UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

MARCOS PAULINO RORIZ JUNIOR

Middleware de Objetos Distribuídos para Rede de Sensores Sem Fio

Goiânia
2010

UNIVERSIDADE FEDERAL DE GOIÁS

INSTITUTO DE INFORMÁTICA

AUTORIZAÇÃO PARA PUBLICAÇÃO DE TRABALHO DE CONCLUSÃO DE CURSO EM FORMATO ELETRÔNICO

Na qualidade de titular dos direitos de autor, **AUTORIZO** o Instituto de Informática da Universidade Federal de Goiás – UFG a reproduzir, inclusive em outro formato ou mídia e através de armazenamento permanente ou temporário, bem como a publicar na rede mundial de computadores (*Internet*) e na biblioteca virtual da UFG, entendendo-se os termos "reproduzir" e "publicar" conforme definições dos incisos VI e I, respectivamente, do artigo 5º da Lei nº 9610/98 de 10/02/1998, a obra abaixo especificada, sem que me seja devido pagamento a título de direitos autorais, desde que a reprodução e/ou publicação tenham a finalidade exclusiva de uso por quem a consulta, e a título de divulgação da produção acadêmica gerada pela Universidade, a partir desta data.

Título: Middleware de Objetos Distribuídos para Rede de Sensores Sem Fio

Autor(a): Marcos Paulino Roriz Junior

Goiânia, 02 de Dezembro de 2010.

Marcos Paulino Roriz Junior – Autor

Fábio Moreira Costa – Orientador

MARCOS PAULINO RORIZ JUNIOR

Middleware de Objetos Distribuídos para Rede de Sensores Sem Fio

Trabalho de Conclusão apresentado à Coordenação do Curso de Ciências da Computação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Bacharel em Ciências da Computação.

Área de concentração: Sistemas Distribuídos.

Orientador: Prof. Fábio Moreira Costa

Goiânia
2010

MARCOS PAULINO RORIZ JUNIOR

Middleware de Objetos Distribuídos para Rede de Sensores Sem Fio

Trabalho de Conclusão apresentado à Coordenação do Curso de Ciências da Computação do Instituto de Informática da Universidade Federal de Goiás como requisito parcial para obtenção do título de Bacharel em Ciências da Computação, aprovada em 02 de Dezembro de 2010, pela Banca Examinadora constituída pelos professores:

Prof. Fábio Moreira Costa
Instituto de Informática – UFG
Presidente da Banca

Prof. Vagner José do Sacramento Rodrigues
Instituto de Informática – UFG

Marcos Paulino Roriz Junior

Graduando em Ciências da Computação pela Universidade Federal de Goiás (UFG) e bolsista do Grupo de Pesquisa Aplicada à Internet e Sistemas Distribuídos (GEApIS), atuando como desenvolvedor na área de sistemas distribuídos. Trabalha com diversas soluções desta área e também com a estrutura interna da plataforma Java em diversas máquinas virtuais (Harmony, Squawk e phoneME). Possui conhecimento avançado em sistemas Linux e proficiência com a linguagem de programação Java. Foi bolsista do projeto GingaCDN do módulo de Inter Comunicação entre Aplicações do *middleware*. Participou do Google Summer of Code 2010 pelo projeto GNU Classpath, uma implementação livre do conjunto de classes padrão da plataforma Java.

Dedico este trabalho a minha família: Vó Rosalha Maria Borges, Mãe Luciane Franco Borges, Tia Cristiane Franco Borges e Irmão Diogo Borges Nery

Agradecimentos

Antes de tudo gostaria de agradecer minha família: Vó Rosalha Maria Borges, Mãe Luciane Franco Borges, Tia Cristiane Franco Borges e Irmão Diogo Borges Nery que sempre estiveram me apoiando na vida, principalmente nos momentos difíceis como nos que tive de ausentar da presença deles para realiza este trabalho. Possuem o meu eterno amor, gratidão e admiração.

Ao meu orientador Dr. Fábio Moreira Costa, ao qual tenho uma imensa gratidão e admiração, por sua excelente orientação, disponibilidade e por ter acreditado em mim e ter me dado a honra de trabalhar ao seu lado. Também gostaria de agradecer o meu avaliador Dr. Vagner José do Sacramento Rodrigues, por sua imensa paciência com minhas dúvidas. Tenho orgulho de ser aluno dos senhores.

Aos amigos Guilherme Kenedy Ferreira, Gustavo Santos Nunes, João Guilherme Araújo, Marco Aurélio Lino Massarani e Renner Ricardo Leão pelo imenso apoio durante o curso e pelos momentos divertidos que passamos juntos! Nunca esquecerei vocês!

"The Network is the Computer."

John Gage,
Sun Microsystems.

Resumo

Roriz Junior, Marcos Paulino. **Middleware de Objetos Distribuídos para Rede de Sensores Sem Fio**. Goiânia, 2010. 52p. Relatório de Graduação. Instituto de Informática, Universidade Federal de Goiás.

Este projeto propõe o desenvolvimento de uma arquitetura de *Middleware* de Objetos Distribuídos para Rede de Sensores Sem Fio. O *middleware* irá abstrair os detalhes de comunicação entre sensores, permitindo o desenvolvimento de aplicações com um maior nível de abstração. Uma implementação deste foi desenvolvida para a plataforma SunSPOT com objetivo de demonstrar e validar a arquitetura.

Palavras–chave

Middleware RMI, Rede de Sensores Sem Fio, Sensor, SunSPOT

Abstract

Roriz Junior, Marcos Paulino. **Distributed Object Middleware for Wireless Sensor Network**. Goiânia, 2010. 52p. Relatório de Graduação. Instituto de Informática, Universidade Federal de Goiás.

This project proposes the development of a Distributed Object Middleware architecture for Wireless Sensor Network. The middleware will abstract the details of communication between sensors, enabling the development of applications with a higher level of abstraction. An implementation of this was developed to the SunSPOT platform in order to demonstrate and validate the architecture.

Keywords

RMI Middleware, Wireless Sensor Networks, Sensor, SunSPOT

Sumário

Lista de Figuras **10**

Lista de Tabelas **11**

Lista de Códigos de Programas **12**

1 Introdução **13**
- 1.1 Estrutura do Trabalho 15

2 Conceitos Básicos **16**
- 2.1 Chamada de Métodos Remotos 16
- 2.2 Middleware para Rede de Sensores Sem Fio 17
- 2.3 Sun SPOT 19

3 Arquitetura **22**
- 3.1 Módulo de Referência Remota 23
- 3.2 Servidor de Nomes 25
- 3.3 Módulo de *Bufferização* 30
- 3.4 Módulo de Comunicação 30
- 3.5 Serialização 32

4 Implementação **34**
- 4.1 Estabelecimento de Conexão 34
- 4.2 Operações 36
- 4.3 Empacotamento e Armazenamento de Dados 38
- 4.4 Esqueletos e *Proxies* 38
- 4.5 Geração de Esqueletos e *Proxies* 41
- 4.6 Avaliação 42

5 Trabalhos Relacionados **44**

6 Conclusão **45**

Referências Bibliográficas **46**

A *Opcodes* **49**

B Unidade de dados do protocolo **51**

Lista de Figuras

1.1 Sensores atuando como roteadores em uma RSSF 14

2.1 Exemplo de utilização de RMI [6] 16
2.2 *Hardware* do Sun SPOT 19
2.3 Emulador Solarium 21

3.1 Arquitetura do *middleware* 22
3.2 Diagrama de Sequência para a operação de *bind* 27
3.3 Diagrama de Sequência para a operação de *list* 28
3.4 Diagrama de Sequência para a operação de *lookup* 28
3.5 Diagrama de Sequência para a operação de *rebind* 29
3.6 Diagrama de Sequência para a operação de *unbind* 29
3.7 *Buffer* de Comunicação 31

