronizada e independente de plataforma como e onde os Serviços Web podem ser conectados e utilizados através da rede [W3C 08b].

O documento WSDL representa um contrato entre o provedor de serviços e seus clientes. Foi desenvolvido através da união de grandes empresas como Microsoft, IBM e, em seguida, enviada a W3C (*World Wide Web Consortium*) objetivando a padronização. A especificação da linguagem contém um esquema XML que descreve a estrutura em que cada documento WSDL deve obedecer [GOM 04].

Segundo [MEL 06] um documento WSDL é independente de linguagem e de plataforma e tem por objetivo: (1) descrever quais são os serviços oferecidos; (2) mostrar como os clientes e provedores irão processar as requisições; e (3) indicar em qual formato o serviço deve enviar as informações para um cliente.

Em um documento WSDL, serviços são definidos como uma coleção de portas na rede. Um documento WSDL descreve um Serviço Web em dois estágios fundamentais:

abstrato e concreto. A parte abstrata descreve o que o Serviço Web faz em termos de mensagem que consome e produz. A parte concreta refere-se à implementação e define como e onde os serviços são oferecidos [MEL 06], conforme ilustrado na Tabela 5:

Tabela 5 - Elementos abstratos e concretos de um documento WSDL.

Abstrato	*types*: define os tipos de dados (independente de plataforma ou linguagem) utilizados para descrever as mensagens trocadas entre aplicações, normalmente representada por um documento XSD (*XML Schema Definition*).
	message: representa a mensagem que será trocada por meio das definições de dos tipos de dados.
	porttype: descreve um conjunto abstrato de operações mapeadas para um ou mais serviços, os quais são descritos como pontos finais de rede ou portas. Cada operação se refere a mensagens de entrada, saída ou erro.
	operation: descreve uma ação fornecida pelo serviço.
Concreto	*binding*: define uma especificação de protocolo e formato de dados para as mensagens definidas em um *porttype*.
	port: uma combinação entre o elemento *binding* e o endereço de rede URI, provendo assim um endereço único para acessar um serviço.
	service: é uma coleção de elementos *ports* relacionados. Cada elemento *ports* relaciona-se com um elemento *binding* particular, indicando qual interface e qual protocolo de comunicação está sendo utilizado na implementação.

Cada um desses elementos pode ser descrito em diferentes documentos XML, e a combinação de todos formam uma descrição completa de um Serviço Web. A independência dos elementos traz uma grande flexibilidade que proporciona a disponibilização dos serviços.

Publicação e Descoberta dos Serviços

A especificação do UDDI define uma forma padronizada para publicação e descoberta de serviços dentro da SOA, parte fundamental na pilha de protocolos dos Serviços Web [MEL 06].

O UDDI é um registro central de acesso e controle que descreve como os dados devem ser armazenados em um registro, além de definir os métodos possíveis para publicação e busca desses registros. Os dados incluem os contatos da organização, uma lista de serviços disponíveis e como usá-los por meio de algum tipo de programação. Todos os

acessos aos registros são realizados utilizando o padrão SOAP tanto para consulta como atualização [KRO 03].

De forma geral, o registro de serviços UDDI possui dois tipos de clientes. O primeiro envolve as aplicações que desejam publicar serviços e suas interfaces, o segundo tipo envolve os clientes que desejam obter e se ligar a Serviços Web.

O UDDI, em uma analogia, é a lista telefônica para Serviços Web. Conceitualmente, as informações disponíveis no UDDI consistem de três componentes: "páginas brancas", incluem os detalhes como nome e informações para contato; "páginas amarelas", provêem a categorização baseada nos tipos de serviço e negócios disponíveis; "páginas verdes", incluem dados técnicos sobre os serviços.

A especificação UDDI define quatro estruturas de dados, ou registros, descritas como documentos XML: *businessEntity*, *businessService*, *bindingTemplate* e *tModel*, conforme mostra a Figura 4.2.

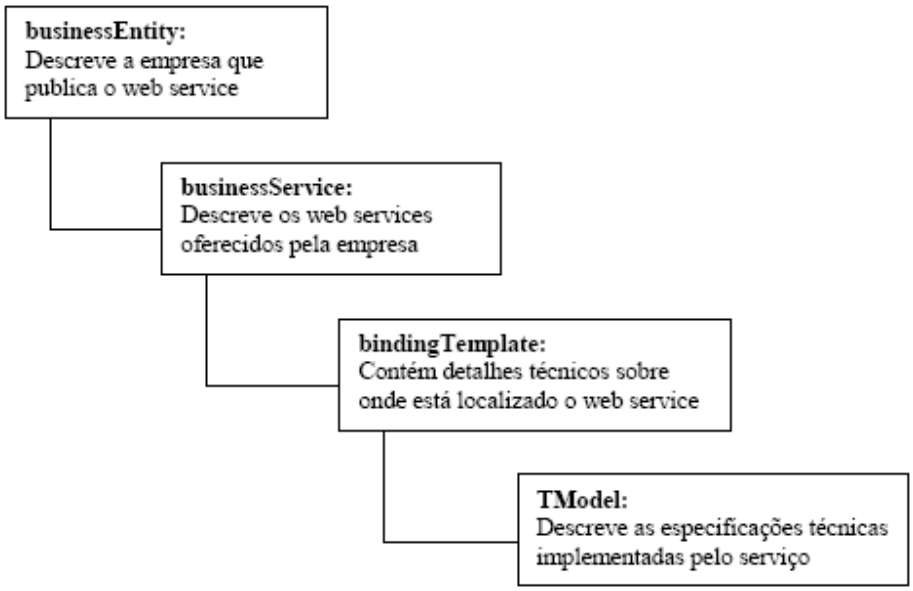

Figura 4.2 - Estrutura UDDI e seus relacionamentos.

O *businessEntity* é uma estrutura de alto nível (página brancas) que contém as informações (nome, categoria, identificadores, entre outros) de cada serviço, sobre a organização que publicou o Serviço Web. O *businessService* contém informações sobre cada um dos serviços oferecidos pela organização (páginas amarelas), tais como nome e descrição do serviço publicado. O *bindingTemplate* (páginas verde) contém informações técnicas, tais como realizar o acesso e quais os endereços dos pontos de entrada do Serviço Web. Por fim o TModel é o mecanismo usado para a troca de definições abstratas (metadados) sobre um

Serviço Web. Ele aponta opcionalmente para um documento WSDL que descreve a interface de serviço.

Protocolos de Comunicação

O protocolo SOAP (*Simple Object Access Protocolo*) foi criado em 1998 através de um grupo de desenvolvedores de grandes empresas como a Microsoft, *User Land Software e DevelopMentor*. Essas empresas tinham em mente a idéia de desenvolver um protocolo que definisse um mecanismo para a transmissão de documentos XML e que proporcionasse comandos capazes de disparar operações ou requisitar respostas em sistemas remotos. Assim surgiu SOAP e vem sendo adotado como um padrão para Serviços Web. [GOM 04].

Embora XML seja feito para a troca de dados, sozinho não é suficiente para o intercâmbio de informações através da Web, por isso a necessidade de um protocolo que possa enviar e receber mensagens contendo documentos XML. O propósito de SOAP é o de fazer chamada a procedimentos remotos sobre o protocolo HTTP, ou outro protocolo padronizado da Web, com a grande vantagem de não impor restrições de algum tipo de implementação para os pontos de acesso, como fazem RMI, CORBA e DCOM. Portanto, o papel do SOAP, na arquitetura dos Serviços Web, é estabelecer a comunicação entre o consumidor de serviços e o provedor de serviços através de troca de mensagens [GOM 04].

SOAP é basicamente um protocolo *stateless*, que possui o paradigma de troca de mensagem baseado em *one-way*. Ou seja, as mensagens transmitidas não são requisições que necessitem de respostas, são simplesmente mensagens que contêm dados a serem enviadas a um receptor. Entretanto, SOAP possibilita a construção de padrões de interações complexas, como por exemplo, requisição e resposta, ou requisição e múltipla-respostas [GRA 05].

Uma mensagem SOAP é um documento XML composto por três partes: envelope SOAP (obrigatório), cabeçalho SOAP (opcional) e o corpo SOAP (obrigatório). Conforme mostra a Figura 4.3.

O envelope SOAP é o elemento de nível mais elevado da mensagem e representa a mensagem propriamente dita. Contém o cabeçalho e o corpo SOAP. O cabeçalho SOAP deve estar logo após o envelope SOAP. Quando uma mensagem SOAP sai da sua origem e chega ao destino provavelmente passa por diversos nós enquanto viaja pela web, ao passar por nós intermediários, o cabeçalho pode ser usado para fazer um processamento com a

mensagem. Esses tipos de processamentos podem ser autorização, gerenciamento, entre outros. Já o corpo SOAP contém a carga útil, ou seja, as informações que devem ser recebidas pelo destinatário. Dentro do corpo SOAP pode estar contido o elemento *fault* que é usado para transportar informações sobre erros que possam vir a ocorrer no processamento das mensagens.

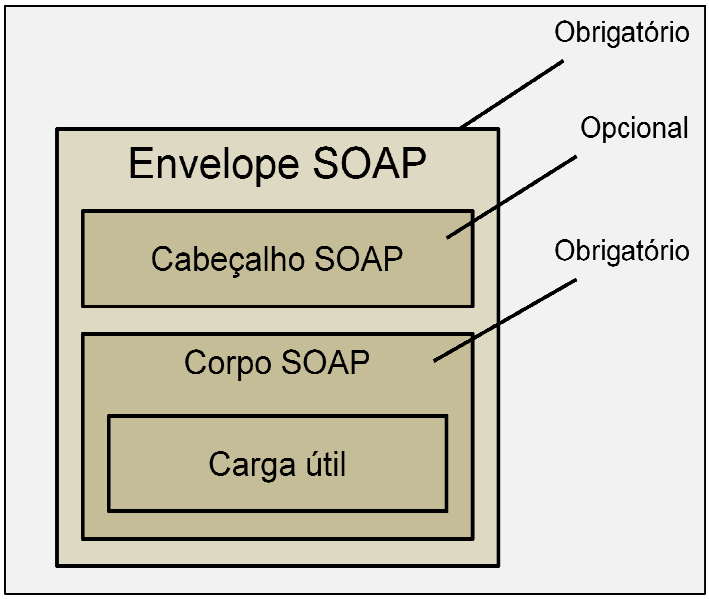

Figura 4.3 - Representação gráfica de uma mensagem SOAP [GOM 04].

A especificação do protocolo SOAP tem quatro partes:

- Envelope SOAP: define uma infra-estrutura global para expressar o que está contido na mensagem, ou seja, o que é a mensagem, quem deve tratá-la, se ela é opcional ou obrigatória e como sinalizar erros.
- Infra-estrutura de ligação: define uma infra-estrutura abstrata para trocar envelopes SOAP entre pares comunicantes, usando um protocolo para o transporte.
- Regras de codificação: define um mecanismo de serialização que pode ser usado para trocar instâncias de dados definidos pela aplicação, vetores e tipos compostos.
- SOAP RPC: define uma convenção que pode ser usada para representar chamadas remotas a procedimentos e suas respectivas respostas.

Apesar da combinação SOAP e HTTP serem amplamente utilizada, a especificação SOAP não se refere à dependência ao protocolo HTTP, com isso SOAP pode ser utilizado em conjunto com outros protocolos como, por exemplo, FTP ou SMTP.

4.2.5 Publish/Subscribe

O paradigma *publish/subscribe* fornece um fraco acoplamento de interação em grande escala [EUG 03]. Os *subscribers* são entidades que estão interessadas em receber eventos e podem definir seus interesses em cada evento, a notificação ocorre quando um evento de interesse for gerado por um *publisher*. Um evento é propagado de forma assíncrona para todos os *subscribers* que registraram um interesse. Este estilo de interação fornece um desacoplamento entre produtores e consumidores no tempo (publicadores e subscritores não precisam estar ativos na interação ao mesmo tempo), espaço (publicadores e subscritores não precisam conhecer um ao outro) e fluxo (publicadores e subscritores não precisam estar sincronizados para interagir).

O esquema *publish/subscribe* provê que, assinantes ou *subscribers*, tenham a habilidade de expressar seus interesses em um evento ou um padrão de eventos, e posteriormente serem notificados de algum evento gerado por um publicador. Em outras palavras, produtores publicam informações sob um barramento de software e consumidores subscrevem a informação que querem receber. As variantes mais comuns de subscrição e publicação são [EUG 03]:

1. Baseado em Tópico: essa variante é baseada na noção de tópicos ou assuntos. Participantes podem publicar eventos e subscrever em tópicos, nos quais são identificados por palavras-chaves. Na prática, sistemas *publish/subscribe* baseado em tópico apresentam uma abstração de programação que mapeia tópicos de forma individual para distintos canais de comunicação.

2. Baseado em Conteúdo: essa variante apresenta um esquema de subscrição baseado no conteúdo de eventos. Eventos não são classificados de acordo com algum critério pré definido, mas de acordo com as propriedades do seu próprio evento. Tais propriedades podem ser de atributos internos de estrutura de dados do evento, ou meta dados associados ao evento.

3. Baseado em Tipo: essa variante originou-se a partir de esquemas anteriores. Tópicos normalmente reagrupam eventos comuns que se apresentam não apenas em termos de conteúdo, mas também na estrutura. Essa observação levou a idéia de substituir o modelo baseado em tópico por um esquema que filtra o evento de acordo com seu tipo. As variantes anteriores são limitadas na concepção de orientação a objetos,

consequentemente, orientado a objetos e orientado a mensagens frequentemente são incompatíveis. A variante baseada em tipo fornece recursos da orientação a objeto como tipo de segurança e encapsulamento.

4.2.6 Espaço Compartilhado

O paradigma DSM (*Distributed Shared Memory*) fornece aos *hosts* a visão de um espaço comum compartilhado, no qual a sincronização e comunicação entre participantes acontecem por meio de operações sobre os dados compartilhados [EUG 03]. A noção de espaço de tuplas (*Tuple Space*) foi originalmente integrada em nível de linguagem em Linda, e fornece uma abstração simples e poderosa para acessar a memória compartilhada. Um espaço de tuplas é composto por uma coleção de tuplas ordenadas, igualmente acessível a todos os hosts de um sistema distribuído. Comunicação entre hosts acontece através da inserção ou remoção das tuplas no espaço de tuplas. Três principais operações podem ser executadas: *out*() exporta uma tupla para dentro do espaço de tuplas, *in*() importa e remove a tupla do espaço de tuplas, e *read*() lê, sem remover a tupla do espaço de tuplas.

O modelo de interação fornece desacoplamento no tempo e espaço, os produtores e consumidores de tuplas ficam anônimos com respeito a cada outro. O criador de uma tupla não necessita conhecer sobre o futuro uso de sua tupla ou seu destino. Uma interação baseado em *in* implementa a semântica 1 de N, ou seja, apenas um consumidor lê uma dada tupla. Enquanto que interações baseado em *read* podem ser usadas para implementar a entrega de mensagens 1 para N, ou seja, uma dada tupla pode ser lida por todos os consumidores. Diferente do paradigma *publish/subscribe*, o modelo DSM não fornece desacoplamento na sincronização, porque consumidores extraem tuplas do espaço em um estilo síncrono. Isso limita a escalabilidade do modelo devido à requerida sincronização entre os participantes. Para compensar a carência do desacoplamento de sincronização, alguns modernos sistemas de espaço de tuplas como JavaSpace, TSpaces, e WCL, estendem o modelo de espaço de tuplas de Linda com notificações assíncronas [EUG 03].

4.3 Discussão Sobre os Principais Paradigmas de Comunicação de Suporte a Sistemas Distribuídos

Alguns paradigmas apresentados neste capítulo constituem paradigmas de comunicação alternativos ao esquema *publish/subscribe*. A fim de comparação é dado ênfase a capacidade de desacoplamento entre os participantes. O desacoplamento que é fornecido entre publicadores e consumidores pode ser decomposto em três dimensões relacionadas à Tabela 6 [EUG 03].

- Desacoplamento no Espaço: os participantes de interação não necessitam conhecer um ao outro. Os publicadores publicam eventos através de serviço de evento e subscritores obtêm esses eventos indiretamente através de serviço de eventos.
- Desacoplamento no Tempo: os participantes de interação não necessitam estar ativamente participando na interação ao mesmo tempo. Em particular, os publicadores talvez publiquem alguns eventos enquanto o subscritor está desconectado, por outro lado, o subscritor talvez obtenha notificações sobre a ocorrência de algum evento enquanto o publicador original do evento está desconectado.
- Desacoplamento do fluxo: publicadores não são bloqueados enquanto produzem eventos, e subscritores podem obter notificações de forma assíncrona (por meio de um *callback*) da ocorrência de um evento enquanto desempenha alguma atividade concorrente. O produtor e o consumidor da mensagem não precisam estar sincronizados para interagir.

Desacoplar o produtor e o consumidor da informação aumenta a escalabilidade removendo todas as dependências entre as interações dos participantes [EUG 03].

Tabela 6 – Desacoplamento dos paradigmas de interação. (Adaptado de [EUG 03])

Abstração	Desacoplamento no espaço	Desacoplamento no tempo	Desacoplamento no fluxo
Passagem de Mensagem	Não	Não	Do lado do produtor
RPC/RMI	Não	Não	Do lado do produtor
Notificação	Não	Não	Sim
Espaço de Compartilhado	Sim	Sim	Do lado do produtor
Fila de Mensagens	Sim	Sim	Do lado do produtor
Publish/Subscribe	Sim	Sim	Sim

Na variante baseada em tópico do paradigma *publish/subscribe*, os eventos são definidos estaticamente em tempo de projeto. As entidades subscritoras precisam conhecer antecipadamente os tópicos definidos. Tópicos podem ser derivados em sub-tópicos, isto adiciona uma difícil manutenção e gerência da árvore de tópicos, além disso, as árvores de sub-tópicos podem se repetir à medida que a árvore cresce [INF 02].

Na variante baseada em tipo, os eventos são definidos como objetos de linguagem orientada a objetos com instancias de tipos definidos pela aplicação. Não apresenta uma hierarquia de tópicos nem alguma outra noção específica de tipos de eventos. Essa abordagem preserva o encapsulamento dos objetos por não forçar os eventos (tipos) a revelar os estados deles, pois os filtros de conteúdo podem fazer uso de métodos públicos. Os objetos de eventos são definidos como atributos e disponíveis através de métodos públicos. Porém, quando um subscritor está interessando em um determinado assunto, o objeto publicado é então copiado para o subscritor [EUG 07].

Já a variante baseado em conteúdo possui mais flexibilidade e expressividade em relação às abordagens anteriores [INF 02]. As aplicações não necessitam saber os tópicos definidos antecipadamente para realizar uma subscrição, ou seja, o evento não é classificado de acordo com algum critério externo pré-definido, mas de acordo com as propriedades do evento. As propriedades podem ser atributos internos de estruturas de dados que transporta o evento e consumidores subscrevem a eventos seletivos especificando um filtro. Os filtros definem restrições, sob a forma de pares de nome-valor, e operadores de comparações básicas (=, <, > etc.) que identifica valores válidos. Restrições podem ser combinados (and, or) para formar complexos padrões de subscrições [HAU 08]. Além disso, essa abordagem remove o custo administrativo de manter e definir grupos ou tópicos, e pode ser usado para implementar facilmente um *publish/subscribe* baseado em Tópico [AGU 99].

Neste trabalho a comunicação no interior da RSSF é baseada no paradigma *publish/subscribe* com a variante baseada em conteúdo, pois fornece maior flexibilidade e expressividade nos interesses dos subscritores, além de proporcionar menor tráfego de mensagens na rede, já que os publicadores transmitem somente as mensagens que expressam as restrições de interesses dos subscritores.

A comunicação em mais alto nível é realizada pelos Serviços Web, pois o acesso aos serviços do *middleware* deve ser realizado via Internet. As tecnologias dos Serviços Web fornecem *interfaces* reusáveis e flexíveis às aplicações que acessam os serviços, além de propiciar interoperabilidade entre diferentes plataformas de hardware e software.

4.4 Considerações Finais

Neste capítulo foram revisados alguns paradigmas de suporte a comunicação distribuída. Foi realizada uma descrição minuciosa com os principais componentes da arquitetura SOA e dos Serviços Web, tecnologia utilizada para prover os serviços via Web. O paradigma *publish/subscribe* prove total desacoplamento no tempo, no espaço e no fluxo, requisito importante para as aplicações de RSSFs.

No capitulo 5 será descrito: a especificação, projeto, arquitetura, implementação e avaliação do *middleware* de serviços multi-camadas para RSSFs.

5. Projeto de um Middleware de Serviços Multi-Camadas para RSSFs

5.1 Considerações Iniciais

Um dos projetos em desenvolvimento no Laboratório de Redes Sem Fio e Simulação Interativa Distribuída do Departamento de Computação da UFSCar envolve a pesquisa de métodos, técnicas e soluções para o monitoramento de infra-estruturas críticas, gerenciamento da emergência e treinamento de equipes na preparação e resposta a emergências em ambientes virtuais. O objetivo desse projeto é reduzir riscos de perda de vidas e patrimônio. Exemplos de emergências incluem: incêndio, explosão, vazamento de gás ou de substâncias tóxicas. As áreas de aplicação se concentram em monitoramento de condições críticas em plantas industriais e aeronaves. A especificação e implementação de um *middleware* de serviços multi-camadas para RSSFs é parte desse projeto maior.

O monitoramento de condições críticas consiste em supervisionar situações que podem levar a um incidente e acidente caso as situações não sejam controladas. Um ambiente é considerado em condição crítica quando algum contexto apresenta um estado anormal, como por exemplo: temperatura acima ou oxigênio abaixo de certo limiar, e presença de fumaça em um determinado local. Um sistema que gerencia emergências deve ser capaz de alertar a ocorrência de um incidente ou acidente, ou seja, prover suporte à segurança relacionado à perda de materiais, e que pode colocar em risco vidas humanas.

Com o avanço nas áreas de dispositivos heterogêneos de visualização e acesso à informação, sensores e as redes de comunicação sem fio vêm sendo utilizadas para impulsionar as aplicações de monitoramento preciso e em tempo real à prevenção, combate e avaliação de situações de emergências.

Sistema que gerencia e monitora condições de emergência em ambientes cientes de contexto consiste neste trabalho de um ambiente físico ou lógico povoado por nós sensores, que capturam informações do ambiente físico e remetem as informações a solicitantes (aplicações ou serviços). Além disso, neste trabalho, define-se que dado é uma representação de um evento, composto, por exemplo, por uma tupla (temperatura = 70).