4.1 Logotipo do spotSHOUT e da fundação Apache 34
4.2 Diagrama de sequência para o estabelecimento de conexão entre nós 35
4.3 Exemplo de escrita da operação *LookupRequest* 36
4.4 Hierarquia de operações 37
4.5 Hierarquia de classes de empacotamento 38
4.6 Interface gráfica para geração de esqueletos e *proxies* 41
4.7 Comparação por tamanho (KiB) 42
4.8 Comparação por uso de memória 42
4.9 Comparação por linhas de código 43

Lista de Tabelas

1.1 Comparação entre Sensores [25][21] 13

3.1 Tabela de registro local que guarda metadados sobre objetos remotos 24
3.2 Cabeçalho comum a todas operações 31
3.3 Alguns códigos de operações 31
3.4 Cabeçalhos comuns a operações de requisição e resposta 32
3.5 Cabeçalho da operação *Lookup Reply* 32

Lista de Códigos de Programas

2.1 `SunSpotApplication` 20
4.1 `Interface Remota Calculator` 39
4.2 `Calculator Proxy` 39
4.3 `Calculator Skel` 40

CAPÍTULO 1

Introdução

Com o avanço da tecnologia de redes sem fio e sistemas embarcados, estamos vivendo cada vez mais a miniaturização dos sistemas computacionais, desde celulares a pequenos sensores. Os sensores têm se destacado entre esses pequenos sistemas pela capacidade de se tornarem ubíquos e de conseguir aproximar o mundo físico do virtual, isto é, transformando valores analógicos (temperatura, luminosidade, etc) em valores digitais [14]. Explorando essa tecnologia foram construídas diversas aplicações desde monitoramento ambiental a rastreamento de produtos. Pela dimensão do ambiente e aplicações, é necessário a utilização de diversos sensores, formando assim uma rede destes. Uma **R**ede de **S**ensores **S**em **F**io (RSSF) consiste em um conjunto de sensores sem fio que realizam sensoriamento distribuído e processamento de dados coletivos [24]. Uma RSSF utiliza protocolos e algoritmos distribuídos para permitir esta interação. As aplicações são construídas em cima da infraestrutura fornecia pela RSSF. A Figura 1.1 mostra uma rede de sensores sem fio aonde cada sensor serve como unidade e roteador para a comunicação de dados.

Por causa do tamanho dos sensores eles possuem pequena capacidade de processamento e fonte de energia limitada. Até recentemente, as aplicações eram altamente acopladas aos protocolos e algoritmos de RSSF [24]. Porém, este cenário está mudando, como podemos ver na Tabela 1.1. Os sensores aumentaram significativamente seu poder de processamento e memória, embora a capacidade de energia ainda seja um problema.

Sensor	MicaZ	Telos	Sun SPOT	iMote 2.0
Ano	2002	2004	2007	2007
CPU	AtMega128L	TI MSP430	ARM920T	Marvell ARM
Memória RAM	4 KB	10 KB	512 KB	32 MB
Memória Flash	128 KB	48 KB	4 MB	32 MB

Tabela 1.1: *Comparação entre Sensores [25][21]*

Mesmo com o aumento significativo da capacidade de processamento e memória dos sensores, o *software* de comunicação desses não seguiu essa evolução. Atualmente, para enviar uma mensagem entre dois sensores, é necessário lidar com várias questões

de baixo nível, como endereço MAC de destino, algoritmo de roteamento e tamanho em *bytes* do pacote a ser enviado. Além disso na maioria dos sensores, só é possível enviar dados primitivos. Isto dificulta enormemente a construção de aplicações um pouco mais elaboradas para sensores, isto é, que fogem do paradigma de apenas coletar dados. Uma solução possível para este problema é a construção de um *middleware*, que nada mais é do que uma camada de abstração de *software* situada entre aplicativos e o ambiente de execução fazendo a mediação entre esses aplicativos [6] [22]. A mediação, é feita através de serviços e é utilizada geralmente por aplicações distribuída, com o objetivo de facilitar o desenvolvimento delas através da abstração dos problemas (heterogeneidade, migração, segurança, etc.) da computação distribuída. Infelizmente a maior parte das soluções de *middleware* para RSSF focam nas restrições físicas de sensores do que requisitos de aplicações [7].

As soluções de *middleware* para RSSF que focam na comunicação entre sensores em geral tratam de criar novas abstrações na utilização do meio de comunicação, como criar uma memória compartilhada distribuída entre os sensores, ao contrário de simplesmente prover transparência de comunicação. Uma categoria de *middleware* interessante é os ORBs (***O**bject **R**equest **B**roker*), que permite facilita a comunicação entre objetos de aplicações um distribuídas. O ORB também trata dos problemas envolvidos na comunicação, como transparência, localização, *marshalling* e sistemas heterogêneos. A

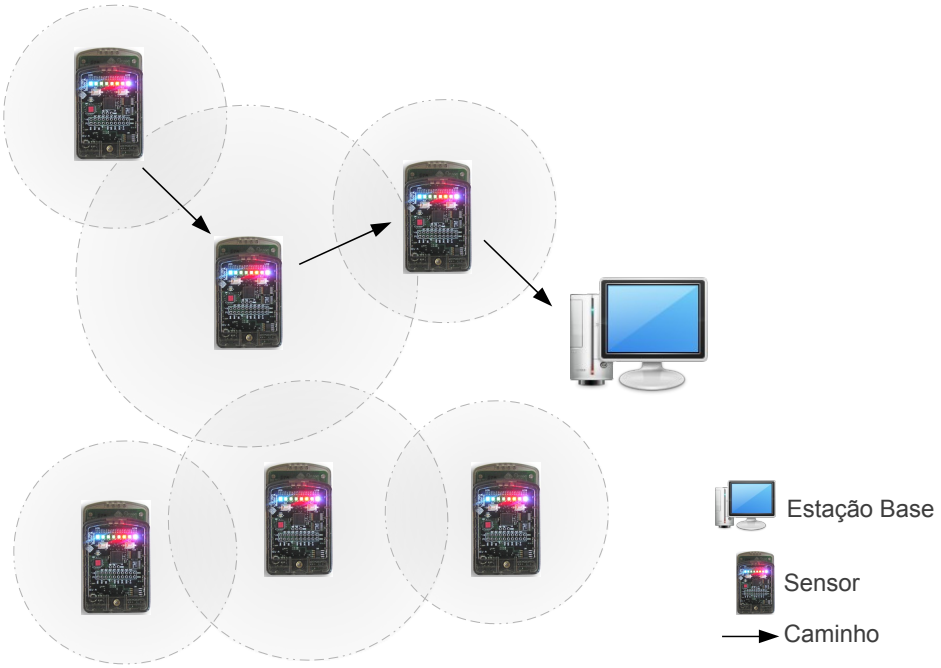

Figura 1.1: *Sensores atuando como roteadores em uma RSSF*

chamada de métodos remotos (RMI – ***R**emote **M**ethod **I**nvocation*) é uma implementação simplificada de um ORB para a linguagem Java. Devido a essas características um *middleware* RMI chegou a ser avaliado como candidato para ser portado para a arquitetura de RSSF em 2004 [19], porém foi 'descartado' por ser pesado, o que é válido se pensarmos que os sistemas possuíam apenas 10 KB RAM ao contrário de 32 MB RAM de hoje. O objetivo deste trabalho é desenvolver uma arquitetura para a construção/adaptação de um *middleware* de RMI para RSSF. Com a implementação deste *middleware* será possível desenvolver aplicações com maior nível de abstração para sensores. Exemplos de aplicações que pode ser beneficiados por esta abstração são descritos conforme a seguir:

- Dois robôs controlados por sensores podem chamar métodos de alto nível entre si para traçar uma estratégia para um objetivo em comum, como por exemplo pegar um objeto em uma dada localização do espaço físico. Trocando dados complexos (objetos) com o intuito de chamar ações mais complexas facilita a abstração desta tarefa.
- Uma RSSF monitora e controla o estado de pacientes em um hospital. Com a abordagem de RMI, um médico poderia requisitar, a qualquer momento, uma visão geral de um paciente retornando assim um dado complexo (objeto paciente).

1.1 Estrutura do Trabalho

O restante do trabalho está organizado como se segue. O capítulo 2 apresenta conceitos gerais para a compreensão de um *middleware* ORB e RSSF; O capítulo 3 apresenta a arquitetura do *middleware*, seus componentes e funcionalidades. O capítulo 4 apresenta os detalhes da implementação destes para a plataforma SunSPOT e no final mostra uma avaliação comparativa desta com outras soluções de *middleware* para o SunSPOT. O capítulo 5 apresenta trabalhos relacionados a este. Finalmente no capítulo 6 é feito a conclusão do trabalho com ideias futuras.

CAPÍTULO 2

Conceitos Básicos

A fundamentação teórica deste trabalho se interlaça entre as diversas áreas da Ciência da Computação, como Redes de Computadores, Sistemas Distribuídos e Algoritmos. Devido a extensão destas áreas iremos focar nas partes específicas pertinentes ao trabalho. Na parte prática do trabalho iremos utilizar a plataforma SunSPOT, que permite a programação de aplicações na linguagem Java. Neste capítulo será introduzido conceitos básicos destas áreas para uma compreensão do trabalho. Será explicado o funcionamento da chamada de métodos remotos na literatura (utilizando como exemplo o Java RMI) e identificado os requisitos para um *middleware* de RSSF. Além disso será feito uma concisa explicação do modelo de programação do SunSPOT.

2.1 Chamada de Métodos Remotos

Chamada de Métodos Remotos (RMI – ***R**emote **M**ethod **I**nvocation*) é uma implementação de um ORB simplificado para Java. Este permite que um objeto em um processo chame os métodos de um objeto em outro processo [6], seja na mesma máquina ou em máquinas separadas. Um *middleware* RMI trata de todos os problemas existentes na localização e comunicação com objetos remotos. A Figura 2.1 mostra um exemplo de comunicação utilizando RMI. Um objeto *A* chama um objeto remoto *B*. Este dispara chamadas locais que no final decai em outra chamada remota, neste caso no objeto *F*.

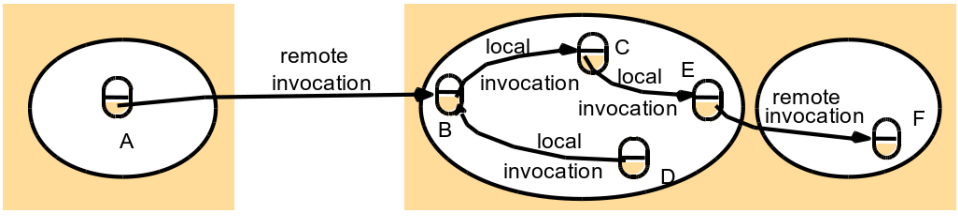

Figura 2.1: *Exemplo de utilização de RMI [6]*

Para permitir esta comunicação entre os objetos é necessário um mecanismo de comunicação, o que denominamos de módulo de comunicação. Este transmite as

mensagens utilizando um protocolo requisição-resposta tradicional. A mensagem de requisição irá empacotar metadados da chamada, como identificação do método chamado e os argumentos utilizado na chamada e a mensagem de resposta conterá metadados de resposta e o valor da resposta, se houver.

Antes de toda essa comunicação deve haver um contrato entres estes objetos para que um possa localizar o outro. O objeto remoto alvo anunciará sua presença ao registro de objetos da rede utilizando o módulo de referencia remota. O módulo de referência remota é responsável por criar uma referência remota, "endereço", a partir de um objeto remoto. Além disso, ele também é responsável por salvar esta associação, geralmente em uma tabela, para depois poder traduzir as chamadas de requisição e resposta, esta função é denominada vinculação (*binding*). Este módulo está presente tanto no cliente quanto no servidor. Por exemplo, ele gera o *proxy* ao receber uma referência de um objeto remoto. Para auxiliar a implementação destes módulos utilizamos alguns componentes do lado do cliente e do lado do servidor, conforme descrito a seguir.

- Cliente - *Proxy*:
  O *proxy* tem como objetivo tornar a chamada remota transparente de localização para o cliente. Ele faz isso implementando a interface remota, porém com o intuito de empacotar a requisição, com informação como o método foi chamado e os argumentos utilizados, e enviá-lo para o objeto remoto através do módulo de comunicação. Como ele implementa a interface remota, o *proxy* se apresenta como um objeto local para o cliente. Cada objeto remoto, acessível ao cliente, possui um *proxy* no cliente.
- Servidor - Despachante e Esqueleto:
  O papel do despachante é receber a requisição do módulo de comunicação e identificar o método adequado a ser invocado no esqueleto. O esqueleto também implementa a interface remota, só que de maneira diferente. Ele desempacota os argumentos e finalmente faz a invocação localmente no objeto alvo da chamada remota. O esqueleto espera o método terminar, empacota o resultado e o retorna ao despachante, que repassa a mensagem ao módulo de comunicação, que por sua vez retorna a resposta ao cliente.

Todas estes componentes serão fundamentais na construção da arquitetura do *middleware* RMI para RSSF proposto neste trabalho.

2.2 Middleware para Rede de Sensores Sem Fio

Um *middleware* para RSSF tem como objetivo facilitar o desenvolvimento de aplicações em RSSF removendo o acoplamento das soluções atuais e preenchendo a

lacuna entre os protocolos de rede e aplicação [18]. Para atender esses requisitos é necessário repensar os protocolos e algoritmos convencionais. Os requisitos para essa plataforma de *middleware* variam conforme o tipo e os serviços prestados por ele, embora existam requisitos comuns a todos eles [1], [8] e [24], conforme descritos abaixo:

- Facilidade de Uso: O *middleware* deve facilitar o desenvolvimento de aplicações distribuídas e não adicionar maior complexidade.
- Localização: Os aplicativos de sensores devem usar o *middleware* com a mesma semântica da linguagem de programação do sensor.
- Carga computacional: Como os sensores ainda possuem hardware restrito, é necessário que a implementação do *middleware* seja bastante leve. São necessárias então estratégias preexistentes para isto, como por exemplo geração de *stubs* estáticos com o intuito de diminuir o *overhead* da geração desse dinâmico.
- Adaptação e Escalabilidade: Como sensores podem estar sujeitos a diversas adversidades devido às características dinâmicas de uma RSSF, o *middleware* deve ser capaz de adaptar-se a uma modificação da topologia da rede. Este requisito geralmente é implementado nos sensores atuais através do algoritmo de roteamento *Ad hoc On-Demand Distance Vector* [11].

Com base nesses requisitos as soluções atuais de *middleware* para RSSF [7], em geral, podem ser classificadas nas seguintes categorias:

- Banco de Dados: Soluções que abstrai a RSSF como um grande banco de dados e utiliza consultas em SQL para recuperar dados.
- Espaço de Tuplas: É uma implementação de memória distribuída compartilhada em forma de tuplas aonde aplicativos podem ler, escrever, retirar e ser notificados de tuplas nessa memória. [3]
- Eventos: Utiliza-se de eventos, como *publish/subscribe*, para agregar e ser o meio de comunicação de dados da RSSF.
- Descoberta de Serviços: Utiliza-se de protocolos de descoberta de serviços para localizar sensores específicos para os requisitos da aplicação.

O RMI não está diretamente em nenhuma das categorias utilizadas pelos *middlewares* de RSSF atuais. Sendo que uma categoria adequada para este seria a de objetos distribuídos. A explicação disso é que o foco das pesquisas em *middleware* de RSSF até agora ser na busca de novas soluções de comunicações com base nos limites físicos dos sensores ao invés de utilizar-se/adequar-se soluções existentes.