Informação é uma união de vários dados e, após algum processamento, tem-se a interpretação de contextos.

Aspectos e requisitos importantes de um sistema que, monitora e gerencia emergência em ambientes físicos sob risco de incidentes e acidentes referem-se às seguintes questões: suporte e aquisição de informações do ambiente físico com apropriados paradigmas que abstraem a complexidade subjacente de rede, serviços que auxiliem no gerenciamento e interpretação de contexto, visualização em tempo real de incidentes em progresso, por meio de interfaces multimídia e de interfaces não convencionais (realidade virtual, realidade aumentada) em dispositivos que vão de celulares a cavernas digitais e gravação dos eventos que ocorreram no ambiente físico, bem como a gravação do ambiente virtual em 3D que representa o ambiente físico e os incidentes correspondentes para posterior reprodução.

Para atender os requisitos da classe de aplicações de emergência, uma arquitetura de *middleware* foi especificada e projetada. O *middleware* projetado procura atender também os princípios de projetos, características e alguns dos desafios identificados no capítulo 3.

Para isso, o *middleware* foi abstraído em dois níveis, alto nível e baixo nível, conforme mostra a Figura 5.1. A abstração de programação é a base de *middleware* para RSSFs. A abstração de programação fornece *interfaces* de alto nível que separam o desenvolvedor de aplicações de operações de rede subjacente [MIA 08]. Neste projeto os serviços do *middleware* são disponibilizados com tecnologias padrão e aberta da Web e a rede de sensores é abstraída com uma variante baseada em conteúdo do paradigma *publish/subscribe*.

O nível 1 refere-se ao *middleware* de serviços que provê os serviços em nós mais robustos, como o *sink*. Este nível fornece suporte a um mecanismo de interpretação de contexto, implementado por [CAM 09] e [ROC 09b], que realiza uma interpretação complexa que integra lógica *fuzzy* e ontologias para apoio à tomada de decisões para equipes de preparação e resposta a emergências. O *middleware* de nível 1 foi definido para prover serviços independentes de plataformas, sistema operacional ou linguagem de programação para as aplicações.

O nível 0 do *middleware* provê mecanismos de subscrições e publicações de dados no interior da RSSF. Neste nível, uma interpretação simples (fusão/agregação de dados) é realizada na camada de rede da pilha de protocolos de RSSFs. A variante baseada em conteúdo do *publish/subscribe* é utilizada para abstrair toda a complexidade de comunicação

distribuída na RSSF fornecendo um modelo de comunicação flexível e assíncrono para as aplicações.

Neste capítulo é descrita a estrutura de RSSF considerada neste trabalho em termos de seus componentes físicos. A arquitetura de um *middleware* de serviços multi-camadas é apresentada com seu funcionamento e detalhes de implementação. Uma avaliação das latências envolvidas na ativação de alguns dos serviços do *middleware* é realizada, bem como dos recursos envolvidos em termos de memória no nível da RSSF.

5.2 Estrutura Física e Organizacional da RSSFs

A arquitetura da RSSFs considerada neste trabalho é composta por uma estrutura física constituída por vários tipos de nós, dentre eles nós sensores, nós *sinks*, nós atuadores e, futuramente, *Rfids* (*Radio Frequency IDentification).*

Uma RSSF pode ser homogênea ou heterogênea. A rede pode ser formada por um único tipo de nó sensor, ou seja, todos os nós que compõem a rede têm as mesmas características. Esta rede é chamada homogênea. As redes formadas por nós de diferentes dimensões e características são ditas heterogêneas [LOU 03].

Neste trabalho, define-se que os nós sensores são homogêneos em relação às suas capacidades de processamento e armazenamento, assim como quanto a seus recursos de energia. Entretanto, os nós sensores podem possuir um ou diferentes ou múltiplos dispositivos de sensoriamento.

Sinks ou sorvedouros são nós que constituem a rede geralmente com maiores recursos computacionais e ligados à rede elétrica. Estes nós atuam como *interfaces* que provêem a aplicações clientes ou serviços de alto nível suporte ao submeter consultas para a rede e, posteriormente, obter os dados resultantes. Em função da *interface* de acesso, os *sinks* podem ser acessados localmente por aplicações ou remotamente através da Internet. O *middleware* de nível 1 é alocado no *sink.*

Atuadores são dispositivos com maior poder de processamento, comunicação e de energia, que podem tomar decisões e executar ações no ambiente físico, em resposta a situações interpretadas dos sensores.

Rfids serão integradas futuramente neste projeto e são etiquetas que identificam todos os objetos importantes em um ambiente físico. Leitoras de *Rfid* são posicionadas estrategicamente no ambiente para realizar as leituras de dados das etiquetas. As *rfids* podem ser integradas no interior de uma RSSFs para aumentar o nível de informação de um ambiente físico.

Na arquitetura proposta, as funcionalidades são designadas aos nós, de acordo com os recursos computacionais fornecidos. Por exemplo, o nível 1 consiste de funcionalidades que necessitam de nós mais robustos, portanto, situa-se, tipicamente no *sink*. Já o nível 0 provê a abstração para realizar as tarefas submetidas pela aplicação ou pelo nível 1, para a comunicação, aquisição e processamento de dados que devem ser realizadas no interior da RSSF.

5.3 Arquitetura do *Middleware*

O nível 1 do *middleware* foi baseado em Serviços Web. O nível 1 foi projetado para prover *interfaces* reusáveis e flexíveis às aplicações que acessam os serviços do *middleware*, além de fornecer interoperabilidade entre diferentes plataformas de hardware e software.

A Arquitetura Orientada a Serviço (SOA) foi usada para promover a comunicação entre aplicações e/ou serviços que acessam os serviços do nível 1 do *middleware*. SOA é um modelo de projeto para aplicações distribuídas que simplifica o desenvolvimento de aplicações distribuídas heterogêneas. Um aspecto importante é a clara separação entre a interface e a implementação de um serviço. Este aspecto é desejável quando se busca interoperabilidade, transparência de localização e fraco acoplamento entre consumidores e provedores de serviços [W3C 08a].

Várias iniciativas vêm sendo adotados como padrão para Serviços Web, o XML possui um formato padrão para troca de informações independente de plataforma e de linguagem de programação. O resultado de anos de pesquisa no desenvolvimento de protocolos baseados em XML foi à especificação XML-RPC, baseado em XML e HTTP [GOM 04]. Para mensagem foi especificado o protocolo SOAP baseado em XML e diversos protocolos de comunicação [OAS 08]. Para publicação e descoberta de serviços foi

especificado o UDDI [W3C 08a]. Interfaces são descritas em XML chamadas WSDL [W3C 08b]. WSDL inclui todas as informações necessárias para poder invocar métodos de serviços distribuídos.

O nível 0 do *middleware* foi baseado no paradigma *publish/subscribe* baseado em conteúdo. Ele foi projetado para prover maior flexibilidade e expressividade dos interesses das aplicações para RSSFs, auxiliando de certa forma no serviço básico de coleta, processamento e entrega de dados para as aplicações em RSSFs. Este esquema foi utilizado por prover total desacoplamento entre produtores e requisitantes de informações.

Um aspecto importante deste trabalho prevê que os serviços oferecidos pelo *middleware* possam ser acessados por vários tipos de dispositivos por meio da Internet. Uma visão geral da arquitetura do *middeware* é ilustrada na Figura 5.1.

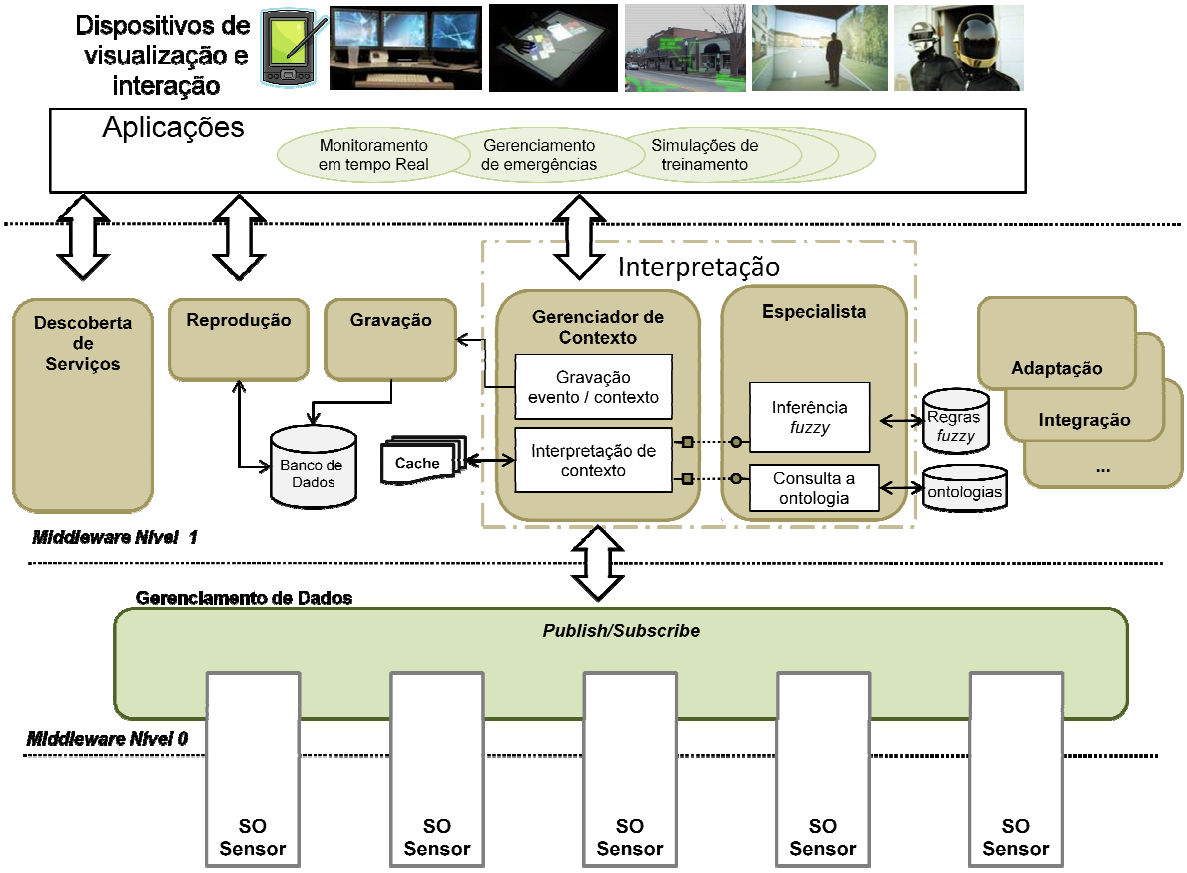

Figura 5.1 – Visão geral da arquitetura do *middleware*.

Os principais serviços do *middleware* são descritos abaixo:

Descoberta de Serviços: responsável por prover a localização dos serviços do *middleware*. Uma vez os serviços localizados, eles podem ser acessados por aplicações ou por outros serviços via Serviços Web. Com a utilização de SOAP e XML, ambos parte da

arquitetura de Serviços Web, é natural que o UDDI seja o protocolo escolhido para a descoberta de serviços.

Gravação: responsável por gravar diferentes níveis de informações interpretadas, de dados agregados a informações complexas de incidente e acidente. Essas informações são armazenadas em uma base de dados de contexto junto com o tempo que ocorreu.

Reprodução: responsável por exibir o ambiente virtual 3D. O 3DVE reproduz o ambiente físico tão próximo possível o real. O 3DVE é renderizado em sincronia com o contexto armazenado no Serviço de Gravação junto com um intervalo de tempo. Os contextos armazenados são mapeados em uma linguagem visual e renderizado por um browser.

Os serviços de Gravação e Reprodução foram implementados por [LOP 06] integrante do grupo de pesquisa.

Gerenciador de Contexto: responsável por interpretar e traduzir requisições de aplicações de alto nível ou de eventos que ocorrem no interior da rede. O interpretador possui a função de realizar a inferência no serviço Especialista. Os resultados são utilizados pelas aplicações de alto nível e também pelo serviço de gravação que grava as informações geradas.

Especialista: responsável por fornecer informações de alto nível sobre o que é percebido em uma RSSF. Este serviço utiliza uma base de conhecimento composta de ontologias e regras *fuzzy*, uma base de dados e de contextos, e futuramente deve incorporar redes bayesianas. O principal propósito deste serviço é, basicamente, aumentar o nível de abstração da informação para auxiliar o especialista de emergência no processo de tomada de decisões. O Serviço Especialista prove informações de alto nível do que ocorre no ambiente físico e será utilizado no gerenciamento da ocorrência de incêndio e acidentes em plantas industriais com risco de explosões. Este serviço foi desenvolvido por [CAM 09] e [ROC 09b], integrantes do grupo de pesquisa.

Os serviços, Gerenciador de Contexto e Especialista, fazem parte de uma unidade lógica definida como interpretação de contexto.

Adaptação: responsável por executar mudanças ou adaptações na rede e dispositivos de acordo com as respectivas capacidades e por políticas de adaptação. Este serviço será desenvolvido por outro integrante do grupo de pesquisa.

Base de Conhecimento: um conjunto lógico que contêm repositórios de ontologias, regras *fuzzy*, base de dados e de contextos. Os repositórios de ontologias e regras

fuzzy são utilizados por *inferenceFuzzy* e *queryOntology* para identificação de eventuais anormalidades no ambiente físico monitorado.

Os serviços podem ser utilizados por outros serviços e aplicações desenvolvidos por terceiros, necessitando apenas realizar uma descoberta de serviços e utilizar o documento WSDL do serviço para implementar a *interface* de acesso.

Alguns serviços são definidos como comuns em uma arquitetura de *middleware* para RSSF, a saber: Gerenciamento de Dados, Gerenciamento de Código, Descoberta de Recursos, Gerenciamento de Recursos, Integração. Alguns dos serviços podem ser disponibilizados ou não em um projeto, ou seja, são dependentes dos requisitos do sistema de *middleware* [MIA 08]. Por exemplo, o serviço Gerenciador de Código é responsável pela migração e atualização de código em uma rede. Esse serviço é visível em *middlewares* com a abordagem de Máquina Virtual, como Impala e Maté.

As funcionalidades dos serviços, Gerenciador de Dados e Integração, fazem parte deste projeto conforme descrito abaixo.

Gerenciamento de Dados: responsável principalmente pela aquisição, processamento, e entrega de dados. As primitivas do TinyOS e a abordagem *publish/subscribe* realizam estas funções neste trabalho. Essa abordagem tem duas principais vantagens em suportar aquisição de dados baseado em eventos. Primeiro, suporta comunicação assíncrona. Segundo, facilita a troca de mensagens entre nós sensores e o *sink*.

Integração: responsável por integrar/prover a RSSFs e os serviços a outras redes, tal como a Internet. Neste trabalho, adotou-se a tecnologia de Serviços Web para prover interoperabilidade e reusabilidade aos serviços do *middleware*.

Na arquitetura está previsto um *cache* que fica armazenando os últimos dados de contextos oriundos da RSSF. O *cache* é útil quando uma solicitação feita pela aplicação não retorna todos os dados necessários para realizar a inferência no serviço Especialista e os dados faltantes podem ser recuperados no *cache*. A próxima subseção apresenta brevemente o serviço especialista do *middleware* de nível 1, classificado como um serviço do domínio em [MIA 08].

5.3.1 Serviço Especialista do *Middleware*

O serviço Especialista é um interpretador de contexto que foi implementado por [CAM 09] e [ROC 09b]. O serviço Especialista é responsável por receber subscrições de

aplicações ou de outros serviços que requerem informações mais precisas sobre o que está ocorrendo no ambiente monitorado. Essas informações são obtidas por meio de uma base de conhecimento que é composta por ontologias e regras *fuzzy*.

A base de conhecimento é composta por um conjunto de cinco ontologias e regras *fuzzy* para interpretação de informações coletadas de sensores espalhados no ambiente físico. A interpretação de contexto é realizada a partir da consulta nas ontologias e da inferência realizada pelo sistema de inferência *fuzzy* nas regras *fuzzy*.

O grau de relevância do serviço Especialista pode ser observado como importante ferramenta de apoio ao assessorar equipes de emergência. O exemplo de uma ocorrência de incêndio, questões como:

1. Qual a fase do fogo?;
2. Qual é a melhor técnica para combate ao sinistro (água, espuma, CO_2)?;
3. Qual é o risco principal?;
4. Quais são os riscos vizinhos ao acidente?.

As respostas a estas questões são obtidas na base de conhecimentos, por técnicas que utilizam ontologias e regras *fuzzy* [CAM 09].

A Figura 5.2 mostra uma visão geral do serviço Especialista.

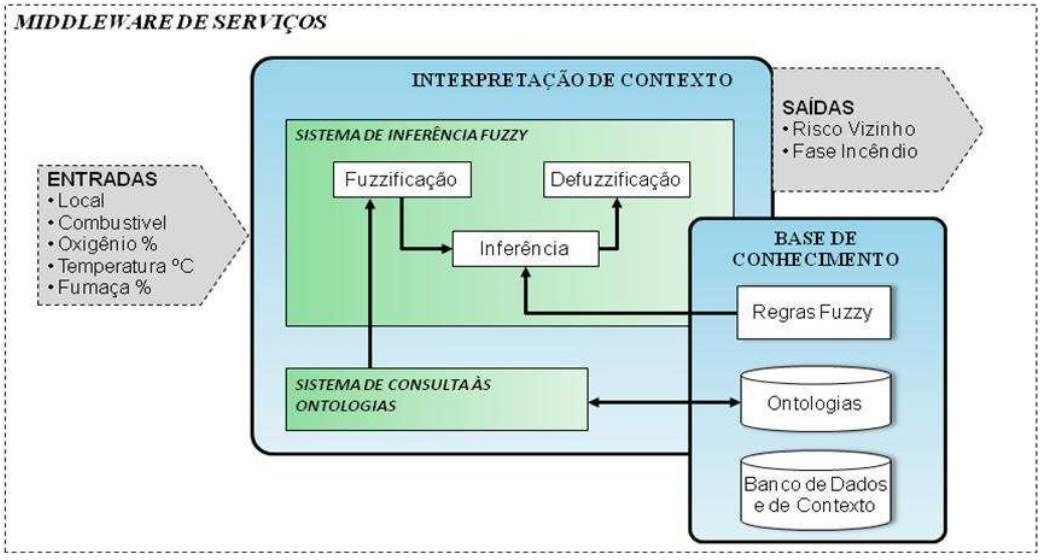

Figura 5.2 – Visão geral do interpretador de contexto do serviço especialista [CAM 09].

O serviço Especialista será utilizado no gerenciamento da ocorrência de incêndio e acidentes em plantas industriais com risco de explosões. Estudos foram baseados

sobre três tipos de explosões (*flash over*[1], *backdraft*[2] e *boil over*[3]) a partir da ocorrência de um incêndio.

A fim de disponibilizar o serviço Especialista como um Serviço Web, o administrador da rede deve registrá-lo em um serviço de registros UDDI. Desta forma, devem-se criar quatro tipos diferentes de "registros": o *businessEntity* registra a instituição proprietária da rede, o *businessServices* registra o serviço específico fornecido pela rede, o *bindingTemplate* inclui informações sobre como e onde acessar um Serviço Web específico (por exemplo, no registro é especificado que o serviço é disponível via SOAP em uma dada URL) e o *tModel*, ou modelo técnico, fornece ponteiros para especificações técnicas externas, que neste trabalho, consiste nos documentos WSDL do Serviço Especialista (Apêndice A e B).

5.3.2 Operação do *Nível 1* do *Middleware* Segundo o Padrão SOA

O *nível 1* do *middleware* conforma-se ao padrão SOA. Neste nível o *middleware* é visto como um fornecedor de serviços especializados para as aplicações.

O objetivo desta subseção é mostrar como os componentes arquiteturais do *middleware* aderem ao padrão SOA e como os papéis e operações definidos no padrão SOA são mapeados para a arquitetura proposta. As decisões de projeto foram motivadas pelo fato das tecnologias dos Serviços Web proporcionarem alto grau de flexibilidade, reusabilidade e interoperabilidade às aplicações e serviços, devidos as tecnologias basearem-se em protocolos e padrões ubíquos da Web.

A Figura 5.3 mostra o padrão de operação realizado pelo nível 1 do *middleware*. Uma aplicação desempenha o papel de solicitante que deseja receber informações do que está acontecendo no ambiente físico. Os nós *sinks* atuam como provedores de serviços para o meio externo. Eles funcionam como ponto de acesso a RSSF, fornecendo informações ricas e interpretadas de informações que chegam da rede. O endereço de rede desses nós (URL) é registrado em um serviço de registros UDDI, as aplicações

[1] **Flash over:** havendo uma oxigenação adequada com semelhante elevação da temperatura, o incêndio poderá progredir para uma ignição súbita generalizada.
[2] **Backdraft:** se a oxigenação é inadequada (incêndio controlado pela falta de ventilação) e a temperatura permanece em elevação, poderemos progredir para uma ignição explosiva, caso ocorra uma entrada brusca de oxigênio.
[3] **Boil over:** é a expulsão total ou parcial de petróleo ou outros líquidos combustíveis, em forma de espuma, de um tanque em chamas, quando o calor atinge a água acumulada no fundo do tanque.

realizam uma descoberta de serviços (*find*) no registro UDDI a fim de descobrir os Serviços Web oferecidos pelo *middleware*. A aplicação obtém o endereço do *sink* que provê o serviço que atenda a seus propósitos e então o contata diretamente para obter o documento WSDL. A aplicação, em posse do documento WSDL que fornece todo o contrato que o solicitante deve satisfazer para utilizar o serviço, pode então submeter à consulta.

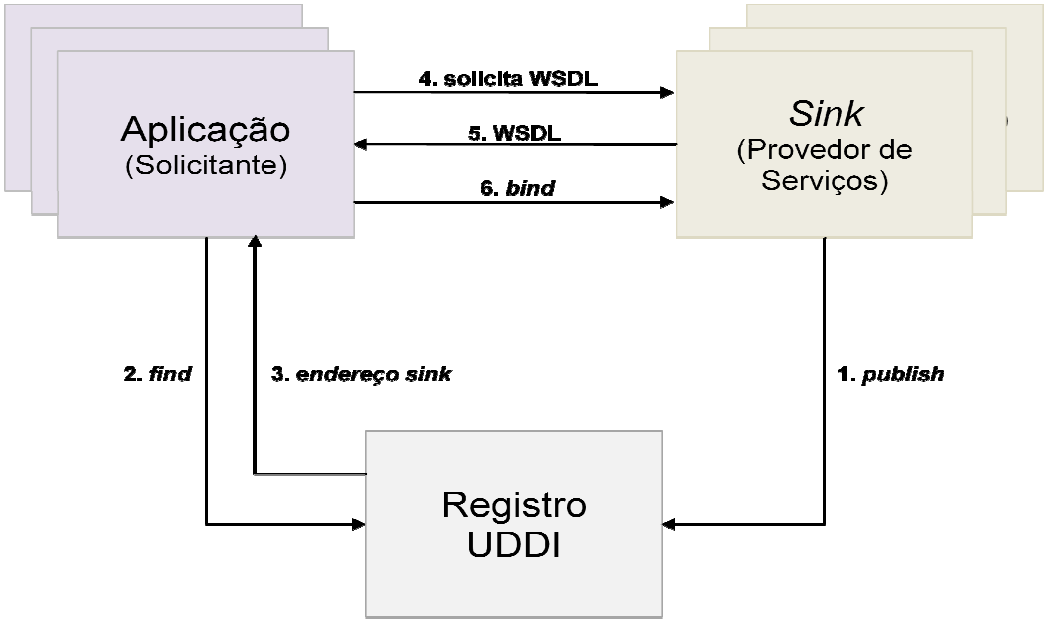

Figura 5.3 – Operação do *middleware* de nível 1 segundo o padrão SOA.

Na seção 4.5.4.1, foi descrito como os Serviços Web aderem ao padrão SOA. No *nível 1* do *middleware*, o protocolo SOAP é utilizado para comunicação, a descrição de serviços é realizada pelo WSDL e a descoberta de serviços acontece por meio do registro UDDI. A utilização das tecnologias que compõem os Serviços Web no projeto físico do *middleware* faz com que os diferentes serviços oferecidos sejam expostos como Serviços Web. As *interfaces* entre os serviços são descritas por meio de documento WSDL e as mensagens trocadas entre os componentes são implementadas como mensagens SOAP.

5.3.3 Operação do *Nível 0* do *Middleware* Segundo o Paradigma *Publish/Subscribe* Baseado em Conteúdo

Aplicações para RSSFs coletam e integram continuamente os dados gerados a partir de nós sensores. Como característica dessas redes há um grande número de dispositivos

trocando dados, enquanto algumas fontes de informação e *sinks* podem não estar presentes ao mesmo tempo. De acordo com esse cenário a comunicação do tipo *request/response* (síncrona) nem sempre é adequada para satisfazer aos requisitos das RSSFs. A passagem de mensagens (assíncrona) do tipo *publish/subscribe* é mais adequada para o modelo de disseminação de informação de aplicações dirigida a dados, requeridos pelas aplicações de rede de sensores [GUI 06].

O nível 0 do *middleware* mantém conformidade com o paradigma *publish/subscribe* baseado em conteúdo. As decisões de projeto foram motivadas pelo fato da variante baseado em conteúdo fornecer mais dinâmica, flexibilidade e expressividade nos interesses de eventos pelas aplicações. Neste nível, o *middleware* provê suporte à subscrição, publicação e notificação de eventos para as aplicações. A Figura 5.4 mostra a dinâmica do modelo *publish/subscribe* baseado em conteúdo.

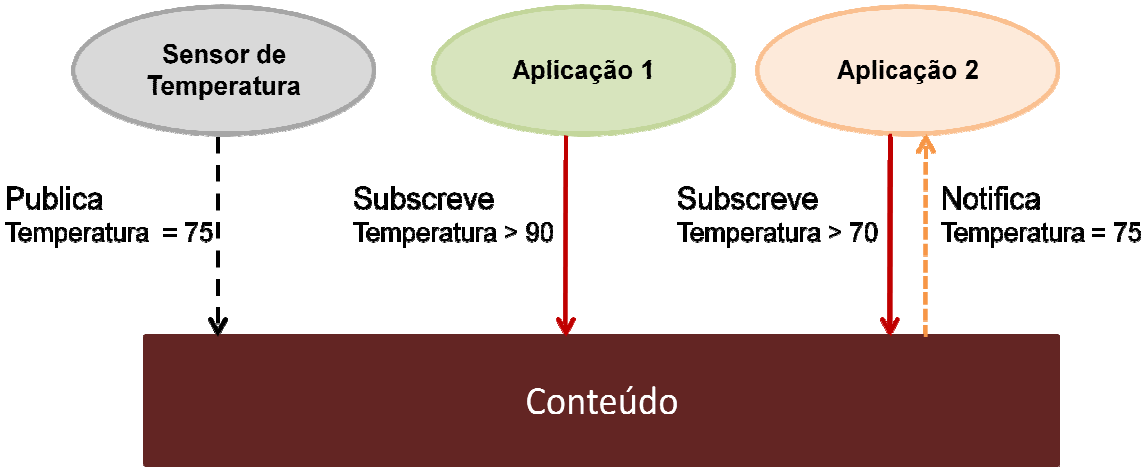

Figura 5.4 – *Publish/Subscribe* baseado em conteúdo.

O *publish/subscribe* baseado em conteúdo é mais geral em que ele pode ser usado para implementar facilmente a variante baseado em tópico enquanto o reverso não é verdadeiro [AGU 99], conforme mostra a Figura 5.5.

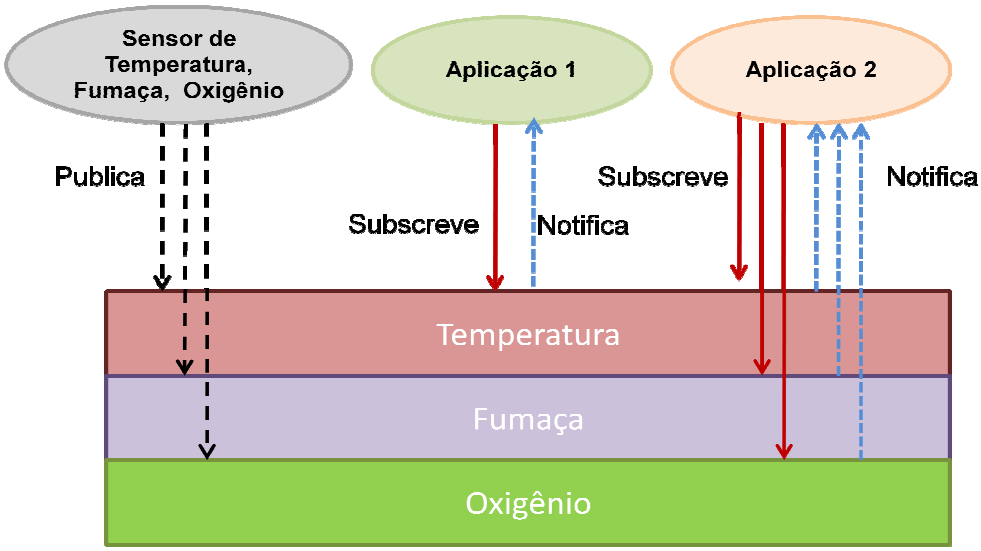

Figura 5.5 – Operação de tópicos no *Publish/Subscribe* baseado em conteúdo.

5.4 Funcionamento do *Middleware*

Para demonstrar o funcionamento do *middleware*, uma aplicação de gerenciamento de emergência foi implementada para acionar os Serviços Web do *middleware* de nível 1 para checar três tipos de interpretação de contexto do serviço Especialista: informações sobre o risco vizinho de um local, probabilidade de ocorrer *boil over*, e informações sobre se há um incêndio, e qual a fase e a probabilidade de ocorrer às explosões *flash over* e *backdraft*.

Os diagramas de sequência descritos nas próximas subseções contemplam aplicações complexas, como a de gerenciamento de emergência, e também aplicações simples. As aplicações simples são aquelas que estão interessadas em alguma informação sentida pela rede e não precisa passar por alguma interpretação de contexto do serviço Especialista. Por exemplo, a aplicação está interessada em valores de temperatura > 70°C.

O funcionamento básico do *middleware* proposto pode ser descrito em duas visões. Inicialmente, será explicado com um exemplo como funciona a aplicação de gerenciamento de emergências solicitando informações para a RSSF e posteriormente a RSSF notificando a aplicação de monitoramento em tempo real.

5.4.1 Solicitação da Aplicação de Gerenciamento de Emergências para RSSFs

O *middleware* proposto consiste em uma série de fases que são entrelaçadas com o funcionamento da rede de sensores. Neste exemplo, a aplicação de gerenciamento de emergências solicita informações para RSSFs e as seguintes fases são realizadas: descoberta de serviços, submissão de interesses da aplicação, aquisição e entrega de dados ao *sink*, e interpretação de contextos e entrega da informação para a aplicação de gerenciamento de emergências, conforme mostra a Figura 5.6.

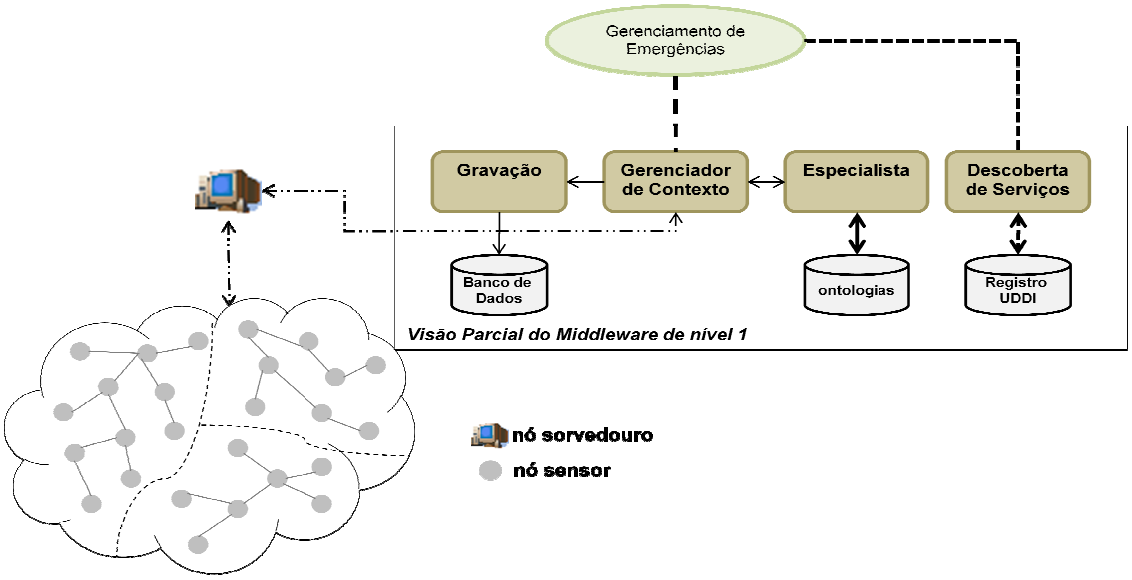

Figura 5.6 – Funcionamento do middleware, aplicação para a RSSF.

A fase de descoberta de serviço, de uma forma geral, acontece quando a aplicação está interessada em algum serviço que o sistema fornece. Com a utilização de SOAP e XML, ambos parte da arquitetura de Serviços Web, é natural que o UDDI seja o protocolo escolhido para a descoberta de serviços. Os nós *sinks* foram projetados para fornecer Serviços Web com todas as funcionalidades oferecidas pelo nível 1 do *middleware* e mantém um repositório com os documentos WSDL que descrevem o acesso aos serviços. Quando uma aplicação localiza um nó *sink* por meio do protocolo UDDI, ela adquire o documento WSDL a fim de entender o contrato disponibilizado por tal serviço. Tal contrato se refere ao formato das mensagens que o cliente deve obedecer para acessar o serviço.

Na fase de descoberta de serviços, a aplicação passa a conhecer o endereço do nó que disponibiliza os Serviços Web e o formato das mensagens para invocá-los. A aplicação, então, pode invocar as operações fornecidas pelo Serviço Web que são descritos

como elementos *operation* no documento WSDL por meio de envio de mensagens SOAP. Neste exemplo, a aplicação de gerenciamento de emergências realiza uma consulta para verificar se existe algum incêndio. Após conhecer as operações definidas a aplicação então submete a consulta ao *sink*. O Gerenciador de Contexto traduz a consulta em parâmetros e invoca o comando de envio do nó *sink* a fim de transmitir os interesses para a rede "*send(interest)*". Por exemplo, para consultar se existe algum incêndio, o Gerenciador de Contexto traduz o interesse nos parâmetros (temp, smoke, oxig, local) que são difundidos na rede por meio do nó *sink*. Em cada nó que recebe a mensagem via *broadcast* é sinalizado um evento *receive*. A Figura 5.7 mostra o diagrama de sequência referente à submissão de interesses da aplicação.

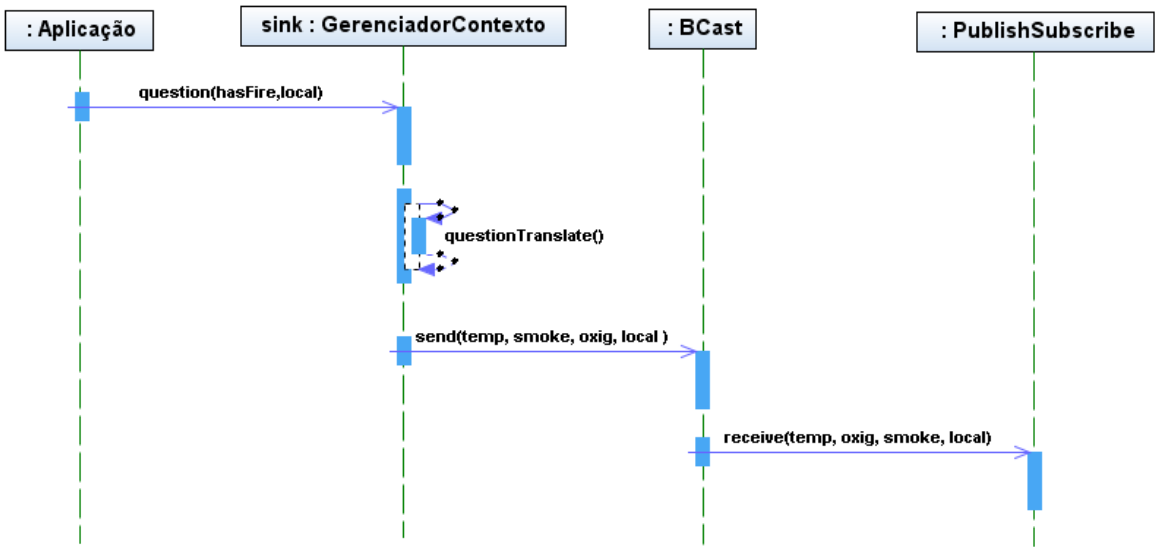

Figura 5.7 – Diagrama de sequência da submissão de interesses da aplicação.

Segue o fluxo normal e alternativo do diagrama de sequência da Figura 5.7.

Fluxo Normal:

1- Aplicação de gerenciamento de emergência solicita uma tarefa questionando "tem fogo ?" e o local alvo.
2- O Gerenciador de Contexto traduz a pergunta em parâmetros (temp, smoke, oxig, local) e invoca o comando de envio a fim de transmitir os interesses para a rede.
3- O *sink* envia para a rede os eventos a serem monitorados.
4- Em cada nó que recebe a mensagem via *broadcast* é sinalizado um evento *receive* com os parâmetros.
5- Encerra o caso de uso.

A próxima fase refere-se à aquisição e entrega de dados. Os nós sensores espalhados em um ambiente físico coletam dados sobre eventos, tais como, temperatura, presença de oxigênio, fumaça, etc. e enviam em direção ao nó *sink*. Neste trabalho a agregação de dados é realizada na camada de rede por meio do protocolo DAARP (*Data-Aggregation Aware Routing Protocol*) [VIL 09]. Uma das funcionalidades desse protocolo é otimizar as tarefas de roteamento para usar as capacidades de processamento disponível ao longo do caminho de roteamento, ou seja, os dados são agregados ao fluir em direção ao nó *sink*.

No protocolo DAARP enquanto o nó tem dados para transmitir, ele verifica se tem mais de um filho que transmite dados para ele. Se a condição for verdadeira então ele aguarda um período de tempo e agrega os dados recebidos e envia os dados agregados para o seu próximo *hop*. O nó transmissor aguarda um período de tempo para receber uma confirmação que o pacote foi entregue. Se a confirmação não for recebida, um novo nó destino é selecionado e a mensagem é transmitida [VIL 09]. O componente de agregação realizado por esse protocolo é mostrado no diagrama de sequência da Figura 5.8.

A aplicação do nó sensor publica os dados referentes a algum evento monitorado, enquanto o nó tem dados para transmitir, ele verifica se tem mais do que um filho que transmite dados para ele. Caso seja verdadeiro, os dados da rede são então agregados com os dados percebidos no nó local. Após agregar o dado (na camada de rede) a mensagem é transmitida de acordo com o algoritmo de roteamento representado pelo componente MultiHop. Ao passar por nós intermediários em direção ao *sink*, os dados recebem o mesmo tratamento, ou seja, se tem mais do que um filho que transmite dados ao nó e a aplicação local publicou dados, então é realizada a agregação. Finalmente, o Gerenciador de Contexto é notificado com o resultado.

A Figura 5.8 mostra o diagrama de sequência referente à aquisição e entrega de dados realizados na RSSF.

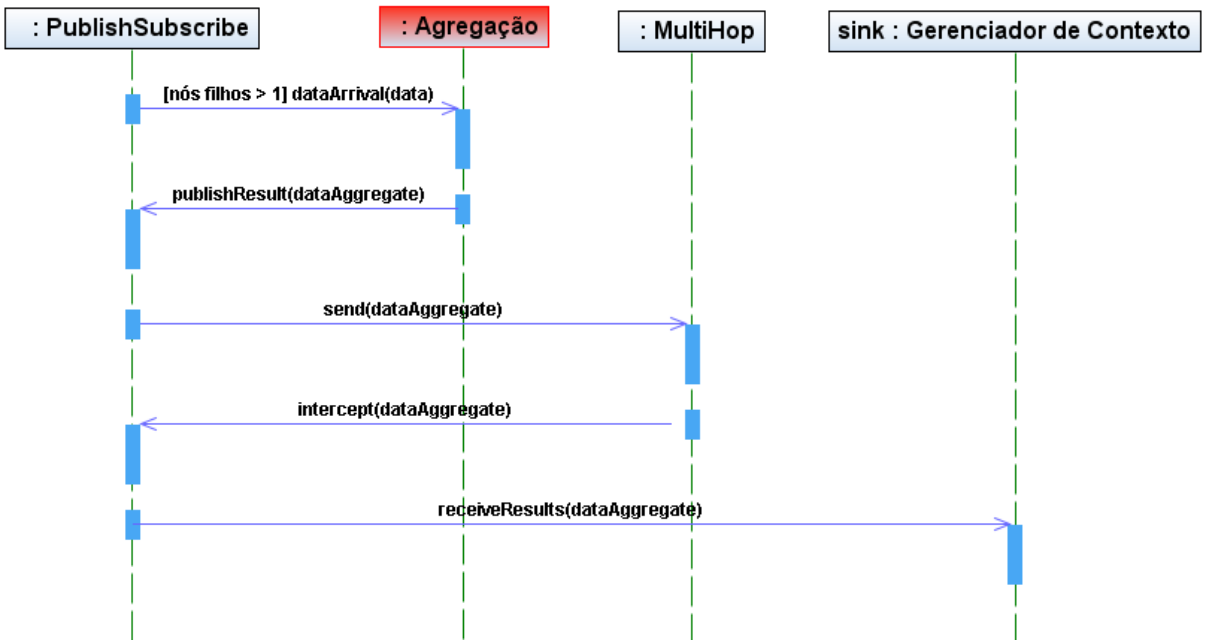

Figura 5.8 – Diagrama de sequência da aquisição e entrega de dados para o *Sink*.

Segue o fluxo normal e alternativo do diagrama de sequência da Figura 5.8.

Fluxo Normal:

1- O componente *PublishSubscribe* sinaliza o evento *dataArrival* informando os dados publicados localmente. A agregação é realizada na camada de rede com os dados vindos da rede, caso a condição [nós filhos > 1] seja verdadeira, de acordo com o protocolo DAARP.

2- A mensagem *publishResult*() contém os dados.

3- A mensagem é transmitida para o próximo nó através da primitiva *send*.

4- O próximo nó na hierarquia de roteamento intercepta a mensagem que contém *dataAggregate*.

5- Encerra o caso de uso.

Fluxo Alternativo:

1 - A condição [nós filhos > 1] é falsa.

1.1 - A agregação não é realizada, caso a condição [nós filhos > 1] seja falsa. O componente *PublishSubscribe* encapsula a mensagem e transmite-a para o próximo *hop* de acordo com o algoritmo de roteamento.

4 - O próximo nó é o *sink* na hierarquia de roteamento.

4.1 - O componente *PublishSubscribe* notifica o solicitante, neste exemplo, o serviço Gerenciador de Contexto.

A última fase refere-se à interpretação de contextos e entrega da informação solicitada. Mensagens de rede chegam ao *sink* com os dados solicitados. O Gerenciador de Contexto pode, então, realizar a inferência em *fuzzy* e a consulta a ontologia para solicitar os pontos de fulgor e de ignição do material. O resultado final é enviado para a aplicação de gerenciamento de emergência informando se está ocorrendo algum acidente.

A Figura 5.9 mostra o diagrama de sequência referente à interpretação de contextos e entrega da informação para a aplicação.