2.3 Sun SPOT

O Sun SPOT (*Sun Small Programmable Object Technology*) é um sensor e plataforma de RSSF desenvolvido pela Sun com o intuito de estimular e facilitar o desenvolvimento de aplicações em sensores [10]. Tanto a arquitetura do *hardware* quanto o código da plataforma são liberados sobre a licença GNU General Public License (GPLv2). O *hardware* do equipamento é construído em cima do padrão IEEE 802.15.4 que especifica as camadas de enlace e física para pequenos dispositivos de Rede de Área Pessoal (*Personal Area Network* – PAN) [23]. A Figura 2.2 mostra alguns componentes do sensor. Os componentes dele são: [23]:

- Processamento:
  - 180 MHz 32 bit ARM920T core - 512K RAM - 4M Flash
  - 2.4 GHz IEEE 802.15.4 radio com antena integrada
  - Interface USB
- Sensor:
  - 2G/6G acelerômetro (três eixos)
  - Sensor de Temperatura
  - Sensor de Luminosidade
  - 2 *Switches*
  - 8 LEDs de três cores
  - 6 entradas (pinos) analógicos
  - 5 pinos de uso geral de entrada/saída e 4 pinos de alta voltagem
- Bateria:
  - Bateria recarregável de lítio 3.7V e 750 mAh
  - Gerência automática da bateria provida pelo *software*

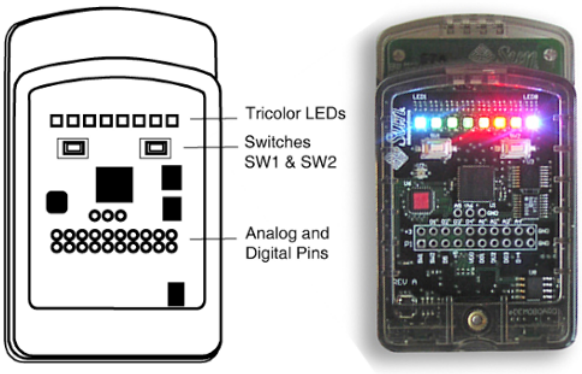

Figura 2.2: *Hardware do Sun SPOT*

O Sun SPOT utiliza a Squawk VM, uma pequena máquina virtual J2ME, escrita quase totalmente em Java, que permite o desenvolvimento de aplicativos na linguagem Java . A Squawk VM possui como configuração a *Connected Limited Device Configuration* 1.1 (CLDC) [13] e implementa o perfil *Information Module Profile* 1.0 (IMP) [12], um subconjunto do *Mobile Information Device Profile* 1.0 (MIDP).

O IMP, assim como o MIDP, especifica o ciclo de vida de suas aplicações através da classe `Midlet` e das operações `startApp()`, `pauseApp()` e `destroyApp()`. O gerenciador de aplicações inicia a *midlet*, classe que extende `Midlet`, através do método `startApp()`. Caso seja necessário pausar a aplicação o método `pauseApp()` é chamado, permitindo a aplicação tratar este estado. A implementação da pause de aplicações ainda não foi realizada no sensor. Por último, o método `destrouApp()` é executado em casso de erro na aplicação, chamada do gerenciador de aplicações ou manualmente pelo usuário. O código 2.1 ilustra uma aplicação típica.

Código 2.1 `SunSpotApplication`

```
1  public class SunSpotApplication extends MIDlet {
2      private ITriColorLEDArray leds =
3              (ITriColorLEDArray) Resources.lookup(ITriColorLEDArray.class);
4
5      protected void startApp() throws MIDletStateChangeException {
6          ISwitch sw1 = (ISwitch) Resources.lookup(
7                          ISwitch.class, "SW1"); // Switch
8          ITriColorLED led = leds.getLED(0);    // Pegue o primeiro LED
9          led.setRGB(100,0,0);                  // Coloca cor vermelha no LED
10
11         while (sw1.isOpen()) {                // Termina app quando apertar sw1
12             led.setOn();                      // Ligue o led
13             Utils.sleep(250);                 // Espere 1/4 segundo
14             led.setOff();                     // Desligue o led
15             Utils.sleep(1000);                // Espere 1 segundo
16         }
17         notifyDestroyed();                    // Notifique que a app terminou
18     }
19
20     protected void pauseApp() {
21         // Não é chamado atualmente pela Squawk VM
22     }
23
24     protected void destroyApp(boolean unconditional)
25                                throws MIDletStateChangeException {
26         System.out.println("App terminou!");
27     }
28 }
29
```

A iniciativa da Sun de trazer a linguagem Java para sensores faz parta da famosa estratégia da empresa *"Write Once, Run Anywhere"*, no sentido de tornar a linguagem e programas desta tecnologia independente de plataformas. Como a linguagem Java é de alto nível e popular, é extremamente fácil para um programador desenvolver aplicativos para o Sun SPOT, ao contrário de outros sensores que utilizam linguagens específicas de sistemas embarcados, como nesC. A Squawk VM roda em cima do processador do sensor [17], i.e., o sensor não possui sistema operacional, o que melhora o processamento e diminui o *overhead*.

Para facilitar a programação a Sun também disponibiliza um SDK para a plataforma Sun SPOT. Neste SDK temos incluso documentação, tutoriais, demos, *plugins* e emulador. Por exemplo o *plugin* para NetBeans permite fazer *debugging* da sua aplicação direto no sensor. O emulador Solarium permite a interação de Sun SPOT virtuais, emulados, com Sun SPOT reais. A Figura 2.3 mostra o emulador Solarium.

Como podemos ver o Sun SPOT não só é um sensor de alta potência como também é uma plataforma ao fornecer um imenso suporte ao desenvolvimento de aplicativos. Por esta razão a implementação exemplo do *middleware* será feita nele.

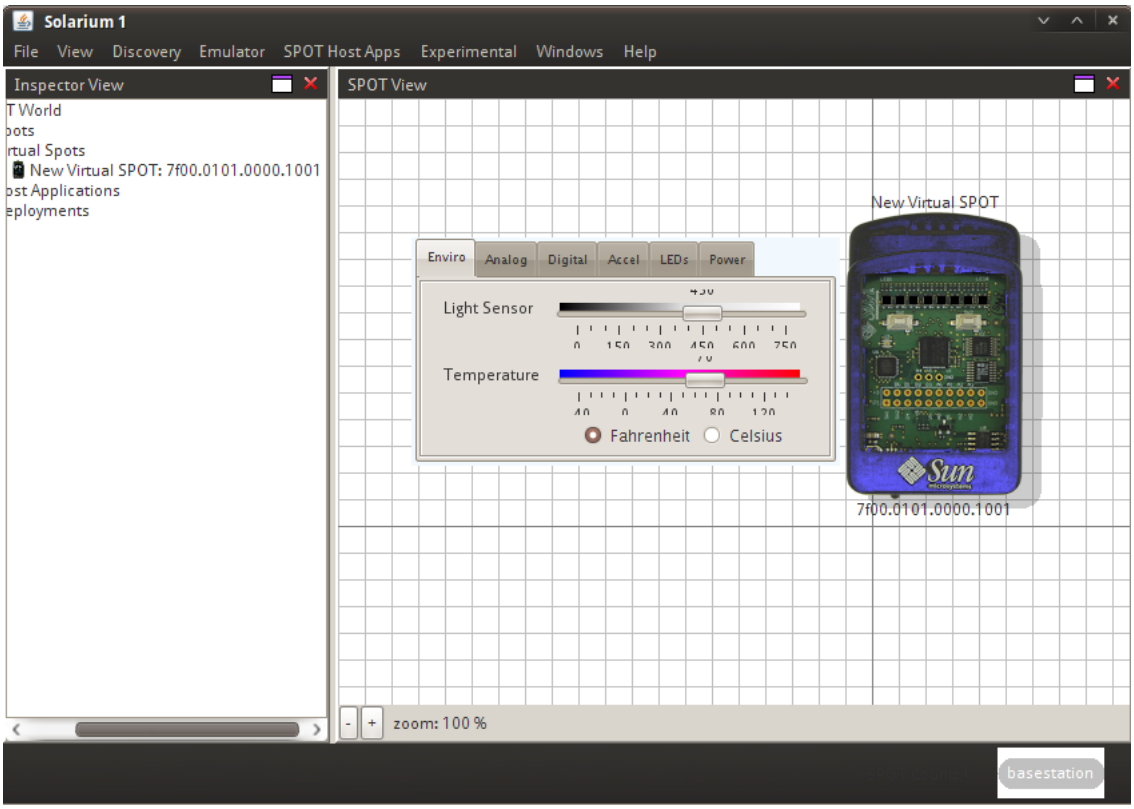

Figura 2.3: *Emulador Solarium*

CAPÍTULO 3

Arquitetura

Para atender os requisitos de um *middleware* de RSSF visto na seção 2.2 como facilidade de uso, transparência e *hardware* restrito projetamos uma arquitetura compacta contendo apenas os módulos essenciais adaptados da teoria de objetos distribuídos. Primeiramente, será descrito uma sucinta visão geral da arquitetura para em seguida aprofundar em cada módulo desta.

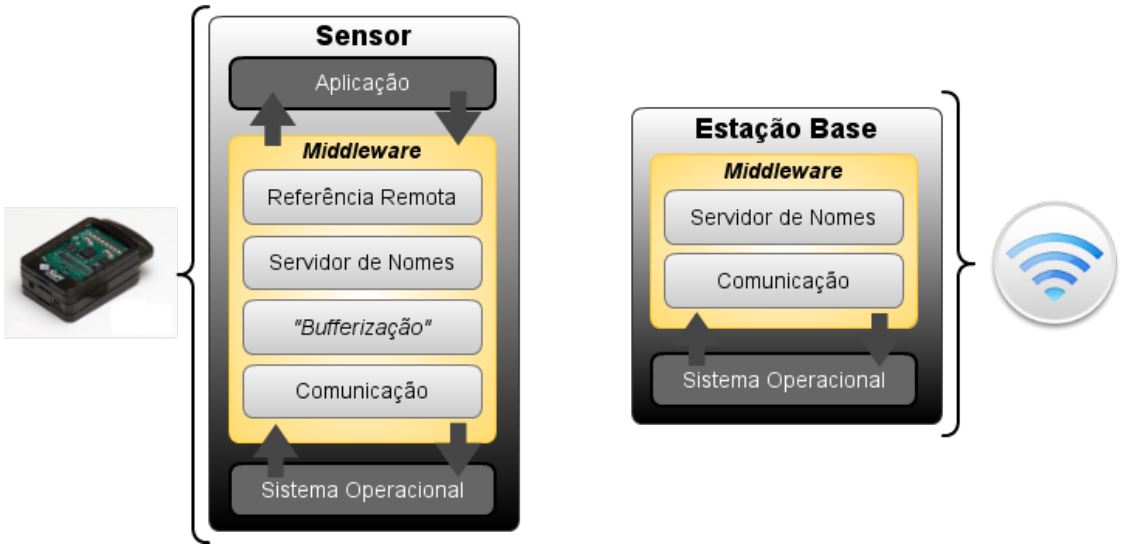

Figura 3.1: *Arquitetura do middleware*

A visão geral da arquitetura para um sensor e estação base é retratada através da Figura 3.1. Nesta aplicação remota qualquer interage com o *middleware* de maneira transparente através dos módulos de referência remota. Os módulos herdam as características descritas da seção de objetos remotos (2.1) com algumas alterações para o ambiente de sensorescla.

O módulo de referência remota é responsável por transformar um objeto remoto em uma referência capaz de receber/enviar requisições. Este módulo utiliza o conceito *proxies* e esqueledos, gerados estaticamente, para realizar esta tarefa. O módulo de servidor de nomes é responsável por operações relacionadas ao anúncio de nome na rede de sensores, como: vinculação, remoção e procura de nomes. Um módulo adicionado a

arquitetura, não existe na arquitetura base, foi o de *bufferização*, que é responsável por um simples algoritmo de enfileiramento de requisições e por permitir que o *buffer* seja tratado por um dado algoritmo de fusão de dados. Já o módulo de comunicação contém a definição de um protocolo compacto de requisição e resposta para uma intercomunicação eficiente entre os módulos do *middleware* para suportar as operações deste. O protocolo é compacto porque trabalha utilizando sempre representações do menor tipo de dado afim de diminuir o número de *bits* transmitidos.

3.1 Módulo de Referência Remota

Um objeto de uma aplicação remota que deseja interagir com outras precisa de uma interface comum, i.e., um contrato com os outros objetos da RSSF, a isso denominamos interface remota. A interface remota especifica quais métodos um objeto servidor está disponibilizando para ser chamado. O objeto que deseja ser chamado (objeto remoto) deve implementar essa interface remota, porém, isso não é suficiente para um objeto cliente invocar os métodos remotos, pois é necessária uma referência a este objeto na rede. Esta referência é composta de identificadores do objeto como IP, porta, interface remota, hora, etc [6].

O Módulo de Referência Remota é responsável por lidar com as tarefas relacionadas a manipulação e controle de objetos remotos, como instanciação e destruição de referência remota, isto é, ele é responsável por capacitar o objeto remoto a enviar/receber requisições. Para isso, este módulo deve criar *proxies* e esqueletos do objeto remoto. No Java RMI eles são criados dinamicamente a partir de carregamento de classes e reflexão. Porém, este cenário é inviável em sensores, uma vez que a aplicação é carregada estaticamente (*flash*) de uma só vez na memória do dispositivo, não permitindo assim o carregamento de novas classes. Para contornar isto desenvolveremos uma ferramenta que gera os *proxies* e esqueletos estaticamente para a aplicação, isto é, eles serão adicionados a suíte de classes da aplicação através de uma ferramenta. Um detalhamento desta será apresentado em 4.5. Ao utilizar *proxies* e esqueletos, nos atendemos os requisitos de facilidade de uso e localização identificados devido estes se comportarem como . Além das competências citadas o módulo também gerencia o número de referências aos *proxies* e esqueletos afins de realizar a coleta de lixo distribuída com o propósito de melhorar a utilização de memória em sensores.

A interface com o módulo de referência remota do lado do servidor é dada pela classe `UnicastRemoteObject` que possui uma única função, a de tornar o objeto remoto acessível a na rede a partir do seu esqueleto. A função recebe um objeto remoto e retorna uma instância de um esqueleto para esse em uma porta aleatória ou pré-definida na chamada da função. O esqueleto é instanciado com um construtor vazio (padrão) e

é injetado neste os metadados para poder receber requisições e encaminhá-la ao objeto remoto alvo. A função de exportação possui as seguintes assinaturas:

- *exportObject(obj : Remote) : Esqueleto*
  Esta versão instancia um esqueleto que escutará por chamadas em uma porta aleatória.
- *exportObject(obj : Remote, port : int) : Esqueleto*
  Esta versão instancia um esqueleto que escutará por chamadas em uma porta pré-definida (*port*).

Após a criação do *esqueleto*, o programa servidor irá registrá-lo utilizando a operação *bind* da classe `Registry` da API de Java RMI. A operação de *lookup* após obter os metadados do servidor de nomes irá instanciar um *proxy* vazio e injetar estes dados para a comunicação com o `Registry` do objeto remoto para repassar ao devido esqueleto. O detalhamento das operações de API de `Registry` serão tratadas na próxima Seção 3.2.

As referências remotas exportadas (*bind*) e obtidas (*lookup*) serão salvas localmente, em uma tabela *LocalRegistry*, para servir de identificação em requisições externas de outros objetos. A Tabela 3.1 mostra um exemplo deste conceito.