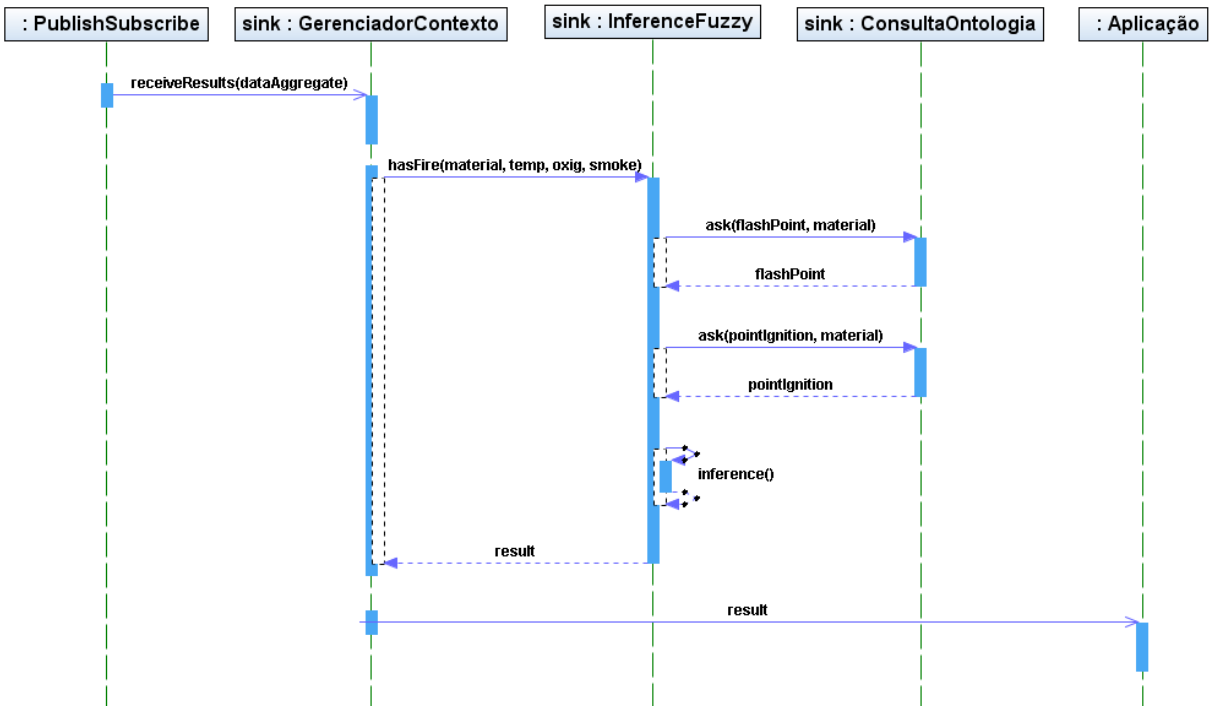

Figura 5.9 – Diagrama de sequência da interpretação de contextos e entrega da informação.

Segue o fluxo normal e alternativo do diagrama de sequência da Figura 5.9.

Fluxo Normal:

1- O componente *PublishSubscribe* notifica o serviço Gerenciador de Contexto, por meio da primitiva *receiveResults*().

2- O Gerenciador de Contexto solicita o serviço de *InferenceFuzzy* passando os parâmetros (material, temperatura, oxigênio e fumaça).

3- A *InfereceFuzzy* realiza uma consulta a ontologia solicitando os pontos de fugor e de ignição do material.

4- É realizada a inferência em *fuzzy*.

5- O resultado do processamento é enviado ao Gerenciador de Contexto.
6- O Gerenciado de Contexto envia a mensagem para a aplicação.
7- Encerra o caso de uso.

5.4.2 Notificação da RSSF para a Aplicação de Monitoramento em Tempo Real

A rede continuamente monitora um ambiente físico e notifica eventos que acontecem no interior da rede ao *sink* assim que alguma anormalidade é sentida pelos sensores. Como exemplo, a aplicação de monitoramento em tempo real recebe eventos percebidos e notificados pela RSSFs, os dados brutos são gravados pelo serviço de Gravação no Banco de Dados e o componente *PublishSubscribe* envia os dados para a aplicação de monitoramento em tempo real, conforme mostra um exemplo na Figura 5.10.

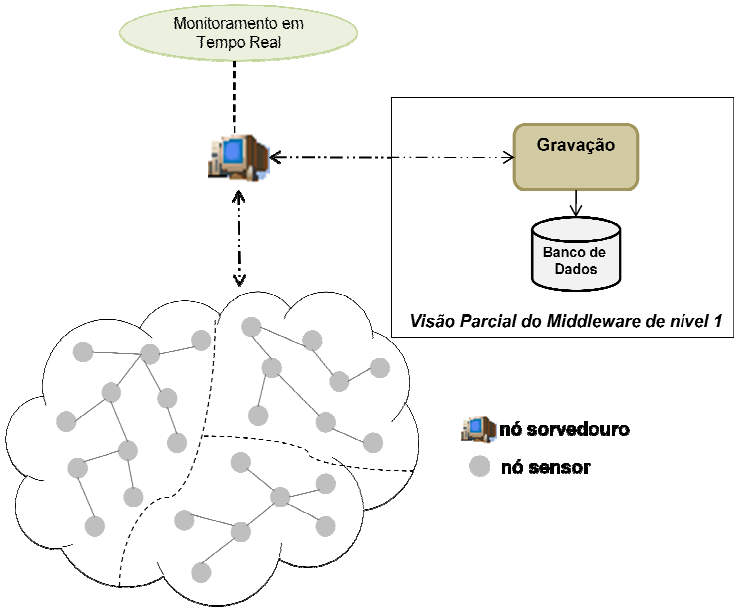

Figura 5.10 – Funcionamento do middleware, notificação da RSSF.

Similar ao diagrama de sequências da Figura 5.8, o nó sensor percebe algum evento anormal e a aplicação publica os dados referentes a algum evento monitorado. A agregação de dados é realizada na camada de rede ao fluir em direção ao nó *sink*. Ao chegar ao *sink* os dados são enviadas para a aplicação de monitoramento em tempo real. O usuário pode então solicitar mais informações do que está acontecendo no ambiente físico. Estas solicitações podem ser informações simples, por exemplo, verificar como está a pressão, se a

temperatura está acima de um limiar etc. As solicitações podem ser também mais refinadas, com as funcionalidades disponibilizadas pelo interpretador de contexto no serviço Especialista. A solicitação mais refinada é ilustrada no diagrama de sequência da Figura 5.7 quando a aplicação de gerenciamento de emergência pergunta se o ambiente físico tem fogo.

A Figura 5.11 mostra o diagrama de sequência referente à entrega de eventos da RSSFs para o *sink*.

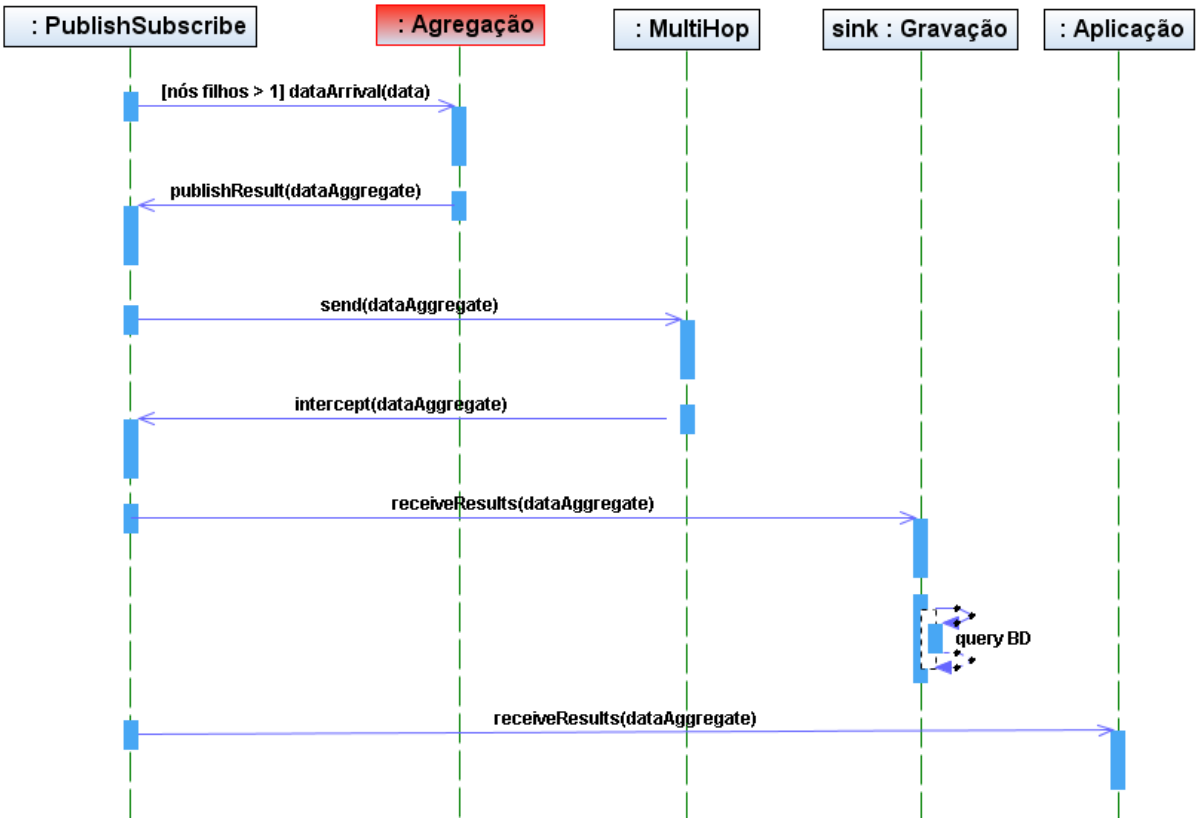

Figura 5.11 – Diagrama de sequência entrega de eventos da RSSFs para o *Sink*.

Segue o fluxo normal e alternativo do diagrama de sequência da Figura 5.11.

Fluxo Normal:

1. O componente *PublishSubscribe* sinaliza o evento *dataArrival* informando os dados publicados localmente. A agregação é realizada na camada de rede com os dados vindos da rede, caso a condição [nós filhos > 1] seja verdadeira, de acordo com o protocolo DAARP.
2. A mensagem *publishResult*() contém os dados.
3. A mensagem é transmitida para o próximo nó através da primitiva *send*.
4. O próximo nó na hierarquia de roteamento intercepta a mensagem que contém *dataAggregate*.

5. Encerra o caso de uso

Fluxo Alternativo:

1 - A condição [nós filhos > 1] é falsa.

1.1 - A agregação não é realizada, caso a condição [nós filhos > 1] seja falsa. O componente *PublishSubscribe* encapsula a mensagem e transmite-a para o próximo *hop* de acordo com o algoritmo de roteamento.

4 - O próximo nó é o *sink* na hierarquia de roteamento.

4.1 - O serviço de Gravação grava os dados no Banco de Dados.

4.2 - Os dados são enviados para aplicação de monitoramento em tempo real.

5.4.3 Implementação do *Publish/Subscribe* no *Middleware* de Nível 0

Esta subseção tem por objetivo analisar, o custo referente à ocupação de memória nos limitados nós sensores, ao adicionar na pilha de protocolos o DAARP, o sistema operacional TinyOS e o paradigma *publish/subscribe* baseado em conteúdo.

O *middleware* de nível 0 foi projetado com o paradigma *publish/subscribe* baseado em conteúdo e utiliza como interface as primitivas do sistema operacional TinyOS. Uma aplicação pode solicitar em um determinado momento informação de temperatura > 80°C em uma determinada área, conforme mostra a Figura 5.12. A principal vantagem neste nível do *middleware* é que diferentes subscrições podem ser feitas pelas aplicações nos nós sensores sem precisar esperar as respostas das primeiras solicitações, devido ao assincronismo proporcionado pelo paradigma *publish/subscribe*.

Na pilha de protocolos foram adicionados o TinyOS e o *publish/subscribe*. A camada de rede foi projetada com o protocolo DAARP que realiza a agregação de dados.

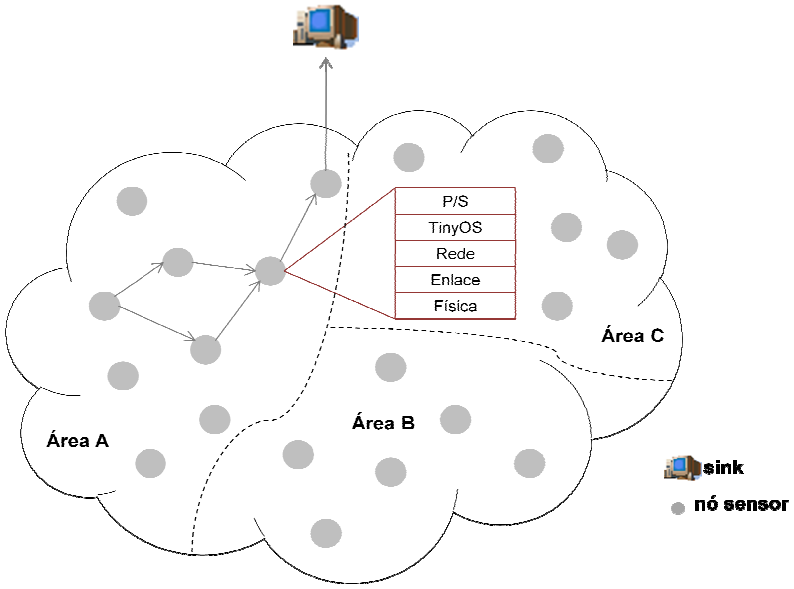

Figura 5.12 – RSSFs, pilha de protocolos, TinyOS e Publish/Subscribe.

Foi analisado e estimado o custo em adicionar na pilha de protocolos básica para a RSSF, o protocolo DAARP, o TinyOS e o *publish/subscribe*.

A pilha de protocolos básica para a RSSF já com o TinyOS é mostrada na Tabela 7 [HIL 00].

Tabela 7 – Pilha de protocolos básica com TinyOS (Adaptado de [HIL 00]).

Componentes	Código (bytes)	Dados (bytes)
Processor_init	172	30
TinyOS scheduler	178	16
C runtime	82	0
AM_dispatch	40	0
AM_temperature	78	32
AM_light	146	8
AM	356	40
Multihop router	88	0
Packet	334	40
UART_packet	314	40
UART	196	1
Radio_byte	810	8
Temperature	64	1
Photo	84	1
I2c_bus	198	8
RFM	310	1
Total	**3450**	**226**

A fim de analisar e estimar o custo, foram adicionados na pilha de protocolos básica para a RSSF o protocolo DAARP [VIL 09] e uma aplicação *publish/subscribe* baseada

em conteúdo [HAU 08], conforme mostra a Tabela 8, que contém as mesmas características de algumas das aplicações definidas neste trabalho.

Tabela 8 – Pilha de protocolos básica com DAARP, TinyOS e *Publish/Subscribe* (Adaptado de [HIL 00]).

Componentes	**Código (bytes)**	**Dados (bytes)**	
Subscriber	554	129	Publish/Subscribe
Publisher	738	70	Publish/Subscribe
Processor_init	172	30	TinyOS
TinyOS scheduler	178	16	TinyOS
C runtime	82	0	TinyOS
AM_dispatch	40	0	Active Message
AM_temperature	78	32	Active Message
AM_light	146	8	Active Message
AM	356	40	Active Message
DAARP	462	38	Rede
Packet	334	40	Enlace
UART_packet	314	40	Enlace
UART	196	1	Enlace
Radio_byte	810	8	Enlace
Temperature	64	1	Física
Photo	84	1	Física
I2c_bus	198	8	Física
RFM	310	1	Física
Total	**5116**	**463**	

Na plataforma de hardware dos motes Mica 2, a ROM é uma memória flash de 128 KB utilizada para armazenar os programas. Já a RAM é de aproximadamente de 4 KB e é utilizada para armazenar os dados da aplicação [LEW 10]. A Figura 5.13 mostra a ocupação de memória ROM em bytes na plataforma Mica 2.

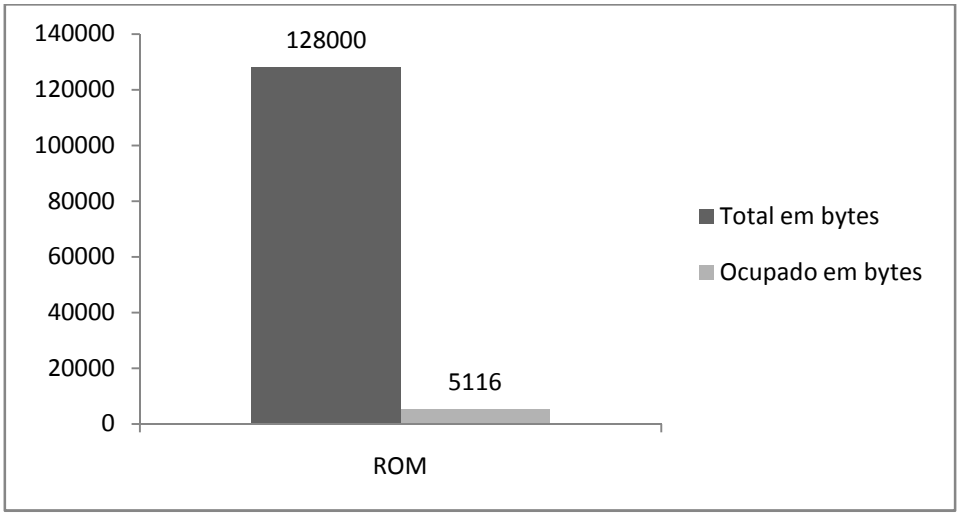

Figura 5.13 – Gráfico de ocupação de memória ROM na plataforma Mica 2.

A Figura 5.14 mostra a ocupação de memória RAM em bytes na plataforma Mica 2.

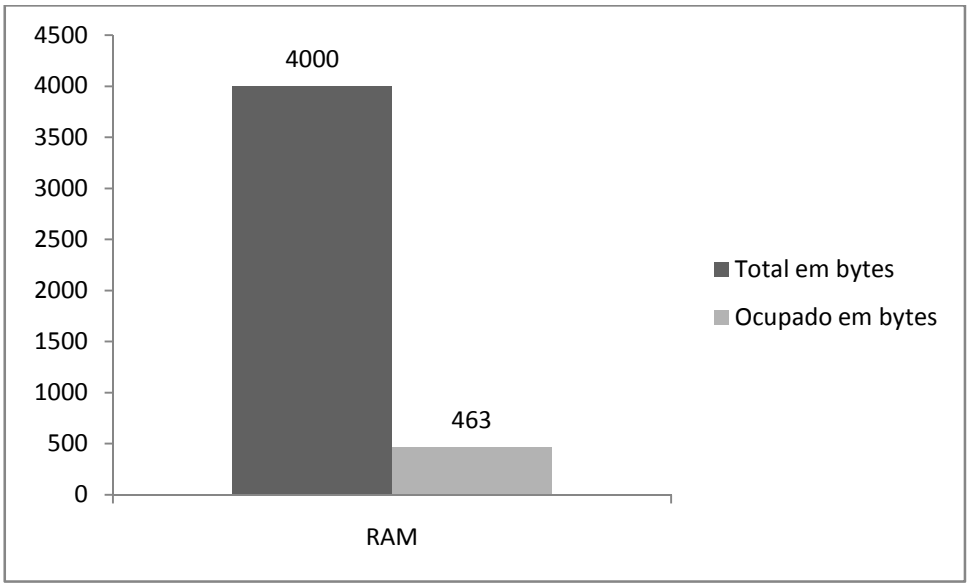

Figura 5.14 – Gráfico de ocupação de memória RAM na plataforma Mica 2.

O custo em adicionar na pilha de protocolos básica o protocolo DAARP e a aplicação *publish/subscribe* baseado em conteúdo foi de aproximadamente 1666 bytes de tamanho de código e de 237 bytes de dados. Deve-se considerar que foi realizada uma estimativa de custo com trabalhos reportados na literatura. Portanto, é viável sob o ponto de vista de ocupação de memória a utilização do protocolo DAARP, o TinyOS e o paradigma *publish/subscribe* baseado em conteúdo na plataforma Mica2.

5.5 Desenvolvimento do Protótipo do *Middleware* de Nível 1

Como prova de conceito para a proposta de sistema de *middleware* apresentada nesta dissertação, foi construído um protótipo com algumas das principais funcionalidades do *middleware* de nível 1. O objetivo da construção do protótipo é validar as interações entre a aplicação, os serviços do *middleware* de nível 1 e a RSSF. Para avaliar o *middleware* de nível 1, foi utilizado no interior da RSSF um trabalho de mestrado de outro integrante do nosso grupo de pesquisa [SPA 09], que utiliza agentes móveis ao invés do esquema *publish/subscribe* na RSSF. Os agentes móveis têm a tarefa de coletar e agregar dados na rede de sensores e comunicar o resultado ao nó *sink* [SPA 09]. Deve-se notar que foi utilizada esta

abordagem apenas para obter a latência da rede ao solicitar uma tarefa, já que foi realizada esta medição com os agentes móveis. Outra observação importante sobre esta avaliação refere-se ao fato que a agregação de dados é feita por agentes móveis e não pelo protocolo DAARP.

A fim de utilizar a abordagem de agentes móveis, um serviço simples de Gerenciamento de Agentes foi projetado e implementado no *sink* para coordenar e gerenciar o ciclo de vida de agentes móveis. Um arquivo texto foi gerado no *sink* com os resultados de processamento dos agentes móveis. Esse arquivo é usado pelo Gerenciador de Agentes para enviar as informações solicitadas pelo Gerenciador de Contexto conforme mostra a Figura 5.15.

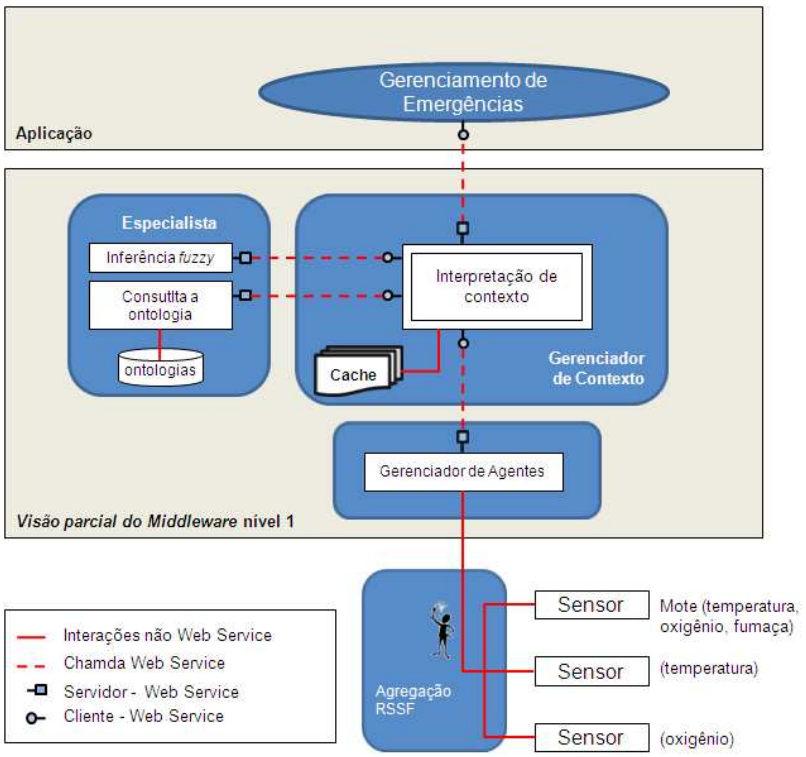

Figura 5.15 – Visão parcial do *middleware* com a integração do serviço Gerenciador de Agentes.

A construção do protótipo consiste do paradigma de orientação a objetos. O protótipo foi implementado usando a linguagem de programação Java. O nó *sink* onde foi executado o protótipo consistiu em um Dual Core 2.0, com 2 GBytes de memória RAM e HD de 80 GBytes. As classes representando as funcionalidades necessárias para o *sink*, bem como os Serviços Web, foram implementados utilizando o ambiente Apache Axis para desenvolvimento de Serviços Web [AXI 09] e a plataforma J2SE (Java Standard Edition) versão 1.4.2_13. O Axis fornece uma implementação da máquina SOAP e todas as classes necessárias para a análise e composição de mensagens SOAP, assim como a infra-estrutura

necessária para aplicações clientes invocarem operações dos Serviços Web. O Axis inclui suporte completo para a linguagem WSDL das seguintes formas:

- Quando um serviço é instalado no Axis, os usuários interessados acessam a URL do serviço com um navegador Web e adicionam " ?WSDL " ao final da URL, como por exemplo http://localhost:8080/axis2/services/inferenceFuzzy?wsdl. Desta forma é obtido um documento WSDL gerado automaticamente que descreve o respectivo serviço.
- Ferramentas conhecidas como WSDL2Java e Java2WSDL são disponibilizadas por meio do *plugin* Axis2 Codegen Wizard 1.3.0, as quais constroem classes Java a partir da descrição de serviços em documentos WSDL e constroem documentos WSDL a partir de classes Java.

Os Serviços Web representando os serviços do *middleware* de nível 1 foram instalados no servidor de aplicações TomCat [TOM 09] junto com os documentos que descrevem os serviços disponíveis escritos na linguagem WSDL (Apêndices A, B, C e D).

Para gerar as interfaces de Serviços Web, o Axis recebe como entrada um documento WSDL representando um serviço e gera chamadas de métodos em Java que correspondem à invocação das operações disponibilizadas pelo serviço. O Axis pode ser usado para gerar as classes necessárias tanto para o lado cliente quanto para o servidor.

Uma classe *Stub* contém o código que converte as chamadas de métodos em mensagens SOAP. Ela funciona como um *proxy* para o serviço remoto, permitindo que o mesmo seja chamado exatamente como se fosse um objeto local. A classe *Stub* esconde todos esses detalhes do usuário.

A Figura 5.16 mostra um fragmento do código da classe *Stub* do serviço Gerenciador de Contexto com a definição do método gerado.

Figura 5.16 – Fragmento de código da classe *Stub* do Gerenciador de Contexto.

```
public  mContext.ManagerContextStub.GetContextResponse getContext(
          mContext.ManagerContextStub.GetContext getContext0)
          throws java.rmi.RemoteException  {
          ....

        // cria um contexto de mensagem
        _messageContext = new org.apache.axis2.context.MessageContext();



        // cria um envelope SOAP
        org.apache.axiom.soap.SOAPEnvelope env = null;
```

```
        env = toEnvelope(getFactory(_operationClient.getOptions().getSoapVersionURI()),
                                    getContext0,
                                    optimizeContent(new javax.xml.namespace.QName("http://mContext",
                                    "getContext")));

    //adiciona o cabeçalho SOAP
     _serviceClient.addHeadersToEnvelope(env);

    // configura o contexto da mensagem com o envelope SOAP
    _messageContext.setEnvelope(env);

    // adiciona o contexto da mensagem para operação do cliente
    _operationClient.addMessageContext(_messageContext);

    //executa a operação do cliente
    _operationClient.execute(true);
    ....
```

Quando uma aplicação cliente quiser invocar as funcionalidades de um Serviço Web, ela deve instanciar uma classe localizadora de serviços e chamar um método *get* dessa classe. A classe localizadora é derivada da cláusula *service* no documento WSDL. A Figura 5.17 mostra esta operação.

Figura 5.17 – Aplicação cliente solicitando o serviço Gerenciador de Contexto.

```
try{
stub = new ManagerContextStub ("http://10.222.1.11:8080/axis2/services/managerContext");

 ManagerContextStub.GetContext gt = new ManagerContextStub.GetContext();
gt.setParam("Backdraft");
            }
          ....
```

A Tabela 9 apresenta a classe *Stub* e a classe adicional gerada no lado cliente.

Tabela 9 – Classes geradas para o lado cliente pelo Axis.

Documento WSDL	Classes Java geradas
managerContext.wsdl	ManagerContextStub.java
	ManagerContextCallbackHandler.java

De forma similar como o Axis gera uma classe *Stub* que representa o lado cliente de um Serviço Web, um *Skeleton* é uma classe Java para o lado servidor.

A classe *Skeleton* é a classe que se situa entre o engenho Axis e a implementação real do Serviço Web. Seu nome é o nome do serviço acrescido do sufixo "*Skeleton*". A classe *skeleton* não contém código algum; seu corpo é vazio. O desenvolvedor do serviço tem a função de preencher a lógica de negócio de acordo com a especificação do

serviço. A Figura 5.18 apresenta um fragmento de código da classe gerada e preenchida com a lógica de negócio do serviço Gerenciador de Contexto.

Figura 5.18 – Classe *Skeleton* do Gerenciador de Contexto.

```
public class ManagerContextSkeleton{

    /**
     * Auto generated method signature
     *
                        * @param getContext
     */

          public mContext.GetContextResponse getContext
          (
          mContext.GetContext getContext
          )
       {
          //TODO : fill this with the necessary business logic
          //hrow new  java.lang.UnsupportedOperationException("Please implement " +
          this.getClass().getName() + "#getContext");

         GetContextResponse response = new GetContextResponse();
          String resposta = "";

          resposta = managerContext.getContext(getContext.getParam());
                              response.set_return(resposta);
          return response;
    }

    ....
```

A Tabela 10 apresenta a classe *Skeleton* e as classes adicionais geradas no lado servidor.

Tabela 10 – Classes geradas para o lado servidor pelo Axis.

Documento WSDL	Classes Java geradas
managerContext.wsdl	ManagerContextSkeleton.java
	ManagerContextMessageReceiverInOut.java
	GetContextResponse.java
	GetContext.java
	ExtensionMapper.java

Uma aplicação simples foi criada para validar a interação entre os Serviços Web. A aplicação de gerenciamento de emergência, que previamente obteve a URL do Serviço Web, invoca a operação de consulta da classe *Stub* do Serviço Web para verificar a probabilidade de ocorrer acidentes como *boilover*, *flash over* e *backdraft*. A Figura 5.19

apresenta os trechos de código da aplicação que interage com os Serviços Web, invocando as operações por meio das chamadas de métodos.

Figura 5.19 – Fragmento de código da aplicação de gerenciamento de emergências.

```
private void okActionPerformed(java.awt.event.ActionEvent evt) {//GEN-FIRST:event_okActionPerformed
 // TODO add your handling code here:

  ManagerContextStub stub;
   List<GetContext> lista = new ArrayList();

   if (explosions.getSelectedItem().toString().equals("Boil over")) {

    try{
          Long seg = System.nanoTime();
          stub = new ManagerContextStub ("http://10.222.1.11:8080/axis2/services/managerContext");
          System.out.println("Comunicação: Aplicação --> Serviço Ger. Contexto. Via Web Service!!! ");

          for (int i=0; i<1; i++){
          ManagerContextStub.GetContext gt = new ManagerContextStub.GetContext();
          lista.add(gt);
          gt.setParam("Boil over");
          }

          for(int i=0; i<lista.size(); i++) {
          ManagerContextStub.GetContextResponse resultado = stub.getContext(lista.get(i));
          String resultadoGeral = resultado.get_return();
          result.setText(resultadoGeral);
          System.out.println("Resposta: "+ resultadoGeral);
          }
          System.out.println( (System.nanoTime()-seg) );
          }
          catch (Exception e){
          }
                    }
   if (explosions.getSelectedItem().toString().equals("Backdraft")) {

    try{
          Long seg = System.nanoTime();
          stub = new ManagerContextStub ("http://10.222.1.11:8080/axis2/services/managerContext");
          System.out.println("Comunicação: Aplicação --> Serviço Ger. Contexto. Via Web Service!!! ");

          for (int i=0; i<1; i++){
          ManagerContextStub.GetContext gt = new ManagerContextStub.GetContext();
          lista.add(gt);
          gt.setParam("Backdraft");
          }

          for(int i=0; i<lista.size(); i++) {
          ManagerContextStub.GetContextResponse resultado = stub.getContext(lista.get(i));
          String resultadoGeral = resultado.get_return();
          result.setText(resultadoGeral);
          System.out.println("Resposta: "+ resultadoGeral);
          }
          System.out.println( (System.nanoTime()-seg) );
          }
          catch (Exception e){
          }
```

```
                    }
    if (explosions.getSelectedItem().toString().equals("Flashover")) {

     try{
          Long seg = System.nanoTime();
          stub = new ManagerContextStub ("http://10.222.1.11:8080/axis2/services/managerContext");
          System.out.println("Comunicação: Aplicação --> Serviço Ger. Contexto. Via Web Service!!! ");

          for (int i=0; i<1; i++){
           ManagerContextStub.GetContext gt = new ManagerContextStub.GetContext();
           lista.add(gt);
           gt.setParam("Flashover");
           }

           for(int i=0; i<lista.size(); i++) {
           ManagerContextStub.GetContextResponse resultado = stub.getContext(lista.get(i));
          String resultadoGeral = resultado.get_return();
          result.setText(resultadoGeral);
          System.out.println("Resposta: "+ resultadoGeral);
          }
          System.out.println( (System.nanoTime()-seg) );
          }
          catch (Exception e){
          }
                    }
}//GEN-LAST:event_okActionPerformed
```

5.5.1 Avaliação do *Middleware*

A fim de avaliação foi possível medir a latência para verificar os possíveis acidentes *boilover*, *backdraft* e *flash over* pela aplicação de Gerenciamento de Emergência para os Serviços Web (Gerenciador de Contexto, *inferenceFuzzy*, *queryOntology* e Gerenciador de Agentes). Foi adicionado o valor da latência ao realizar uma tarefa de submissão para os agentes móveis, pelo Gerenciador de Agentes, para coletar informações da RSSFs. Foram geradas 1, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90 e 100 mensagens, para os Serviços Web localizados no servidor Tomcat 6.0, conforme apresenta a Tabela 11, Tabela 12 e Tabela 13 respectivamente.

Tabela 11 – Medidas realizadas para Boilover.

Mensagens	Serviços Web	Agente Móvel	Total (Ms)
1	316976114	0,7	317,676114
5	540310754	0,7	541,010754
10	1012310725	0,7	1013,010725
20	1558234287	0,7	1558,934287
30	2102922307	0,7	2103,622307

40	2818357006	0,7	2819,057006
50	3296525007	0,7	3297,225007
60	4057429539	0,7	4058,129539
70	4564248297	0,7	4564,948297
80	5786014273	0,7	5786,714273
90	6984522289	0,7	6985,222289
100	8734871835	0,7	8735,571835

Tabela 12 – Medidas realizadas para Backdraft.

Mensagens	Serviços Web	Agente Móvel	Total (Ms)
1	322267368	0,7	322,967368
5	537870857	0,7	538,570857
10	816238125	0,7	816,938125
20	1343980196	0,7	1344,680196
30	2068789483	0,7	2069,489483
40	2824644230	0,7	2825,34423
50	3322585810	0,7	3323,28581
60	4036313807	0,7	4037,013807
70	4769706634	0,7	4770,406634
80	5817330251	0,7	5818,030251
90	7048276802	0,7	7048,976802
100	9229377378	0,7	9230,077378

Tabela 13 – Medidas realizadas para Flahsover.

Mensagens	Serviços Web	Agente Móvel	Total (Ms)
1	309941325	0,7	310,641325
5	536379477	0,7	537,079477
10	1028814031	0,7	1029,514031
20	1592895174	0,7	1593,595174
30	2123919037	0,7	2124,619037
40	2604656772	0,7	2605,356772
50	3505412921	0,7	3506,112921
60	4052550587	0,7	4053,250587
70	4776981305	0,7	4777,681305
80	5777192333	0,7	5777,892333
90	7209447143	0,7	7210,147143
100	9697646242	0,7	9698,346242

A latência é uma métrica importante já que os serviços do *middleware* têm que responder em tempo real a situações de anormalidade ou emergência no ambiente físico. O gráfico da Figura 5.20 mostra uma avaliação em termos de latência.

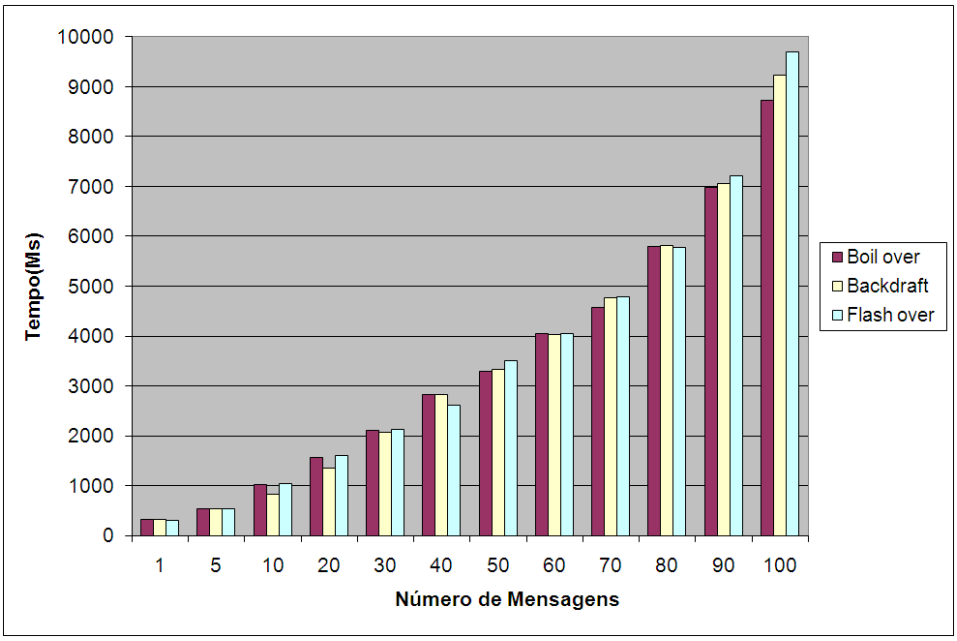

Figura 5.20 – Latência da interação entre aplicação, serviços e a RSSF.

Como observado, a latência aumenta conforme o número de solicitações aumenta. O tempo máximo obtido em 100 solicitações para verificar *flashover* foi de aproximadamente 10 segundos. Deve-se levar em consideração a complexidade dos serviços, em especial o Especialista, que realiza uma inferência em fuzzy e consulta a ontologias. Além de que os resultados foram obtidos na rede interna do Departamento de Computação da UFSCar onde a aplicação e os Serviços Web estavam em *hosts* diferentes. Uma análise mais criteriosa será realizada quando o *middleware* de nível 0 for implementado e integrado ao *middleware* de nível 1 e os acessos ao sistema realizados via Internet.

5.6 Considerações Finais

Neste capítulo foi descrita a arquitetura do *middleware* em dois níveis: nível 0 e nível 1. Os serviços e as tecnologias que envolvem cada nível e exemplos do funcionamento do *middleware* foram também aqui especificados. Parte do *middleware* de nível 1 foi implementado e avaliado, assim como foi analisado o custo da adição do TinyOS, do protocolo DAARP e o *publish/subscribe* na pilha de protocolos da RSSFs.

6. Conclusões

Neste trabalho foi projetado um *middleware* para suportar aplicações em redes de sensores sem fios. O *middleware* foi especificado em dois níveis:

- O nível 1 do *middleware* foi baseado na área de Serviços Web. O nível 1 foi projetado para atender as necessidades da classe de aplicações no domínio de emergência e provê interfaces reusáveis aos serviços do *middleware* às aplicações em mais alto nível, além de fornecer interoperabilidade entre diferentes plataformas de hardware e software.
- O *nível 0* do *middleware* foi baseado no paradigma *publish/subscribe*. O nível 0, foi projetado para prover maior flexibilidade e expressividade dos interesses das aplicações para a RSSF. Este paradigma foi utilizado por prover total desacoplamento entre produtores e requisitantes de informações, um requisito importante em ambientes como as RSSFs. Uma extensão deste paradigma é a variante baseado em conteúdo, que fornece maior dinâmica e flexibilidade nas subscrições realizadas pelas aplicações.

Foi implementado um protótipo com algumas funcionalidades do *middleware* de nível 1 para fim de avaliação. Deve-se, porém frisar que na rede foi utilizada uma solução de agentes móveis. Na avaliação realizada, a agregação de dados é feita por agentes móveis e não pelo protocolo DAARP. Isto se deve ao fato que no trabalho de agentes móveis realizado por [SPA 09] contêm uma simulação da latência envolvida para submeter uma tarefa e obter os resultados agregados na RSSFs, já que o *middleware* de nível 0 projetado ainda não se encontra implementado.

Realizou-se também um estudo da viabilidade em adicionar os componentes do TinyOS, o protocolo de rede DAARP e o *publish/subscribe* na pilha de protocolos básica para a RSSF.

Neste capítulo são descritas as contribuições geradas por este trabalho, trabalhos relacionados, além de relacionar os artigos submetidos e os trabalhos futuros, seguida de considerações finais.

6.1 Contribuições Geradas

Este trabalho traz as seguintes contribuições:

1. Projeto e especificação de um *middleware* para as RSSFs em multi-camadas. O *middleware* foi abstraído em dois níveis. O nível 1, refere-se à middleware de serviços e prove o acesso aos serviços de forma flexível e reusável, por meio da tecnologia de Serviços Web. O nível 0 refere-se à *middleware* de suporte e provê a RSSF com o paradigma *publish/subscribe* baseado em conteúdo. Esta variante provê maior expressividade e flexibilidade para as aplicações de RSSFs.
2. Integração no *middleware* de nível 1, os serviços podem ser acessados via Internet. Integração no *middleware* de nível 0 com uma solução mais na horizontal na pilha de protocolos. O *middleware* de nível 0 foi projetado para realizar a agregação de dados na camada de rede, utilizando o protocolo DAARP. A maioria das soluções de *middleware* para RSSFs realizam a agregação de dados como um serviço, disponibilizado na camada de aplicação.
3. Especificação dos acessos aos serviços do *middleware* (Gerenciador de Contexto, Especialista, Gravação, Reprodução, Adaptação, etc.).
4. Implementação parcial em Java dos serviços Gerenciador de Contexto, e para realizar a avaliação de desempenho o Gerenciador de Agentes.
5. Implementação parcial da aplicação de gerenciamento de emergências, para validar a arquitetura do *middleware* de nível 1, com as interfaces dos Serviços Web.
6. Geração dos Serviços Web com a API do Axis 2 versão 1.4.1, por meio dos documentos WSDL dos serviços (Gerenciador de Contexto, Especialista com *inferenceFuzzy* e *queryOntology*, Gerenciador de Agentes).
7. Especificação do funcionamento do *middleware* com diagramas de sequências.
8. Avaliação do protótipo em termos de latência. Algumas funcionalidades do *middleware* de nível 1 foram medidas, integrado com uma solução baseada em agentes móveis.

9. Viabilidade do custo em relação à memória utilizada no *middleware* de nível 0 com a adição do TinyOS, o protocolo DAARP e o *publish/subscribe*.

6.2 Trabalhos Relacionados

A Tabela 14 mostra as características de alguns dos projetos mencionados no Capítulo 3 e como eles são comparados com este trabalho.

Tabela 14 - Comparação entre as arquiteturas de *middlewares.*

Middleware	Comunicação no interior da RSSF	Interoperabilidade	Serviços Web
Sina	Scripts SQTL	-	-
Mires	Publish/Subscribe baseado em Tópico	-	-
Agilla	Tuple Space	RMI	-
[DEL 05]	SOAP, XML	Serviços Web	Interior da RSSF
Proposta deste trabalho	Publish/Subscribe baseado em conteúdo	Serviços Web	Fora da RSSF

Em [DEL 05] é apresentado uma arquitetura de *middleware* que provê a RSSF como Serviços Web para aplicações clientes. Em relação a este trabalho existem algumas diferenças na arquitetura de cada projeto. Em [DEL 05] fornece alguns serviços no *sink* e outros nos sensores e provê o acesso aos nós da RSSF via Serviços Web. A comunicação na rede e entrega de dados é ditada por um mecanismo de representação de dados e formatação de mensagens baseado em SOAP e XML. Neste trabalho os Serviços Web são disponibilizados apenas no *sink* e consistem de serviços que dão suporte a um mecanismo de interpretação de contexto em mais alto nível, que interpreta os dados recebidos da rede, inferindo com regras *fuzzy* e consulta a ontologias para a representação mais precisa do que está acontecendo no ambiente. Além disso, neste trabalho, o mecanismo de comunicação no interior da RSSF é baseado no *publish/subscribe* baseado em conteúdo.

O Mires é um *middleware* para RSSF baseado no paradigma *publish/subscribe* que desacopla os produtores e consumidores de dados e utiliza um variante baseado em tópico. Em relação a este trabalho foi adotado outra variante do paradigma *publish/subscribe* baseado em conteúdo. A abordagem baseado em conteúdo fornece mais expressividade e

flexibilidade para as aplicações ao solicitar tarefas na RSSF em relação à baseada em Tópico. Além disso, a carga de trabalho solicitada pelas aplicações pode crescer muito e pode ficar ingerenciável com a abordagem baseada em tópico.

A Tabela 15 compara os *middlewares* descritos no Capítulo 3 em relação aos desafios identificados neste trabalho.