Interface Remota	Porta	Esqueleto	Ref. Externas
Temperatura	102	Temperatura_Skel [0x9C]	7
Luz	120	Luz_Skel [0x1A]	2
Robo	129	Robo_Skel [0xF3]	0

Tabela 3.1: *Tabela de registro local que guarda metadados sobre objetos remotos*

Os *proxies* gerados podem ser coletados utilizando o sistema de coleta de lixo local, visto que não são referenciados externamente. Como os esqueletos são referenciados externamente necessitam de uma abordagem diferente.

Para efeito de simplicidade iremos utilizar um simples algoritmo de contagem de referência [5] e a técnica de arrendamento [16]. Durante a operação de *lookup* é notificada a referência remota alvo que esta sendo construído uma referência a este por um dado tempo (pré configurado no *middleware*), prática conhecida como arrendamento. Na tabela de registro local, do objeto referenciado, existe um campo que irar contar o número de referências do objeto. Se no término do período de arrendamento o contrato não for renovado o contador de referência daquele objeto é decrementado. Ao chegar em zero o esqueleto pode ser coletado. Se houver perca de mensagens na rede e o objeto estiver preste a ser removido será prologado o tempo de vida do objeto por mais um período de arrendamento. Se durante este período adicional o objeto não for notificado ele será coletado normalmente. A Tabela 3.1 exemplifica uma tabela local aonde está

sendo contado a referência de vários objetos remotos. Podemos coletar o esqueleto do objeto `Robo`, visto que o mesmo atingiu zero referências externas.

3.2 Servidor de Nomes

O Servidor de Nomes tem o papel de registrar e anunciar os objetos remotos disponíveis na rede. Na prática isto é feito através do mapeamento de um nome a uma referência remota. O anúncio é distribuído a outras estações base através do *broadcasting* e também é armazenado temporariamente (*cache*) no intuito de diminuir a sobrecarga de comunicação nos sensores. É adotado uma arquitetura hierárquica no servidor de nomes, uma versão simplificada da estrutura do DNS. É interessante notar que existe duas formas de localizar o servidor de nomes, são elas:

- Endereço fixo: Requer um conhecimento prévio da localização do servidor de nomes. A vantagem de utilizar esta forma é que está poupa recursos de convergência/descoberta dos sensores porém deixa o nível de abstração de utilização do *middleware* ainda rudimentar.
- Descoberta: Não requer um conhecimento prévio da localização do servidor de nomes. A desvantagem desta forma é o custo de convergência/descoberta dos sensores porém isto aumenta significadamente a abstração de localização e de utilização do *middleware*.

Como mostrado ambas formas possuem vantagens e desvantagens dependendo do tipo de aplicação a ser utilizado. Com isto em mente o *middleware* deve suporta ambas. A solução de endereço fixo é simples, um servidor de nomes neste endereço basta ficar ouvindo em uma porta conhecida.

A solução de descoberta opera da seguinte forma: o sensor no começo realiza um *broadcast* de mensagens de descoberta que contém o código da operação, o seu endereço e uma porta de retorno para o servidor de nomes. Em seguida o sensor começa a ouvir na porta enviada na mensagem de *broadcast*. O servidor de nomes então recebe esta mensagem e retorna a esse uma mensagem de resposta contendo o código de operação de retorno de descoberta de nomes e o endereço do servidor de nomes.

A interface com este módulo é dada em geral pela classe *Registry* que possui as mesmas operações do RMI Java [9]:

- *bind(nome : String, objRemoto : Remote) : void*
  Realiza o vínculo de uma referência remota neste servidor de nomes. O vínculo também é guardado localmente para identificar as chamadas e executar o algoritmo de coleta de lixo distribuída. As exceções possíveis para essa operação são:

  - *AlreadyBoundException*: se o nome já estiver anunciado no servidor de nome.
  - *NullPointException*: se nome ou objRemoto for `null`.
  - *RemoteException*: se houver falha de comunicação com o servidor de nomes.

- *list() : String[]*
  Esta operação retorna um vetor de nomes de objetos anunciados neste servidor de nome. A exceção possível para esta operação é:
  - *RemoteException*: se houver falha de comunicação com o servidor de nomes.

- *lookup(nome : String) : Remote*
  Retorna uma referência remota associada a *nome* no registro. Esta referência então será transformada pelo cliente na interface remota acordada. As exceções possíveis para essa operação são:
  - *NotBoundException*: se o nome não estiver anunciado no servidor de nome.
  - *NullPointException*: se nome for `null`.
  - *RemoteException*: se houver falha de comunicação com o servidor de nomes.

- *rebind(nome : String, objRemoto : Remote) : void*
  Substitui/Realiza o vínculo de uma referência remota no servidor de nomes. O vínculo também é guardado localmente para identificar as chamadas e executar o algoritmo de coleta de lixo distribuída. A diferença desta operação com a *bind* é que esta irá substituir/realizar o vínculo independente da existência ou não do nome em um registro, de maneira que a outra operação irá executar uma exceção se o nome já estiver anunciado no registro. As exceções possíveis para essa operação são:
  - *NullPointException*: se nome ou objRemoto for `null`.
  - *RemoteException*: se houver falha de comunicação com o servidor de nomes.

- *unbind(nome : String) : void*
  Remove o vínculo de uma referência remota associada a *nome* no registro. As exceções possíveis para essa operação são:
  - *AccessException*: se o registro local negar acesso a desvinculação.
  - *NotBoundException*: se o nome não estiver anunciado no servidor de nome.
  - *NullPointException*: se nome for `null`.
  - *RemoteException*: se houver falha de comunicação com o servidor de nomes.

Uma descrição detalhada de cada operação será feita em seguida, vale lembrar que todas possuem mensagens de protocolo definido para intercomunicação entre os módulos do *middleware*. A estrutura do protocolo (*PDU*) destas mensagens será feita na Seção 3.4.

A operação de *bind* irá registrar o objeto tanto localmente quanto externamente (no Servirdor de Nomes). Primeiramente será feita o registro externamente, se a operação

tiver sucesso é realizado a vinculação local. Caso o nome anunciado já esteja vinculado no registro externo iremos tornar uma exceção do tipo *AlreadyBoundException*. Outras exceções possíveis é a *NullPointException* no caso se o objeto a ser associado ser nulo e *RemoteException* em caso de falha de comunicação. Com a operação realizada corretamente é chamado o módulo de referência remota caso não acharmos um esqueleto para instanciarmos um esqueleto daquele objeto remoto (lembre-se que a estrutura do esqueleto fora criado estaticamente antes do *deploy* da aplicação). É injetado no esqueleto o objeto remoto para que o esqueleto repasse as chamadas ao objeto remoto (que implementa a interface remota). Na tabela local será guardado o nome anunciado, a porta do esqueleto, o esqueleto em si, e o número de referências externas a este objeto. A sequência de operações pode ser vista no diagrama de sequência, representado pela Figura 3.2.