Tabela 15 - Comparação entre os *middlewares* para RSSF e a proposta deste trabalho. (Adaptado de [MOL 06])

Desafios	Impala	Mires	SINA	Agilla	Maté	MiLAN	TinyCubus	Proposta deste trabalho
Abstração	✓	✓	✓	✓	✓		✓	✓
Fusão/Agregação de Dados	✓	✓	✓	✓				✓
Restrições de Recursos	✓	✓	✓	✓	✓	✓	✓	✓
Topologia Dinâmica	✓		✓	✓			✓	
Conhecimento da Aplicação						✓	✓	
Paradigma de Programação	✓	✓	✓	✓	✓	✓	✓	✓
Adaptabilidade	✓			✓	✓	✓	✓	✓
Escalabilidade	✓	✓	✓	✓		✓		✓
Segurança					✓			
QoS						✓		
Integração				✓				✓

Uma importante característica deste trabalho é a integração da RSSF com outras existentes infra-estruturas de redes, por exemplo, a Internet. A abordagem orientada a serviços para implementar integração a RSSF é baseado sobre tecnologias arquiteturais de padrões aberto tal como os Serviços Web. No nível 0 do *middleware*, a agregação de dados é projetada em uma abordagem *cross-layer* e interoperabiliza com o esquema *publish/subscribe* baseado em conteúdo e o sistema operacional TinyOS .

6.3 Publicações Submetidas

RIBEIRO, J. E.; ARAUJO, R. B. A multi-layer middleware for Wireless Sensor Networks. In: INTERNATIONAL MIDDLEWARE CONFERENCE, 11., 2010, Bangalore. **Proceedings...** Bangalore: MIDDLEWARE, 2010.

6.4 Trabalhos Futuros

Neste trabalho foi especificado um *middleware* para suportar aplicações em RSSFs. Foram implementados partes do serviço Gerenciador de Contexto e geradas as interfaces dos Serviços Web para os serviços (Gerenciador de Contexto, Especialista com *inferenceFuzzy* e *queryOntology*). A parte não implementada, relacionada com os outros serviços do *middleware* de nível 1 e também ao *middleware* de nível 0, é prevista como trabalhos futuros.

Como continuidade ao trabalho aqui iniciado, as seguintes atividades ainda deverão ser realizadas.

- Implementação do *middleware* de nível 0;
- Integração e geração das *interfaces* de acesso dos serviços de gravação e reprodução no nível 1 do *middleware*;
- Avaliação de desempenho do *middleware* com todos os serviços implementados.
- Avaliação da abordagem baseado em *publish/subscribe* por conteúdo em comparação com as soluções de *middleware* baseado no *publish/subscribe* por tópico.

6.5 Considerações Finais

Conclui-se ao final deste trabalho, que sistemas de *middleware* podem ser utilizados com sucesso para reduzir a “lacuna” entre a rede de sensores e as aplicações.

Este trabalho apresentou o projeto de um *middleware* para RSSFs abstraído em dois níveis. O *middleware* é genérico o suficiente para atender aplicações complexas e simples.

O paradigma *publish/subscribe* baseado em conteúdo foi projetado na rede de sensores e proporciona maior expressividade e flexibilidade para as subscrições e publicações. Como energia é um recurso escasso nos nós sensores o *middleware* pode auxiliar no prolongamento de vida da rede. O *publish/subscribe* baseado em conteúdo proporciona um menor número de mensagens geradas na rede, já que as mensagens são transmitidas na rede apenas se os valores percebidos pelos nós sensores satisfazem as restrições das subscrições.

Outra consideração importante refere-se ao fato do uso de tecnologias padrões da Web no nível 1 do *middleware*. Com os Serviços Web, flexibilidade e reusabilidade podem ser oferecidas para as aplicações interessadas nos serviços disponibilizados no *middleware*, necessitando apenas acessá-los via Internet.

Este trabalho é parte importante de um projeto maior, cujo objetivo é reduzir riscos de perdas de vida e patrimônio em situações de emergência.

7. Referências Bibliográficas

[AGR 00] AGRE, J.; CLARE, L. An Integrated Architecture for Cooperative Sensing Networks. **Computer**, California, v. 33, n. 5, p. 106 - 108, 2000.

[AGU 99] AGUILERA, M. K.; STROM, R. E.; STURMAN, D. C.; ASTLEY, M.; CHANDRA, T. D. Matching events in a content-based subscription system. In: ANNUAL ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 18., 1999, Atlanta. **Proceedings…** Atlanta:PODC, 1999. p. 53 - 61.

[ALE 04] ALEX, H.; KUMAR, M.; SHIRAZI, B. MidFusion: middleware for information fusion in sensor network applications. In: INTELLIGENT SENSORS, SENSOR NETWORKS AND INFORMATION PROCESSING CONFERENCE, 2004, Arlington. **Proceedings…** Arlington:ISSNIP, 2004. p. 617 - 622.

[AKY 02] AKYILDIZ, I. F.; SU, W.; SANKARASUBRAMANIAM, Y.; CAYIRCI, E. A Survey on sensor networks. **IEEE Communications Magazine**, p. 102 - 114, 2002.

[AKY 04] AKYILDIZ, I. F.; KASIMOGLU, I. H. Wireless sensor and actor networks: research challenges. **Elsevier Ad hoc Network Journal**, v. 2, n. 4, p. 351-367, 2004.

[AMO 04] AMORIM, S. **A tecnologia Web Services e sua aplicação num sistema de gerência de telecomunicações**. 2004. 110 f. Dissertação (Mestrado em Computação) – Departamento de Computação, Universidade Estadual de Campinas, Campinas, 2004.

[AXI 09] Axis Apache. **Web Services – Axis**. Disponível em: <http://ws.apache.org/axis/>. Acesso em: 20 fev. 2009.

[BAR 05] BARBOSA, T.M.G.A.; SENE JR., I. G.; CARVALHO, H. S.; DA ROCHA, A. F.; NASCIMENTO, F. A. O. Arquitetura de software para Redes de Sensores Sem Fios: A proeminência do middleware. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 25., 2005, São Leopoldo. **Anais…** São Leopoldo:UNISINOS, 2005. p. 617 - 622.

[BAY 99] BAYERDORFFER, B. Distributed programming with associative broadcast. In: ANNUAL HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, 37., 1994, Austin. **Proceedings…** Austin:HICSS, 1994. p. 353 - 362.

[BLA 04] BLAIR, G.; CAMPBELL, A. T.; SCHMIDT, D. C. Middleware technologies for future communication networks. **IEEE Network**, v. 18, n.1, p. 4 - 5, 2004.

[BON 01] BONNET, P.; GEHRKE, J.; SESHADRI, P. Towards sensor database systems. In: INTERNATIONAL CONFERENCE ON MOBILE DATA MANAGEMENT, 2., 2001, Hon Kong. **Proceedings…** Hong Kong: MDM, 2001. p. 3 - 14.

[BUL 01] BULUSU, N.; ESTRIN, D.; GIROD, L.; HEIDEMANN, J. Scalable coordination for wireless sensor networks: self-configuring localization systems. In: INTERNATIONAL SYMPOSYUM ON COMUNICATION THEORY AND APPLICATION, 6., 2001, Umbleside. **Proceedings…** Umbleside:ISCTA, 2001. p. 6.

[BUO 01] BUONADONNA, P.; HILL, J.; CULLER, D. Active Message Communication for Tiny Networked Sensors. In: ANNUAL JOINT CONFERENCE COMPUTER AND COMMUNICATIONS SOCIETIES, 21., 2001, Anchorage. **Proceedings…** Anchorage:INFOCOM, 2001. p. 1 - 11.

[CAL 04] CALABREZ, C.E. **Uma comparação entre diversas tecnologias de comunicação de objetos distribuídos em Java**. 2004. 104 f. Dissertação

(Mestrado em Computação) – Departamento de Computação, Universidade Estadual de Campinas, Campinas, 2004.

[CAM 09] CAMPOS, M.R. **Projeto e implementação de um serviço de interpretação de contexto em apoio à Preparação e Resposta a Emergências**. 2009. 143 f. Dissertação (Mestrado em Computação) – Departamento de Computação, Universidade Federal de São Carlos, São Carlos, 2009.

[CER 01] CERPA, A.; ELSON, J.; ESTRIN, D.; GIROD, L.; HAMILTON, M.; ZHAO, J. Habitat monitoring: application driver for wireless communications technology. **ACM SIGCOMM Computer Communication Review**, San Jose, v. 31, n. 2, p. 20 - 41, 2001.

[CHE 04] CHEN, D.; VARSHNEY, P. K. QoS Support in Wireless Sensor Network: A Survey. In: INTERNATIONAL CONFERENCE ON WIRELESS NETWORKS, 2004, Las Vegas. **Proceedings…** Las Vegas:ICWN, 2004. p. 227 - 233.

[CHO 01] CHO, S. H.; CHANDRAKASAN, A. Energy efficient protocols for low duty cycle wireless microsensor networks. In: HAWAII INTERNATIONAL CONFERENCE IN SYSTEM SCIENCES, 33., 2001, Hawaii. **Proceedings…** Hawaii:HICSS, 2001. p. 785 - 790.

[CRU 08] CRUZ, S.M.S. **Serviços Web – uma breve introdução**. Disponível em: <https://www.nce.ufrj.br/conceito/artigos/2005/01p1-1.htm>. Acesso em: 17 abr. 2008.

[CUR 05] CURINO, C.; GIANI, M.; GIORGETTA, M.; GIUSTI, A.; MURPHY, A. L.; PICCO, G. P. TinyLIME: bridging mobile and sensor networks through Middleware. In: IEEE INTERNATIONAL CONFERENCE ON PERVASIVE COMPUTING AND COMMUNICATIONS, 30., 2005, Hawaii. **Proceedings…** Hawaii:PERCOM, 2005. p. 61 - 72.

[DEL 05] DELICATO, F.C. **Middleware baseado em serviços para redes de sensores sem fios**. 2005. 209 f. Tese (Doutorado em Computação) – Departamento de Engenharia Elétrica, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2005.

[DUN 04] DUNKELS, A.; GRONVALL, B.; VOIGT, T. Contiki – a lightweight and flexible operating system for tiny networked sensors. In: CONFERENCE LOCAL COMPUTER NETWORKS, 29., 2004, Washington. **Proceedings…** Washington:LCN, 2004. p. 455 - 462.

[ELS 01] ELSON, J.; ESTRIN, D. Random, ephemeral transaction identifers in dynamic sensor networks. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, 21., 2001, Phoenix. **Proceedings…** Phoenix:ICDSC, 2001. p. 459 – 468.

[END 06] ENDLER, M. Large scale body sensing for Infectious Disease Control. **Sentient Future Competition on European Workshop on Wireless Sensor Networks**. Zurich, 2006.

[ERL 04] ERL, Thomas. Introduction to Web Services Technologies. **Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services**. New Jersey: Prentice Hall PTR, 2004. p. 47-127.

[EST 99] ESTRIN, D.; GOVINDAN, R.; HEIDEMANN, J.; KUMAR, S. Next Century challenges: Scalable Coordination in Sensor Network. In: ACM/IEEE INTERNATIONAL CONFERENCE ON MOBILE COMPUTING AND NETWORKING, 5., 1999, Seattle. **Proceedings…** Seattle:MOBICOM, 1999. p. 263 - 270.

[EST 00] ESTRIN, D.; GOVINDAN, R.; HEIDEMANN, J. Embedding the Internet, **Communication of the ACM**, v. 43, n.5, p. 38 - 41, 2000.

[EUG 03] EUGSTER, P.T.; FELBER, P. A.; GUERRAOUI, R. The many faces of publish/subscribe. **ACM Computing Surveys (CSUR)**, v. 35, n. 2, p. 114 – 131, 2003.

[EUG 07] EUGSTER, P. Type-based publish/subscribe: concepts and experiences. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, v. 29, n. 1, p. 1-50, 2007.

[FOK 05] FOK, C.; ROMAN, G.; LU, C. Rapid development and flexible deployment of adaptive wireless sensor network applications. In: IEEE INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, 25., 2005, Washington. . **Proceedings…** Washington:ICDCS, 2005. p. 653 - 662.

[GIR 01] GIROD, L.; ESTRIN, D. Robust range estimation using acoustic and multimodal sensing. In: IEEE/RSJ INTERNATIONAL CONFERENCE ON INTELLIGENT ROBOTS AND SYSTEMS, 2001, Maui. **Proceedings…** Maui:IROS, 2001. p. 1312 – 1320.

[GOM 04] GOMES, M.; Gonçalvez, A. L. Web Services: uma abordagem teórica. **Revista de divulgação técnico-científica do ICPG ISSN 1807- 2836,** v. 2, n. 5, 2004

[GUI 06] GUIMARÃES, G.; SOUTO, E.; VIEIRA, M.; VASCONCELOS, G.; ROSA, N.; FERRAZ, C.; KELNER, J. Middleware para redes de sensores sem-fio: projeto, implementação e avaliação de consumo de energia. In: Simpósio Brasileiro de Redes de Computadores, 24., 2006, Curitiba. **Anais…** Curitiba: PUCPR, 2006. p. 1 - 16.

[HAD 06] HADIM, S.; MOHAMED, N. Middleware for wireless sensor networks: a survey. In: INTERNATIONAL CONFERENCE ON COMUNICATION SYSTEM SOFTWARE AND MIDDEWARE, 1., 2006, New Delli. **Proceedings…** New Delli:COMSWARE, 2006. p. 1 - 7.

[HAN 05] HAN, C. C.; KUMAR, R.; SHEA, R.; KOHLER, E.; SRIVASTAVA, M. A dynamic operating system for sensor nodes. In: INTERNATIONAL CONFERENCE ON MOBILE SYSTEMS, APPLICATIONS, AND SERVICES, 3., 2005, Seattle. **Proceedings…** Seattle:MOBISYS, 2005. p. 163-176.

[HAU 08] HAUER, J.; HANDZISKI, V.; KOPKE, A.; WILLING, A.; WOLISZ, A. A component framework for content-based publish/subscribe in sensor networks. In: EUROPEAN CONFERENCE WIRELESS SENSOR NETWORKS, 5., 2008, Bologna. **Proceedings…** Bologna:EWSN, 2008. p. 369 - 385.

[HEI 03] HEIDMAN, J.; SILVA, F.; ESTRIN, D. Matching Data Dissemination Algorithms to Application Requeriments, In: INTERNATIONAL CONFERENCE ON EMBEDDED NETWORKED SENSOR SYSTEMS, 1., 2003, Los Angeles. **Proceedings…** Los Angeles:ISITR, 2003. p. 218 - 229.

[HEI 04] HEINZELMAN, W. B.; MURPHY, A. L.; CARVALHO, H. S.; PERILLO, M. A. Middleware to support sensor network applications. **IEEE Network**, New York, v. 18, n. 1, p. 6 - 14, 2004.

[HEN 06] HENRICKSEN, K.; ROBINSON, R. A survey of middleware for sensor networks: state-of-the-art and future directions. In: INTERNATIONAL WORKSHOP ON MIDDLEWARE FOR SENSOR NETWORKS, 2006, Melbourne. **Proceedings…** Melbourne:MIDSENS, 2006. p. 60 - 65.

[HIL 00] HILL, J.; SZEWCZYK, R.; WOO, A.; CULLER, D.; PISTER, K. System architecture directions for networked sensors. In: INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 9., 2000, Cambridge. **Proceedings…** Cambridge:ASPLOS, 2000. p. 93 - 104.

[HOB 00] HOBLOS, G.; STAROSWIECHI, M.; AITOUCHE, A. Optimal design of fault tolerant sensor networks. In: INTERNATIONAL CONFERENCE IEEE ON

CONTROL APPLICATIONS, 2000, Anchorage. **Proceedings…** Anchorage:CCA, 2000. p. 467 - 472.

[INF 02] INFORMATIK, V. F. **Large-Scale content-based publish/subscribe systems**. 2002. 200 f. Dissertação de (Mestrado) - Departamento Técnico da Universidade de Darmstad, Darmstadt, 2002.

[JUA 02] JUANG, P.; OKI, H.; WANG, Y.; MARTONOSI, M.; PEH, L. S.; RUBENSTEIN, D. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with ZebraNet. In: ANNUAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS. 10., 2002, San Jose. **Proceedings…** San Jose:ASPLOS, 2002. p. 96 - 107.

[KAH 99] KAHN, J.; KATZ, R.; PISTER, K. Next Century Challenges: Mobile Networking for Smart Dust. In: ANNUAL INTERNATIONAL CONFERENCE ACM ON MOBILE COMPUTING AND NETWORKING, 1999, Washington. **Proceedings…**, Washington:MobiCom, 1999. p 271-278.

[KAL 97] KALYANASUNDARAMM, P. SETHI, A. S.; SHERWIN, C. M.; ZHU, D. A spreadsheet-based scripting environment for SNMP. In: INTERNATIONAL SYMPOSIUM IFIP/IEEE ON INTEGRATED NETWORK MANAGEMENT V: INTEGRATED MANAGEMENT IN A VIRTUAL WORLD. 15., 1997, San Diego. **Proceedings…** San Diego:IM, 1997. p. 752 - 765.

[KIN 00] KINAWI, H.; TAHA, M. M.; EL-SHEIMY, N. Gpsr: greedy perimeter stateless routing for wireless networks. In: ANNUAL ACM/IEEE INTERNATIONAL CONFERENCE ON MOBILE COMPUTING AND NETWORKING, 6., 2000, Boston. **Proceedings…** Boston: MobiCom, 2000. p. 243 - 254.

[KRO 03] KROPIWIEC, C.D. **Proposta de um mecanismo de segurança para web services baseado em Ipsec**. 2003. 193 f. Dissertação (Mestrado em

Computação) - Departamento de Computação, Universidade Católica do Paraná, Curitiba, 2003.

[LEV 02] LEVIS, P.; CULLER, D. Maté: A tiny virtual machine for sensor networks. In: INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 10., 2002, San Jose. **Proceedings…** San Jose:ASPLOS, 2002. p. 85 - 95.

[LEW 10] LEWIS, F. Introduction to crossbow mica2 sensors. **EE5369 Sensor Networks**. 2010.

[LIU 03] LIU, T.; MARTONOSI, M. Impala: a middleware system for managing autonomic, parallel sensor systems. In: SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 9., 2003, San Diego. **Proceedings…** San Diego: SIGPLAN, 2003. p. 107 - 118.

[LOP 06] LOPES, A. R. **Projeto de um Ambiente 3D de Visualização e Reprodução de Eventos Capturados e Interpretados a Partir de Ambientes Físicos Cientes de Contexto para Aplicações de Preparação para Emergência.** 2006. 96 f. Dissertação (Mestrado em Computação) – Departamento de Computação, Universidade Federal do São Carlos, São Carlos, 2006.

[LOU 03] LOUREIRO, A. F. A.; NOGUEIRA, J. M. S.; RUIZ, L. B.; DE FREITAS MINI, R. A.; NAKAMURA, E. F.; FIGUEIREDO, C. M. S. Redes de Sensores Sem Fio. In: Simpósio Brasileiro de Redes de Computadores, 21., 2003, Natal. **Anais…** Natal:SBRC, 2003. p. 179 - 226.

[MAD 05] MADDEN, S. R. FRANKLIN, M. J.; HELLERSTEIN, J. M.; HONG, W. TinyDB: an acquisitional query processing system for sensor networks. **ACM Transactions on Database Systems (TODS)**, v. 30, n. 1, p. 122 - 173, 2005.

[MAI 02] MAINWARING, A.; POLASTRE, J.; SZEWCYK, R.; CULLER, D. ANDERSON, J. Wireless sensor networks for habitat monitoring. In: WORKSHOP ON WIRELESS SENSOR NETWORKS AND

APPLICATIONS, 1., 2002, Altanta. **Proceedings…** Atlanta:W2SN, 2002. p. 88 - 97.

[MAR 05a] MARRON, P. J. Middleware Approaches for Sensor Networks. **Summer School on WSNs and Smart Objects Schloss Dagstuhl**, Germany, 2005

[MAR 05b] MARRON, P. J.; LACHENMANN, A.; MINDER, D.; HAHNER, J.; SAUTER, R.; ROTHERMEL, K. TinyCubus: a flexible and Adaptive framework for sensor networks. In: EUROPEAN WORKSHOP ON WIRELESS SENSOR NETWORKS, 2., 2005, Istanbul:EWSN. **Proceedings…**, Istanbul, 2005. p. 278 – 289.

[GRA 05] GRAHAM, Steve. Building Web Services with Java: **Making Sense of XML, SOAP, WSDL, and UDDI**. 2. ed. Sams Publishing, 2005.

[MAS 02] MASCOLO, C.; CAPRA, L.; EMMERICH, W. Middleware for mobile computing (A Survey). In: ADVANCED LECTURES ON NETWORKING 2002, TUTORIALS, 2002, Pisa. . **Proceedings…** Pisa:NETWORKING, 2002. p. 20 - 58.

[MAS 07] MASRI, W.; MAMMERI, Z. Middleware for wireless sensor networks: In: INTERNATIONAL CONFERENCE ON NETWORK AND PARALLEL COMPUTING, 2007, Liaoning. **Proceedings…** Liaoning: IFIP, 2007. p. 349 - 356.

[MEL 06] MELLO, E. R.; WANGHAM, M. S.; FRAGA, J. S.; CAMARGO, E. T. Segurança em Serviços Web. **Minicurso do SBSeg - Sociedade Brasileira de Computação**, Porto Alegre, p. 1- 48, 2006.

[MIA 08] MIAOMIAO, W.; JIANNONG, C.; JING, L.; SAJAL, D. Middleware for wireless sensor networks: A survey. **Jornal of Computer Science and Technology**, v. 23, n. 3, p. 305 - 326, 2008.

[MIN 01] MIN, R.; BHARDWAJ, M.; CHO, S.; SINHA, A.; SHIH, E.; WANG, A; CHANDRAKASAN, A. Low-Power Wireless Sensor Networks, **VLSI Design**, 2001.

[MIN 02] MINI, R. A. F.; NATH, B.; LOUREIRO, A. A. F. A probabilist approach to predict the energy consumption in wireless sensor networks. In: WORKSHOP DE COMUNICAÇÃO SEM FIO E COMPUTAÇÃO MÓVEL, 4., 2002, São Paulo. **Proceedings…** São Paulo:WCSF, 2002. p. 232 - 245.

[MOL 06] MOLLA, M. M.; AHAMED, S. I. A Survey of middleware for sensor network and challenges. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 2006, Milwaukee. **Proceedings…** Milwaukee:ICPP, 2006. p. 223 - 228.

[MON 97] MONTEZ, C. **Um Modelo de Programação para Aplicações de Tempo Real em Sistemas Abertos**. 1997. Monografia (Exame de Qualificação de Doutorado) – Departamento de Automação e Sistemas, Universidade Federal de Santa Catarina, Santa Catarina, 1997.

[OAS 08] OASIS. **UDDI Specification TC**. Disponível em: <http://www.oasis-open.org/committees/uddi-spec/faq.php>. Acesso em: 18 abr. 2008.

[PIN 04] PINTO, A. J. G. **Mecanismo de Agregação de Dados Empregando Técnicas Paramétricas em Redes de Sensores**. 2004. 140f. Dissertação (Mestrado em Computação), Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2004.

[POT 00] POTTIE, G. J.; KAISER, W. J. Wireless integrated networks sensors. **Communications of the ACM**, v. 43, n.5, p. 51 - 58, 2000.

[RAG 02] RAGHUNATHAN, V.; SCHURGERS, C.; PARK, S.; SRIVASTAVA, M. Energy Aware Wireless Microsensor Networks. **IEEE Signal Processing Magazine**, p. 40-50, 2002.

[ROC 09a] ROCA, I.; KOFUJI, S. T. Abordagem dos sistemas operacionais para redes de sensores sem fio. **Relatório Técnico - Laboratório de Sistemas Integráveis**, 2009.

[ROC 09b] ROCHA, R. V. **Uma arquitetura de suporte a modelagem de simulações de treinamento baseada na arquitetura HLA (*High Level Architecture*)**. 2009. 164 f. Dissertação (Mestrado em Computação) – Departamento de Computação, Universidade Federal de São Carlos, São Carlos, 2009.

[ROM 02] ROMER, K.; KASTEN, O.; MATTERN, F. Middleware challenges for wireless sensor networks. **ACM Mobile Computing and Communications Review**, v. 6, n. 4, p. 59 - 61, 2002.

[RUI 04a] RUIZ, L. B.; NOUGUEIRA, J. M. S.; LOUREIRO, A. A. Handbook of Sensor Network: Compact Wireless and Wired Sensing Systems, **CRCPress,** Florida, p. 3.1 – 3.25, 2004.

[RUI 04b] Ruiz, L. B.; CORREIA, L. H. A.; VIEIRA, L. F. P.; MACEDO, D. F.; NAKAMURA, E. F.; FIGUEIREDO, C. M. S.; VIEIRA, M. A. M.; MECHELANE, E. H.; CAMARA, D.; LOUREIRO, A. A. F. NOGUEIRA, J. M. S.; DA SILVA JR. D. C. Arquitetura para Rede de Sensores Sem Fio. In: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES, 28., 2004, Gramado. **Anais…** Gramado:SBRC, 2004. p. 167 - 218.

[SHE 01] SHEN, C.; SRISATHAPORNPHT, C.; JAIKAE, C. Sensor information networking architecture and applications. **IEEE Personal Communications**, p. 52–59, 2001.

[SIC 09] INSTITUTE OF COMPUTER SCIENCE SWEDISH. **Contiki – The operating system for embedded smart objects**. Disponível em: <http://www.sics.se/contiki/about-contiki.html>. Acesso em: 22 jan. 2010.

[SOU 05] SOUTO, E.; GIMARAES, G.; VASCONCELOS, G.; VIEIRA, M.; ROSA, N.; FERRAZ, C.; KELNER, J. Mires: a publish/subscribe middleware for sensor networks. **Personal and Ubiquitous Computing,** v. 10, n. 1, p. 37 - 44, 2005.

[SPA 09] SPADONI, I.M. B. **Uma solução baseada em agentes para redes de sensores sem fios.** 2009. 111 f. Dissertação (Mestrado em Computação) – Departamento de Computação, Universidade Federal do São Carlos, São Carlos, 2009.

[SRI 00] SRISATHAPORNPHAT, C.; JAIKAEO, C.; SHEN, C. Sensor information networking architecture. In: INTERNATIONAL WORKSHOP ON PARALLEL PROCESSING, 2000, Toronto. **Proceedings…** Toronto:ICPP, 2000. p. 23 - 30.

[TIL 02] TILAK, S.; ABU-GHAZALEH, N; HEINZELMAN, W. A taxonomy of wireless micro-sensor network models. **ACM SIGMOBILE Mobile Computing and Communications Review**, v.6, n.2, p. 28 – 36, 2002.

[TOM 09] Apache Jakarta TomCat Project. Disponível em: <http://tomcat.apache.org/>. Acesso em: 10 mar. 2009.

[VAL 03] VALENTE, M. T. O.; SANTOS, J. P.; COUTO, C. Interceptação de Métodos Remotos em Java RMI. In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO, 7., 2003, Ouro Preto. , 2003. **Anais…** Ouro Preto:SBLP, 2003. p. 50-63.

[VIL 07] VILLAS, L. A. **Protocolos de Roteamento Cientes de QoS para Redes de Sensores e Atuadores Sem Fio**. 2007. 105f. Dissertação (Mestrado em Computação) - Departamento de Computação. Universidade Federal de São Carlos, São Carlos, 2007.

[VIL 09] VILLAS, L. A.; BOUKERCHE, A.; ARAUJO, R. B.; LOUREIRO, A. A. F. A reliable and data aggregation aware routing protocol for wireless sensor network. In: INTERNATIONAL CONFERENCE ON MODELING,

ANALYSIS AND SIMULATION OF WIRELESS AND MOBILE SYSTEMS, 12., 2009, Tenerife. **Proceedings…** Tenerife:MSWiW, 2009. p. 242 - 252.

[WAN 08] WANNER, L. F.; FROHLICH, A. A. Suporte de Sistema Operacional para Redes de Sensores Sem Fio. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO – WORKSHOP DE SISTEMAS OPERACIONAIS, 28., 2008, Belém do Pará. **Anais…** Belém do Pará:SBC, 2008. p. 123 -134.

[WAR 01] WANEKE, B.; LAST, M.; LIEBOWITZ, B.; PISTER, K. S. J. Smart dust: Communicating with a cubic-millimeter computer. **Computer**, v. 34, n. 1, p. 44 - 51, 2001.

[W3C 08a] W3C. **Soap version 1.2 part 0: Primer**. Disponível em: <http://www.w3.org/TR/soap12-part0/>. Acesso em: 25 abr. 2008.

[W3C 08b] W3C. **Web services description language (WSDL) version 2.0 part 1: core language.** Disponível em: <http://www.w3.org/TR/wsdl20/>. Acesso em: 05 mai. 2008.

[YAO 02] YAO, Y.; GEHRKE, J. The cougar approach to in-network query processing in sensor networks. **ACM SIGMOD Record**, v. 31, n. 3, p. 9 - 18, set. 2002.

[YON 03] YONEKI, E. Mobile applications with a middleware system in publishsubscribe paradigm. In: WORKSHOP ON APPLICATIONS AND SERVICES IN WIRELESS NETWORKS, 3., 2003, Suíca. **Proceedings…** Suíca:ASWN, 2003.

[YOU 02] YOUSSEF, M.; YOUNIS, M.; ARISHA, K. A constrained shortest-path energy-aware routing algorithm for wireless sensor networks. In: WIRELESS COMMUNICATIONS AND NETWORKING CONFERENCE, 2002, Orlando. **Proceedings…** Orlando:WCNC, 2002. p. 17 - 21.

[YU 04] YU, Y.; KRISHNAMACHARI, B.; PRASANA, V.K. Issues in designing middleware for wireless sensor networks. **IEEE Network**, v.18, n.1, p.15 - 21, 2004.

APÊNDICE A – Documento WSDL

inferenceFuzzy

```
<?xml version="1.0" encoding="UTF-8" ?>
- <wsdl:definitions targetNamespace="http://fuzzy.controladorSIC" xmlns:ns1="http://org.apache.axis2/xsd"
xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" xmlns:xsd="http://fuzzy.controladorSIC" xmlns:ax21="http://fuzzy.nrc/xsd"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/" xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/">
- <wsdl:types>
- <xs:schema attributeFormDefault="qualified" elementFormDefault="qualified" targetNamespace="http://fuzzy.nrc/xsd"
xmlns:ax22="http://fuzzy.controladorSIC">
 <xs:import namespace="http://fuzzy.controladorSIC" />
- <xs:complexType name="FuzzyException">
- <xs:complexContent>
- <xs:extension base="ax22:Exception">
 <xs:sequence />
 </xs:extension>
 </xs:complexContent>
 </xs:complexType>
 </xs:schema>
- <xs:schema attributeFormDefault="qualified" elementFormDefault="qualified" targetNamespace="http://fuzzy.controladorSIC"
xmlns:ax23="http://fuzzy.nrc/xsd">
 <xs:import namespace="http://fuzzy.nrc/xsd" />
- <xs:complexType name="Exception">
- <xs:sequence>
 <xs:element minOccurs="0" name="Exception" nillable="true" type="xs:anyType" />
 </xs:sequence>
 </xs:complexType>
- <xs:element name="FuzzyException">
- <xs:complexType>
- <xs:sequence>
 <xs:element minOccurs="0" name="FuzzyException" nillable="true" type="ax21:FuzzyException" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
- <xs:element name="hasBoilovel">
- <xs:complexType>
- <xs:sequence>
 <xs:element minOccurs="0" name="urlEmerg" nillable="true" type="xs:string" />
 <xs:element minOccurs="0" name="TemperatureTank" type="xs:double" />
 <xs:element minOccurs="0" name="PercentageWater" type="xs:double" />
```

```
 <xs:element minOccurs="0" name="MaterialTank" nillable="true" type="xs:string" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
- <xs:element name="hasBoilovelResponse">
- <xs:complexType>
- <xs:sequence>
 <xs:element minOccurs="0" name="return" nillable="true" type="xs:string" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
- <xs:element name="hasFire">
- <xs:complexType>
- <xs:sequence>
 <xs:element minOccurs="0" name="urlEmerg" nillable="true" type="xs:string" />
 <xs:element minOccurs="0" name="Oxygen" type="xs:double" />
 <xs:element minOccurs="0" name="Temperature" type="xs:double" />
 <xs:element minOccurs="0" name="Smoke" type="xs:double" />
 <xs:element minOccurs="0" name="Material" nillable="true" type="xs:string" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
- <xs:element name="hasFireResponse">
- <xs:complexType>
- <xs:sequence>
 <xs:element minOccurs="0" name="return" nillable="true" type="xs:string" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
 </xs:schema>
 </wsdl:types>
- <wsdl:message name="hasFireResponse">
 <wsdl:part name="parameters" element="xsd:hasFireResponse" />
 </wsdl:message>
- <wsdl:message name="FuzzyException">
 <wsdl:part name="parameters" element="xsd:FuzzyException" />
 </wsdl:message>
- <wsdl:message name="hasBoilovelResponse">
 <wsdl:part name="parameters" element="xsd:hasBoilovelResponse" />
 </wsdl:message>
- <wsdl:message name="hasFireRequest">
 <wsdl:part name="parameters" element="xsd:hasFire" />
 </wsdl:message>
- <wsdl:message name="hasBoilovelRequest">
 <wsdl:part name="parameters" element="xsd:hasBoilovel" />
 </wsdl:message>
- <wsdl:portType name="inferenceFuzzyPortType">
```

```
- <wsdl:operation name="hasFire">
  <wsdl:input message="xsd:hasFireRequest" wsaw:Action="urn:hasFire" />
  <wsdl:output message="xsd:hasFireResponse" wsaw:Action="urn:hasFireResponse" />
  <wsdl:fault name="FuzzyException" message="xsd:FuzzyException" wsaw:Action="urn:hasFireFuzzyException" />
  </wsdl:operation>
- <wsdl:operation name="hasBoilovel">
  <wsdl:input message="xsd:hasBoilovelRequest" wsaw:Action="urn:hasBoilovel" />
  <wsdl:output message="xsd:hasBoilovelResponse" wsaw:Action="urn:hasBoilovelResponse" />
  <wsdl:fault name="FuzzyException" message="xsd:FuzzyException" wsaw:Action="urn:hasBoilovelFuzzyException" />
  </wsdl:operation>
  </wsdl:portType>
- <wsdl:binding name="inferenceFuzzyHttpBinding" type="xsd:inferenceFuzzyPortType">
  <http:binding verb="POST" />
- <wsdl:operation name="hasFire">
  <http:operation location="inferenceFuzzy/hasFire" />
- <wsdl:input>
  <mime:content part="hasFire" type="text/xml" />
  </wsdl:input>
- <wsdl:output>
  <mime:content part="hasFire" type="text/xml" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="hasBoilovel">
  <http:operation location="inferenceFuzzy/hasBoilovel" />
- <wsdl:input>
  <mime:content part="hasBoilovel" type="text/xml" />
  </wsdl:input>
- <wsdl:output>
  <mime:content part="hasBoilovel" type="text/xml" />
  </wsdl:output>
  </wsdl:operation>
  </wsdl:binding>
- <wsdl:binding name="inferenceFuzzySoap12Binding" type="xsd:inferenceFuzzyPortType">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
- <wsdl:operation name="hasFire">
  <soap12:operation soapAction="urn:hasFire" style="document" />
- <wsdl:input>
  <soap12:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap12:body use="literal" />
  </wsdl:output>
- <wsdl:fault name="FuzzyException">
  <soap12:fault name="FuzzyException" use="literal" />
  </wsdl:fault>
  </wsdl:operation>
- <wsdl:operation name="hasBoilovel">
```

```
<soap12:operation soapAction="urn:hasBoilovel" style="document" />
- <wsdl:input>
<soap12:body use="literal" />
</wsdl:input>
- <wsdl:output>
<soap12:body use="literal" />
</wsdl:output>
- <wsdl:fault name="FuzzyException">
<soap12:fault name="FuzzyException" use="literal" />
</wsdl:fault>
</wsdl:operation>
</wsdl:binding>
- <wsdl:binding name="inferenceFuzzySoap11Binding" type="xsd:inferenceFuzzyPortType">
<soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
- <wsdl:operation name="hasFire">
<soap:operation soapAction="urn:hasFire" style="document" />
- <wsdl:input>
<soap:body use="literal" />
</wsdl:input>
- <wsdl:output>
<soap:body use="literal" />
</wsdl:output>
- <wsdl:fault name="FuzzyException">
<soap:fault name="FuzzyException" use="literal" />
</wsdl:fault>
</wsdl:operation>
- <wsdl:operation name="hasBoilovel">
<soap:operation soapAction="urn:hasBoilovel" style="document" />
- <wsdl:input>
<soap:body use="literal" />
</wsdl:input>
- <wsdl:output>
<soap:body use="literal" />
</wsdl:output>
- <wsdl:fault name="FuzzyException">
<soap:fault name="FuzzyException" use="literal" />
</wsdl:fault>
</wsdl:operation>
</wsdl:binding>
- <wsdl:service name="inferenceFuzzy">
- <wsdl:port name="inferenceFuzzyHttpSoap12Endpoint" binding="xsd:inferenceFuzzySoap12Binding">
<soap12:address location="http://10.222.1.11:8080/axis2/services/inferenceFuzzy/" />
</wsdl:port>
- <wsdl:port name="inferenceFuzzyHttpEndpoint" binding="xsd:inferenceFuzzyHttpBinding">
<http:address location="http://10.222.1.11:8080/axis2/services/inferenceFuzzy/" />
</wsdl:port>
- <wsdl:port name="inferenceFuzzyHttpSoap11Endpoint" binding="xsd:inferenceFuzzySoap11Binding">
```

```
<soap:address location="http://10.222.1.11:8080/axis2/services/inferenceFuzzy/" />
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

APÊNDICE B – Documento WSDL

queryOntology

```
<?xml version="1.0" encoding="UTF-8" ?>
- <wsdl:definitions targetNamespace="http://ontology.controladorSIC" xmlns:ns1="http://org.apache.axis2/xsd"
xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" xmlns:xsd="http://ontology.controladorSIC"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/" xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/">
- <wsdl:types>
- <xs:schema attributeFormDefault="qualified" elementFormDefault="qualified" targetNamespace="http://ontology.controladorSIC">
- <xs:element name="materialIsFuel">
- <xs:complexType>
- <xs:sequence>
 <xs:element minOccurs="0" name="urlEmerg" nillable="true" type="xs:string" />
 <xs:element minOccurs="0" name="material" nillable="true" type="xs:string" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
- <xs:element name="materialIsFuelResponse">
- <xs:complexType>
- <xs:sequence>
 <xs:element minOccurs="0" name="return" type="xs:boolean" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
- <xs:element name="materialIsGasFuel">
- <xs:complexType>
- <xs:sequence>
 <xs:element minOccurs="0" name="urlEmerg" nillable="true" type="xs:string" />
 <xs:element minOccurs="0" name="material" nillable="true" type="xs:string" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
- <xs:element name="materialIsGasFuelResponse">
- <xs:complexType>
- <xs:sequence>
 <xs:element minOccurs="0" name="return" type="xs:boolean" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
- <xs:element name="materialIsLiquidFuel">
```

```
- <xs:complexType>
- <xs:sequence>
  <xs:element minOccurs="0" name="urlEmerg" nillable="true" type="xs:string" />
  <xs:element minOccurs="0" name="material" nillable="true" type="xs:string" />
  </xs:sequence>
  </xs:complexType>
  </xs:element>
- <xs:element name="materialIsLiquidFuelResponse">
- <xs:complexType>
- <xs:sequence>
  <xs:element minOccurs="0" name="return" type="xs:boolean" />
  </xs:sequence>
  </xs:complexType>
  </xs:element>
- <xs:element name="materialIsMiscibleWater">
- <xs:complexType>
- <xs:sequence>
  <xs:element minOccurs="0" name="urlEmerg" nillable="true" type="xs:string" />
  <xs:element minOccurs="0" name="materialTank" nillable="true" type="xs:string" />
  </xs:sequence>
  </xs:complexType>
  </xs:element>
- <xs:element name="materialIsMiscibleWaterResponse">
- <xs:complexType>
- <xs:sequence>
  <xs:element minOccurs="0" name="return" type="xs:boolean" />
  </xs:sequence>
  </xs:complexType>
  </xs:element>
- <xs:element name="materialIsSolidFuel">
- <xs:complexType>
- <xs:sequence>
  <xs:element minOccurs="0" name="urlEmerg" nillable="true" type="xs:string" />
  <xs:element minOccurs="0" name="material" nillable="true" type="xs:string" />
  </xs:sequence>
  </xs:complexType>
  </xs:element>
- <xs:element name="materialIsSolidFuelResponse">
- <xs:complexType>
- <xs:sequence>
  <xs:element minOccurs="0" name="return" type="xs:boolean" />
  </xs:sequence>
  </xs:complexType>
  </xs:element>
- <xs:element name="whatFlashPoint">
- <xs:complexType>
- <xs:sequence>
```

```
<xs:element minOccurs="0" name="urlEmerg" nillable="true" type="xs:string" />
<xs:element minOccurs="0" name="fuel" nillable="true" type="xs:string" />
</xs:sequence>
</xs:complexType>
</xs:element>
- <xs:element name="whatFlashPointResponse">
- <xs:complexType>
- <xs:sequence>
<xs:element minOccurs="0" name="return" type="xs:float" />
</xs:sequence>
</xs:complexType>
</xs:element>
- <xs:element name="whatIgnitionPoint">
- <xs:complexType>
- <xs:sequence>
<xs:element minOccurs="0" name="urlEmerg" nillable="true" type="xs:string" />
<xs:element minOccurs="0" name="fuel" nillable="true" type="xs:string" />
</xs:sequence>
</xs:complexType>
</xs:element>
- <xs:element name="whatIgnitionPointResponse">
- <xs:complexType>
- <xs:sequence>
<xs:element minOccurs="0" name="return" type="xs:float" />
</xs:sequence>
</xs:complexType>
</xs:element>
- <xs:element name="whatMaterial">
- <xs:complexType>
- <xs:sequence>
<xs:element minOccurs="0" name="urlInfra" nillable="true" type="xs:string" />
<xs:element minOccurs="0" name="obj" nillable="true" type="xs:string" />
</xs:sequence>
</xs:complexType>
</xs:element>
- <xs:element name="whatMaterialResponse">
- <xs:complexType>
- <xs:sequence>
<xs:element minOccurs="0" name="return" nillable="true" type="xs:string" />
</xs:sequence>
</xs:complexType>
</xs:element>
- <xs:element name="whatRiskEquipment">
- <xs:complexType>
- <xs:sequence>
<xs:element minOccurs="0" name="urlInfra" nillable="true" type="xs:string" />
<xs:element minOccurs="0" name="Place" nillable="true" type="xs:string" />
```

```
 </xs:sequence>
 </xs:complexType>
 </xs:element>
- <xs:element name="whatRiskEquipmentResponse">
- <xs:complexType>
- <xs:sequence>
 <xs:element minOccurs="0" name="return" nillable="true" type="xs:string" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
- <xs:element name="whatSurroundingRisk">
- <xs:complexType>
- <xs:sequence>
 <xs:element minOccurs="0" name="urlInfra" nillable="true" type="xs:string" />
 <xs:element minOccurs="0" name="place" nillable="true" type="xs:string" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
- <xs:element name="whatSurroundingRiskResponse">
- <xs:complexType>
- <xs:sequence>
 <xs:element minOccurs="0" name="return" nillable="true" type="xs:string" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
 </xs:schema>
 </wsdl:types>
- <wsdl:message name="whatSurroundingRiskResponse">
 <wsdl:part name="parameters" element="xsd:whatSurroundingRiskResponse" />
 </wsdl:message>
- <wsdl:message name="materialIsLiquidFuelRequest">
 <wsdl:part name="parameters" element="xsd:materialIsLiquidFuel" />
 </wsdl:message>
- <wsdl:message name="whatMaterialResponse">
 <wsdl:part name="parameters" element="xsd:whatMaterialResponse" />
 </wsdl:message>
- <wsdl:message name="materialIsMiscibleWaterResponse">
 <wsdl:part name="parameters" element="xsd:materialIsMiscibleWaterResponse" />
 </wsdl:message>
- <wsdl:message name="materialIsFuelRequest">
 <wsdl:part name="parameters" element="xsd:materialIsFuel" />
 </wsdl:message>
- <wsdl:message name="whatIgnitionPointResponse">
 <wsdl:part name="parameters" element="xsd:whatIgnitionPointResponse" />
 </wsdl:message>
- <wsdl:message name="whatIgnitionPointRequest">
 <wsdl:part name="parameters" element="xsd:whatIgnitionPoint" />
```