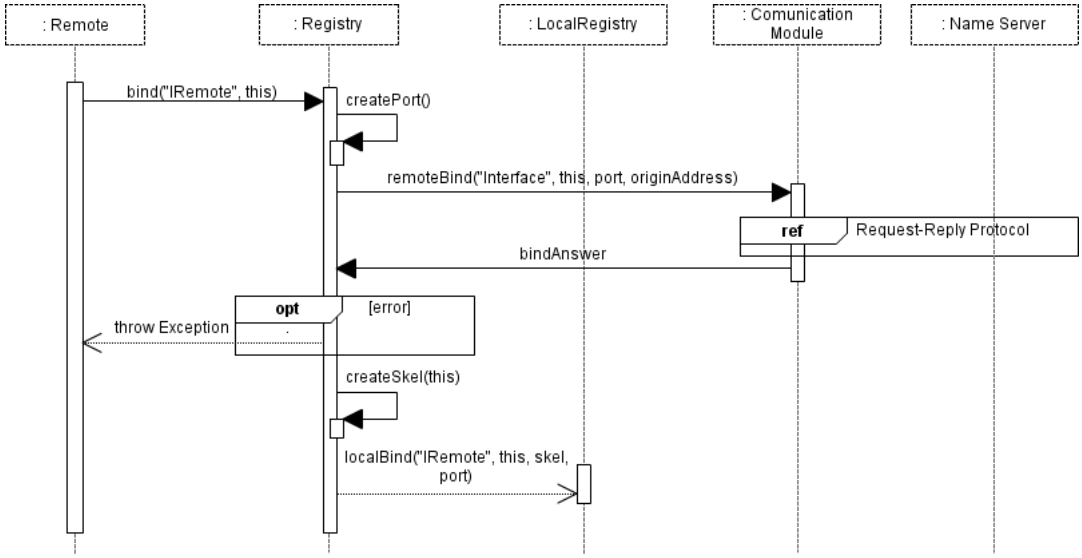

Figura 3.2: *Diagrama de Sequência para a operação de bind*

Se um sensor quiser listar os nomes registrados no servidor de nome este deve utilizar a operação *list*. Esta faz uma consulta ao servidor de nomes que irá retornar um vetor de `String` contendo os nomes anunciados. Caso o servidor de nomes esteja utilizando uma hierarquia de nomes, será necessário para o servidor de nomes percorrer cada servidor de nome conhecido. Isto é, os nomes estarão espalhados em tabelas diferentes na rede. Bastando o servidor procurá-los hierarquicamente, parecido com o DNS. Se houver falha de comunicação será lançado uma exceção do tipo *RemoteException*. A sequência de operações de *list* pode ser vista no diagrama de sequência, representado pela Figura 3.3.

No caso de uma operação de *lookup* o objeto que chamou o *Registry* receberá uma referência remota do servidor de nomes. Primeiramente será feito uma requisição ao servidor de nomes para obter os metadados do objeto alvo, endereço, porta, etc. O

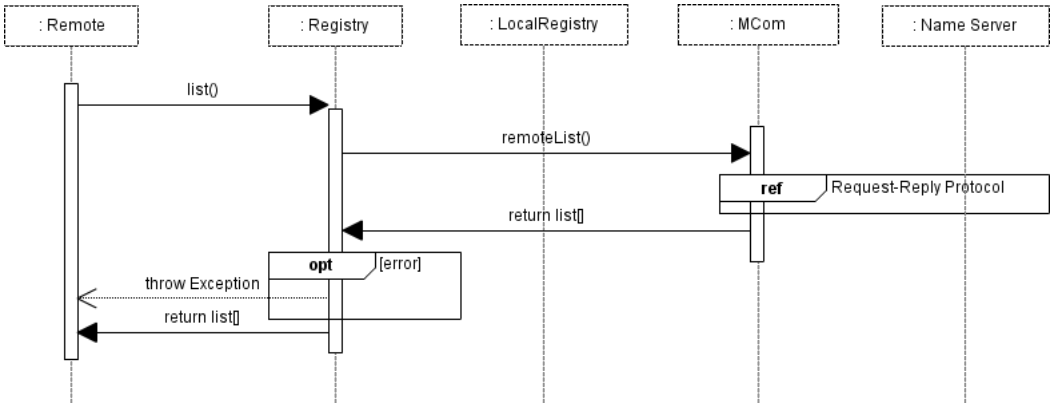

Figura 3.3: *Diagrama de Sequência para a operação de list*

servidor de nomes irá enviar os metadados da referência remota. Logo em seguida é injetado os metadados do objeto remoto nos *proxies* para que estes construam mensagens agregando estes metadados, preenchendo a lacuna necessária para realizar a comunicação com o objeto alvo. Em seguida é feito um arrendamento com o registro do objeto remoto afim de registrar a referência na tabela do objeto remoto (para realizar posteriormente o algoritmo de coleta de lixo). Com os metadados o *proxy* poderá interagir com o módulo de comunicação diretamente. O diagrama de sequência desta operação pode ser visto na Figura 3.4.

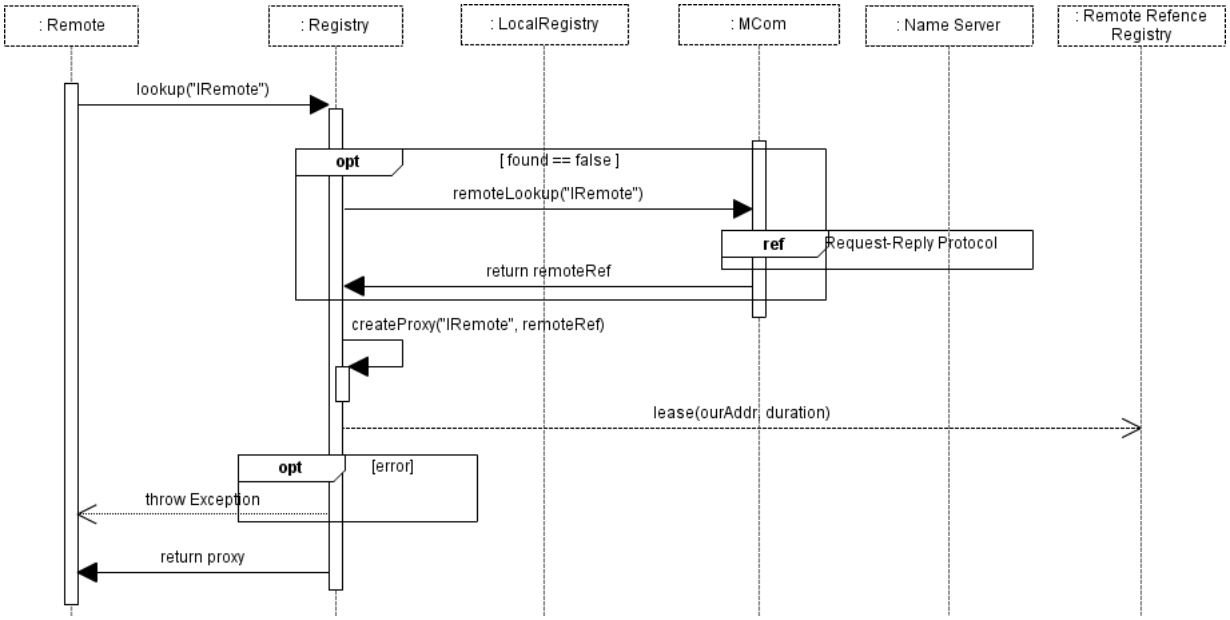

Figura 3.4: *Diagrama de Sequência para a operação de lookup*

A operação de *rebind* possui um fluxo praticamente idêntico sendo a única diferença é que a requisição de associação (*binding*) é assíncrona tornando esta mais rápida do que a operação *bind*. Em caso de erro é lançado uma exceção assíncrona,

os possíveis erros são *NullPointException* e *RemoteException* para erros de nulo e comunicação respectivamente. A Figura 3.5 representa o diagrama de sequência desta operação.