```
  </wsdl:message>
- <wsdl:message name="whatFlashPointResponse">
  <wsdl:part name="parameters" element="xsd:whatFlashPointResponse" />
  </wsdl:message>
- <wsdl:message name="whatMaterialRequest">
  <wsdl:part name="parameters" element="xsd:whatMaterial" />
  </wsdl:message>
- <wsdl:message name="whatFlashPointRequest">
  <wsdl:part name="parameters" element="xsd:whatFlashPoint" />
  </wsdl:message>
- <wsdl:message name="materialIsMiscibleWaterRequest">
  <wsdl:part name="parameters" element="xsd:materialIsMiscibleWater" />
  </wsdl:message>
- <wsdl:message name="whatRiskEquipmentResponse">
  <wsdl:part name="parameters" element="xsd:whatRiskEquipmentResponse" />
  </wsdl:message>
- <wsdl:message name="whatSurroundingRiskRequest">
  <wsdl:part name="parameters" element="xsd:whatSurroundingRisk" />
  </wsdl:message>
- <wsdl:message name="materialIsSolidFuelRequest">
  <wsdl:part name="parameters" element="xsd:materialIsSolidFuel" />
  </wsdl:message>
- <wsdl:message name="materialIsGasFuelRequest">
  <wsdl:part name="parameters" element="xsd:materialIsGasFuel" />
  </wsdl:message>
- <wsdl:message name="materialIsFuelResponse">
  <wsdl:part name="parameters" element="xsd:materialIsFuelResponse" />
  </wsdl:message>
- <wsdl:message name="materialIsGasFuelResponse">
  <wsdl:part name="parameters" element="xsd:materialIsGasFuelResponse" />
  </wsdl:message>
- <wsdl:message name="materialIsSolidFuelResponse">
  <wsdl:part name="parameters" element="xsd:materialIsSolidFuelResponse" />
  </wsdl:message>
- <wsdl:message name="materialIsLiquidFuelResponse">
  <wsdl:part name="parameters" element="xsd:materialIsLiquidFuelResponse" />
  </wsdl:message>
- <wsdl:message name="whatRiskEquipmentRequest">
  <wsdl:part name="parameters" element="xsd:whatRiskEquipment" />
  </wsdl:message>
- <wsdl:portType name="queryOntologyPortType">
- <wsdl:operation name="whatFlashPoint">
  <wsdl:input message="xsd:whatFlashPointRequest" wsaw:Action="urn:whatFlashPoint" />
  <wsdl:output message="xsd:whatFlashPointResponse" wsaw:Action="urn:whatFlashPointResponse" />
  </wsdl:operation>
- <wsdl:operation name="whatRiskEquipment">
  <wsdl:input message="xsd:whatRiskEquipmentRequest" wsaw:Action="urn:whatRiskEquipment" />
```

```
<wsdl:output message="xsd:whatRiskEquipmentResponse" wsaw:Action="urn:whatRiskEquipmentResponse" />
</wsdl:operation>
- <wsdl:operation name="materialIsGasFuel">
<wsdl:input message="xsd:materialIsGasFuelRequest" wsaw:Action="urn:materialIsGasFuel" />
<wsdl:output message="xsd:materialIsGasFuelResponse" wsaw:Action="urn:materialIsGasFuelResponse" />
</wsdl:operation>
- <wsdl:operation name="whatSurroundingRisk">
<wsdl:input message="xsd:whatSurroundingRiskRequest" wsaw:Action="urn:whatSurroundingRisk" />
<wsdl:output message="xsd:whatSurroundingRiskResponse" wsaw:Action="urn:whatSurroundingRiskResponse" />
</wsdl:operation>
- <wsdl:operation name="whatMaterial">
<wsdl:input message="xsd:whatMaterialRequest" wsaw:Action="urn:whatMaterial" />
<wsdl:output message="xsd:whatMaterialResponse" wsaw:Action="urn:whatMaterialResponse" />
</wsdl:operation>
- <wsdl:operation name="materialIsLiquidFuel">
<wsdl:input message="xsd:materialIsLiquidFuelRequest" wsaw:Action="urn:materialIsLiquidFuel" />
<wsdl:output message="xsd:materialIsLiquidFuelResponse" wsaw:Action="urn:materialIsLiquidFuelResponse" />
</wsdl:operation>
- <wsdl:operation name="materialIsSolidFuel">
<wsdl:input message="xsd:materialIsSolidFuelRequest" wsaw:Action="urn:materialIsSolidFuel" />
<wsdl:output message="xsd:materialIsSolidFuelResponse" wsaw:Action="urn:materialIsSolidFuelResponse" />
</wsdl:operation>
- <wsdl:operation name="whatIgnitionPoint">
<wsdl:input message="xsd:whatIgnitionPointRequest" wsaw:Action="urn:whatIgnitionPoint" />
<wsdl:output message="xsd:whatIgnitionPointResponse" wsaw:Action="urn:whatIgnitionPointResponse" />
</wsdl:operation>
- <wsdl:operation name="materialIsMiscibleWater">
<wsdl:input message="xsd:materialIsMiscibleWaterRequest" wsaw:Action="urn:materialIsMiscibleWater" />
<wsdl:output message="xsd:materialIsMiscibleWaterResponse" wsaw:Action="urn:materialIsMiscibleWaterResponse" />
</wsdl:operation>
- <wsdl:operation name="materialIsFuel">
<wsdl:input message="xsd:materialIsFuelRequest" wsaw:Action="urn:materialIsFuel" />
<wsdl:output message="xsd:materialIsFuelResponse" wsaw:Action="urn:materialIsFuelResponse" />
</wsdl:operation>
</wsdl:portType>
- <wsdl:binding name="queryOntologySoap11Binding" type="xsd:queryOntologyPortType">
<soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
- <wsdl:operation name="whatFlashPoint">
<soap:operation soapAction="urn:whatFlashPoint" style="document" />
- <wsdl:input>
<soap:body use="literal" />
</wsdl:input>
- <wsdl:output>
<soap:body use="literal" />
</wsdl:output>
</wsdl:operation>
- <wsdl:operation name="whatRiskEquipment">
```

```
<soap:operation soapAction="urn:whatRiskEquipment" style="document" />
- <wsdl:input>
<soap:body use="literal" />
</wsdl:input>
- <wsdl:output>
<soap:body use="literal" />
</wsdl:output>
</wsdl:operation>
- <wsdl:operation name="materialIsGasFuel">
<soap:operation soapAction="urn:materialIsGasFuel" style="document" />
- <wsdl:input>
<soap:body use="literal" />
</wsdl:input>
- <wsdl:output>
<soap:body use="literal" />
</wsdl:output>
</wsdl:operation>
- <wsdl:operation name="whatSurroundingRisk">
<soap:operation soapAction="urn:whatSurroundingRisk" style="document" />
- <wsdl:input>
<soap:body use="literal" />
</wsdl:input>
- <wsdl:output>
<soap:body use="literal" />
</wsdl:output>
</wsdl:operation>
- <wsdl:operation name="whatMaterial">
<soap:operation soapAction="urn:whatMaterial" style="document" />
- <wsdl:input>
<soap:body use="literal" />
</wsdl:input>
- <wsdl:output>
<soap:body use="literal" />
</wsdl:output>
</wsdl:operation>
- <wsdl:operation name="materialIsLiquidFuel">
<soap:operation soapAction="urn:materialIsLiquidFuel" style="document" />
- <wsdl:input>
<soap:body use="literal" />
</wsdl:input>
- <wsdl:output>
<soap:body use="literal" />
</wsdl:output>
</wsdl:operation>
- <wsdl:operation name="materialIsSolidFuel">
<soap:operation soapAction="urn:materialIsSolidFuel" style="document" />
- <wsdl:input>
```

```
  <soap:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="materialIsMiscibleWater">
  <soap:operation soapAction="urn:materialIsMiscibleWater" style="document" />
- <wsdl:input>
  <soap:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="whatIgnitionPoint">
  <soap:operation soapAction="urn:whatIgnitionPoint" style="document" />
- <wsdl:input>
  <soap:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="materialIsFuel">
  <soap:operation soapAction="urn:materialIsFuel" style="document" />
- <wsdl:input>
  <soap:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
  </wsdl:binding>
- <wsdl:binding name="queryOntologyHttpBinding" type="xsd:queryOntologyPortType">
  <http:binding verb="POST" />
- <wsdl:operation name="whatFlashPoint">
  <http:operation location="queryOntology/whatFlashPoint" />
- <wsdl:input>
  <mime:content part="whatFlashPoint" type="text/xml" />
  </wsdl:input>
- <wsdl:output>
  <mime:content part="whatFlashPoint" type="text/xml" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="whatRiskEquipment">
  <http:operation location="queryOntology/whatRiskEquipment" />
```

```
- <wsdl:input>
  <mime:content part="whatRiskEquipment" type="text/xml" />
  </wsdl:input>
- <wsdl:output>
  <mime:content part="whatRiskEquipment" type="text/xml" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="materialIsGasFuel">
  <http:operation location="queryOntology/materialIsGasFuel" />
- <wsdl:input>
  <mime:content part="materialIsGasFuel" type="text/xml" />
  </wsdl:input>
- <wsdl:output>
  <mime:content part="materialIsGasFuel" type="text/xml" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="whatSurroundingRisk">
  <http:operation location="queryOntology/whatSurroundingRisk" />
- <wsdl:input>
  <mime:content part="whatSurroundingRisk" type="text/xml" />
  </wsdl:input>
- <wsdl:output>
  <mime:content part="whatSurroundingRisk" type="text/xml" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="whatMaterial">
  <http:operation location="queryOntology/whatMaterial" />
- <wsdl:input>
  <mime:content part="whatMaterial" type="text/xml" />
  </wsdl:input>
- <wsdl:output>
  <mime:content part="whatMaterial" type="text/xml" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="materialIsLiquidFuel">
  <http:operation location="queryOntology/materialIsLiquidFuel" />
- <wsdl:input>
  <mime:content part="materialIsLiquidFuel" type="text/xml" />
  </wsdl:input>
- <wsdl:output>
  <mime:content part="materialIsLiquidFuel" type="text/xml" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="materialIsSolidFuel">
  <http:operation location="queryOntology/materialIsSolidFuel" />
- <wsdl:input>
  <mime:content part="materialIsSolidFuel" type="text/xml" />
```

```
  </wsdl:input>
- <wsdl:output>
  <mime:content part="materialIsSolidFuel" type="text/xml" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="materialIsMiscibleWater">
  <http:operation location="queryOntology/materialIsMiscibleWater" />
- <wsdl:input>
  <mime:content part="materialIsMiscibleWater" type="text/xml" />
  </wsdl:input>
- <wsdl:output>
  <mime:content part="materialIsMiscibleWater" type="text/xml" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="whatIgnitionPoint">
  <http:operation location="queryOntology/whatIgnitionPoint" />
- <wsdl:input>
  <mime:content part="whatIgnitionPoint" type="text/xml" />
  </wsdl:input>
- <wsdl:output>
  <mime:content part="whatIgnitionPoint" type="text/xml" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="materialIsFuel">
  <http:operation location="queryOntology/materialIsFuel" />
- <wsdl:input>
  <mime:content part="materialIsFuel" type="text/xml" />
  </wsdl:input>
- <wsdl:output>
  <mime:content part="materialIsFuel" type="text/xml" />
  </wsdl:output>
  </wsdl:operation>
  </wsdl:binding>
- <wsdl:binding name="queryOntologySoap12Binding" type="xsd:queryOntologyPortType">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
- <wsdl:operation name="whatFlashPoint">
  <soap12:operation soapAction="urn:whatFlashPoint" style="document" />
- <wsdl:input>
  <soap12:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap12:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="whatRiskEquipment">
  <soap12:operation soapAction="urn:whatRiskEquipment" style="document" />
- <wsdl:input>
```

```
  <soap12:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap12:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="materialIsGasFuel">
  <soap12:operation soapAction="urn:materialIsGasFuel" style="document" />
- <wsdl:input>
  <soap12:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap12:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="whatSurroundingRisk">
  <soap12:operation soapAction="urn:whatSurroundingRisk" style="document" />
- <wsdl:input>
  <soap12:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap12:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="whatMaterial">
  <soap12:operation soapAction="urn:whatMaterial" style="document" />
- <wsdl:input>
  <soap12:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap12:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="materialIsLiquidFuel">
  <soap12:operation soapAction="urn:materialIsLiquidFuel" style="document" />
- <wsdl:input>
  <soap12:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap12:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="materialIsSolidFuel">
  <soap12:operation soapAction="urn:materialIsSolidFuel" style="document" />
- <wsdl:input>
  <soap12:body use="literal" />
  </wsdl:input>
```

```
- <wsdl:output>
  <soap12:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="materialIsMiscibleWater">
  <soap12:operation soapAction="urn:materialIsMiscibleWater" style="document" />
- <wsdl:input>
  <soap12:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap12:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="whatIgnitionPoint">
  <soap12:operation soapAction="urn:whatIgnitionPoint" style="document" />
- <wsdl:input>
  <soap12:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap12:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="materialIsFuel">
  <soap12:operation soapAction="urn:materialIsFuel" style="document" />
- <wsdl:input>
  <soap12:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap12:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
  </wsdl:binding>
- <wsdl:service name="queryOntology">
- <wsdl:port name="queryOntologyHttpSoap11Endpoint" binding="xsd:queryOntologySoap11Binding">
  <soap:address location="http://10.222.1.11:8080/axis2/services/queryOntology/" />
  </wsdl:port>
- <wsdl:port name="queryOntologyHttpEndpoint" binding="xsd:queryOntologyHttpBinding">
  <http:address location="http://10.222.1.11:8080/axis2/services/queryOntology/" />
  </wsdl:port>
- <wsdl:port name="queryOntologyHttpSoap12Endpoint" binding="xsd:queryOntologySoap12Binding">
  <soap12:address location="http://10.222.1.11:8080/axis2/services/queryOntology/" />
  </wsdl:port>
  </wsdl:service>
  </wsdl:definitions>
```

APÊNDICE C – Documento WSDL

managerContext

```
<?xml version="1.0" encoding="UTF-8" ?>
- <wsdl:definitions targetNamespace="http://mContext" xmlns:ns1="http://org.apache.axis2/xsd"
xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" xmlns:xsd="http://mContext"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/" xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/">
- <wsdl:types>
- <xs:schema attributeFormDefault="qualified" elementFormDefault="qualified" targetNamespace="http://mContext">
- <xs:element name="getContext">
- <xs:complexType>
- <xs:sequence>
 <xs:element minOccurs="0" name="param" nillable="true" type="xs:string" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
- <xs:element name="getContextResponse">
- <xs:complexType>
- <xs:sequence>
 <xs:element minOccurs="0" name="return" nillable="true" type="xs:string" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
 </xs:schema>
 </wsdl:types>
- <wsdl:message name="getContextResponse">
 <wsdl:part name="parameters" element="xsd:getContextResponse" />
 </wsdl:message>
- <wsdl:message name="getContextRequest">
 <wsdl:part name="parameters" element="xsd:getContext" />
 </wsdl:message>
- <wsdl:portType name="managerContextPortType">
- <wsdl:operation name="getContext">
 <wsdl:input message="xsd:getContextRequest" wsaw:Action="urn:getContext" />
 <wsdl:output message="xsd:getContextResponse" wsaw:Action="urn:getContextResponse" />
 </wsdl:operation>
 </wsdl:portType>
- <wsdl:binding name="managerContextSoap11Binding" type="xsd:managerContextPortType">
 <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
- <wsdl:operation name="getContext">
```

```
  <soap:operation soapAction="urn:getContext" style="document" />
- <wsdl:input>
  <soap:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
  </wsdl:binding>
- <wsdl:binding name="managerContextHttpBinding" type="xsd:managerContextPortType">
  <http:binding verb="POST" />
- <wsdl:operation name="getContext">
  <http:operation location="managerContext/getContext" />
- <wsdl:input>
  <mime:content part="getContext" type="text/xml" />
  </wsdl:input>
- <wsdl:output>
  <mime:content part="getContext" type="text/xml" />
  </wsdl:output>
  </wsdl:operation>
  </wsdl:binding>
- <wsdl:binding name="managerContextSoap12Binding" type="xsd:managerContextPortType">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
- <wsdl:operation name="getContext">
  <soap12:operation soapAction="urn:getContext" style="document" />
- <wsdl:input>
  <soap12:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap12:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
  </wsdl:binding>
- <wsdl:service name="managerContext">
- <wsdl:port name="managerContextHttpSoap12Endpoint" binding="xsd:managerContextSoap12Binding">
  <soap12:address location="http://10.222.1.11:8080/axis2/services/managerContext/" />
  </wsdl:port>
- <wsdl:port name="managerContextHttpEndpoint" binding="xsd:managerContextHttpBinding">
  <http:address location="http://10.222.1.11:8080/axis2/services/managerContext/" />
  </wsdl:port>
- <wsdl:port name="managerContextHttpSoap11Endpoint" binding="xsd:managerContextSoap11Binding">
  <soap:address location="http://10.222.1.11:8080/axis2/services/managerContext/" />
  </wsdl:port>
  </wsdl:service>
  </wsdl:definitions>
```

APÊNDICE D – Documento WSDL

managerAgent

```
<?xml version="1.0" encoding="UTF-8" ?>
- <wsdl:definitions targetNamespace="http://mAgent" xmlns:ns1="http://org.apache.axis2/xsd"
xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" xmlns:xsd="http://mAgent" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/">
- <wsdl:types>
- <xs:schema attributeFormDefault="qualified" elementFormDefault="qualified" targetNamespace="http://mAgent">
- <xs:element name="getAgent">
- <xs:complexType>
- <xs:sequence>
 <xs:element minOccurs="0" name="param1" nillable="true" type="xs:string" />
 <xs:element minOccurs="0" name="param2" nillable="true" type="xs:string" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
- <xs:element name="getAgentResponse">
- <xs:complexType>
- <xs:sequence>
 <xs:element minOccurs="0" name="return" nillable="true" type="xs:string" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
- <xs:element name="getAgentbf">
- <xs:complexType>
- <xs:sequence>
 <xs:element minOccurs="0" name="param1" nillable="true" type="xs:string" />
 <xs:element minOccurs="0" name="param2" nillable="true" type="xs:string" />
 <xs:element minOccurs="0" name="param3" nillable="true" type="xs:string" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
- <xs:element name="getAgentbfResponse">
- <xs:complexType>
- <xs:sequence>
 <xs:element minOccurs="0" name="return" nillable="true" type="xs:string" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
 </xs:schema>
```

```
  </wsdl:types>
- <wsdl:message name="getAgentbfRequest">
  <wsdl:part name="parameters" element="xsd:getAgentbf" />
  </wsdl:message>
- <wsdl:message name="getAgentbfResponse">
  <wsdl:part name="parameters" element="xsd:getAgentbfResponse" />
  </wsdl:message>
- <wsdl:message name="getAgentRequest">
  <wsdl:part name="parameters" element="xsd:getAgent" />
  </wsdl:message>
- <wsdl:message name="getAgentResponse">
  <wsdl:part name="parameters" element="xsd:getAgentResponse" />
  </wsdl:message>
- <wsdl:portType name="managerAgentPortType">
- <wsdl:operation name="getAgentbf">
  <wsdl:input message="xsd:getAgentbfRequest" wsaw:Action="urn:getAgentbf" />
  <wsdl:output message="xsd:getAgentbfResponse" wsaw:Action="urn:getAgentbfResponse" />
  </wsdl:operation>
- <wsdl:operation name="getAgent">
  <wsdl:input message="xsd:getAgentRequest" wsaw:Action="urn:getAgent" />
  <wsdl:output message="xsd:getAgentResponse" wsaw:Action="urn:getAgentResponse" />
  </wsdl:operation>
  </wsdl:portType>
- <wsdl:binding name="managerAgentHttpBinding" type="xsd:managerAgentPortType">
  <http:binding verb="POST" />
- <wsdl:operation name="getAgentbf">
  <http:operation location="managerAgent/getAgentbf" />
- <wsdl:input>
  <mime:content part="getAgentbf" type="text/xml" />
  </wsdl:input>
- <wsdl:output>
  <mime:content part="getAgentbf" type="text/xml" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="getAgent">
  <http:operation location="managerAgent/getAgent" />
- <wsdl:input>
  <mime:content part="getAgent" type="text/xml" />
  </wsdl:input>
- <wsdl:output>
  <mime:content part="getAgent" type="text/xml" />
  </wsdl:output>
  </wsdl:operation>
  </wsdl:binding>
- <wsdl:binding name="managerAgentSoap11Binding" type="xsd:managerAgentPortType">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
- <wsdl:operation name="getAgentbf">
```

```
  <soap:operation soapAction="urn:getAgentbf" style="document" />
- <wsdl:input>
  <soap:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="getAgent">
  <soap:operation soapAction="urn:getAgent" style="document" />
- <wsdl:input>
  <soap:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
  </wsdl:binding>
- <wsdl:binding name="managerAgentSoap12Binding" type="xsd:managerAgentPortType">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
- <wsdl:operation name="getAgentbf">
  <soap12:operation soapAction="urn:getAgentbf" style="document" />
- <wsdl:input>
  <soap12:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap12:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="getAgent">
  <soap12:operation soapAction="urn:getAgent" style="document" />
- <wsdl:input>
  <soap12:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap12:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
  </wsdl:binding>
- <wsdl:service name="managerAgent">
- <wsdl:port name="managerAgentHttpEndpoint" binding="xsd:managerAgentHttpBinding">
  <http:address location="http://10.222.1.11:8080/axis2/services/managerAgent/" />
  </wsdl:port>
- <wsdl:port name="managerAgentHttpSoap12Endpoint" binding="xsd:managerAgentSoap12Binding">
  <soap12:address location="http://10.222.1.11:8080/axis2/services/managerAgent/" />
  </wsdl:port>
- <wsdl:port name="managerAgentHttpSoap11Endpoint" binding="xsd:managerAgentSoap11Binding">
```

```
<soap:address location="http://10.222.1.11:8080/axis2/services/managerAgent/" />
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
```