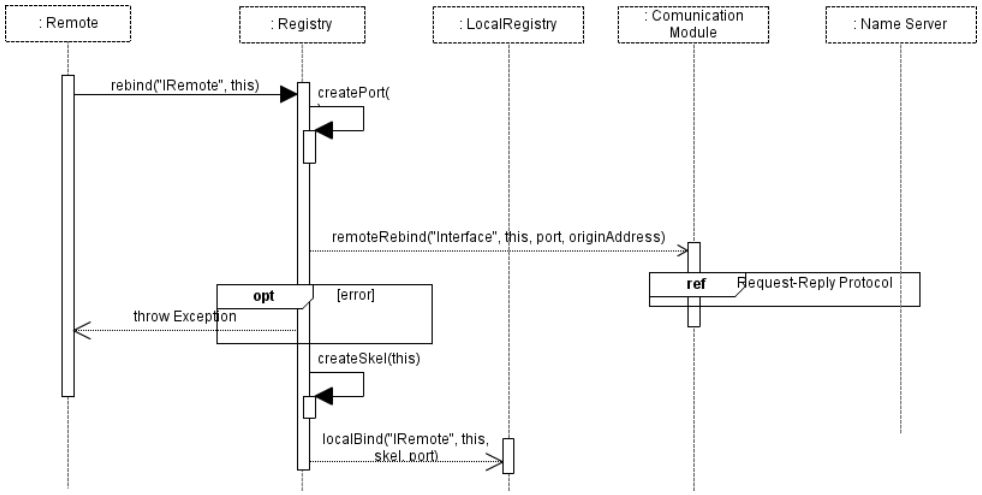

Figura 3.5: *Diagrama de Sequência para a operação de rebind*

A operação de *unbind* também têm uma sequência parecida coma operação de *bind*. Primeiramente ela tenta desvincula o objeto na tabela local, se o objeto não estiver vinculado localmente quer dizer que este registro não havia vinculado este nome no servidor de nomes e logo é lançado uma exceção de acesso negado (*AccessException*). Caso tudo ocorra normalmente, é realizado uma operação para fazer o desligamento no servidor de nomes. Exceções possíveis são *NotBoundException*, *NullPointException* e *RemoteException* em caso de erro de não vinculação no servidor de nomes, referência vazia e comunicação respectivamente. A operação é detalhada no diagrama de sequência representado pela Figura 3.6.

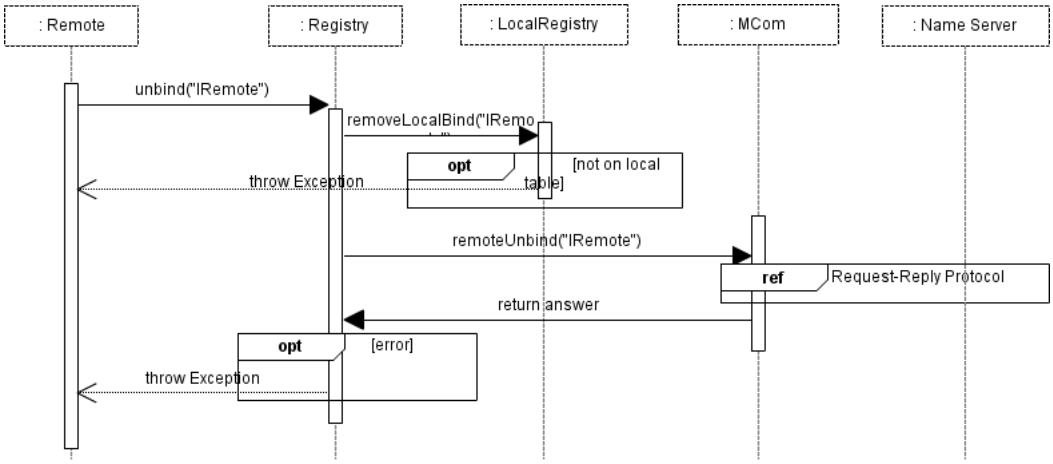

Figura 3.6: *Diagrama de Sequência para a operação de unbind*

3.3 Módulo de *Bufferização*

O Módulo de *Bufferização* tem o papel de enfileirar as requisições no sensor com o objetivo de aumentar a durabilidade do sensor de maneira que atenda o requisito de carga computacional de *middleware* para RSSF. Para fazer esta tarefa, será aplicada dois métodos simples e complementares: um *buffer* de requisições e um algoritmo auxiliar para a fusão de dados. Este módulo é adicional a arquitetura como um todo, sendo que a mesma funciona corretamente sem este módulo, porém, a não implementação deste módulo afeta diretamente o requisito citado. O módulo adota uma abordagem superficial sobre as complicações da t

Afim de diminuir o número de requisições implementamos um *buffer* de requisições que permite acumular as requisições e mandá-las de uma só vez, esta técnica é inspirada no protocolo X11 [15]. A abordagem utilizada é de acumular um número x de requisições e enviá-las caso ocorra uma das seguintes ações:

- Atingiu um número fixo de operações sem retorno;
- Chamou uma requisição com retorno;
- Programa do usuário acionou um método *flush()* para enviar as requisições;
- Nenhuma das ações anteriores ocorreu após um dado tempo (*timeout*).

Antes de enviar as requisições para o módulo de comunicação é realizado uma chamada para um algoritmo de fusão de dados. Esta fusão será feita pelo programa de usuário, aonde temos com entrada um vetor com a fila de requisições (com todas os dados das requisições – argumentos, etc) e como saída teremos um vetor de requisições a ser enviadas. Caso o programa não queira realizar a fusão de dados, ele simplesmente retorna as requisições diretamente, caso contrário, ele pode operar em cima dos valores das requisições afim de detectar requisições próximas e fundi-las. A Figura 3.7 mostra um caso uma requisição com retorno causa todas as requisições do *buffer* a serem enviadas.

3.4 Módulo de Comunicação

Para um objeto enviar uma mensagem para um objeto remoto é necessário um módulo de comunicação, isto é, transmissão de mensagens utilizando um protocolo requisição-resposta. O módulo de comunicação é responsável por tratar destes mecanismos de interação em todos os aspectos (falha, protocolo, etc).

O protocolo de comunicação situa-se na camada de aplicação e é bastante simples. No protocolo será utilizado somente informações essenciais para a operação, sempre utilizando o menor tipo de dado para representá-las. Com um modelo compacto de representação podemos diminuir a mensagem e consequentemente poupar energia de

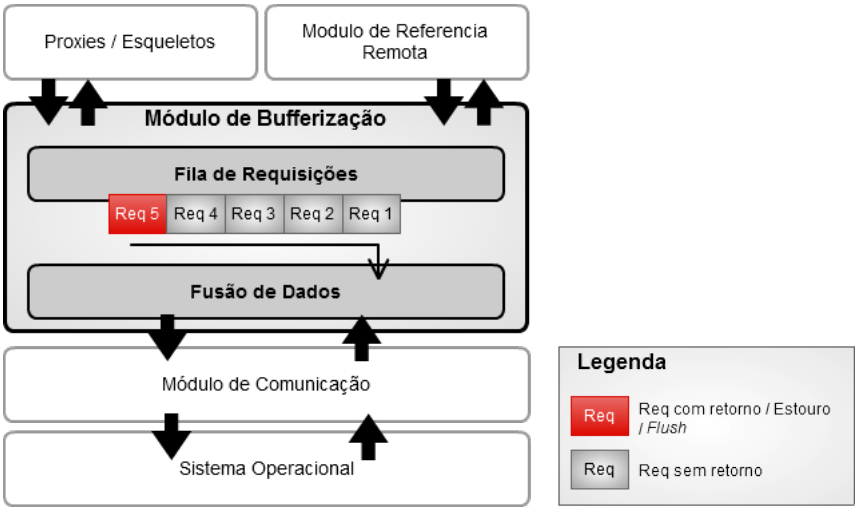

Figura 3.7: *Buffer de Comunicação*

transmissão. A simplicidade e estrutura do protocolo é para atender os requisitos de carga computacional e escalabilidade. A unidade de dados do protocolo (*Protocol Data Unit - PDU*) pode ser dividido em três partes: cabeçalho comum, cabeçalho de requisição ou resposta e por último o cabeçalho específico da operação.

O cabeçalho comum a todas operações é composto de apenas um campo: o código de operação (*operation code – opcode*) representado por um *byte*, que pode ser visto na tabela 3.2.

COMMON HEADER
Opcode (*Byte*)

Tabela 3.2: *Cabeçalho comum a todas operações*

O *opcode* identificam a operação permitindo assim o receptor conhecer a estrutura do outros campos do protocolo. A lista completa com todos os *opcodes* se encontra no Apêndica A. A Tabela 3.3 mostra alguns *opcode*.

OPERATION	CODE
Bind Request	0x07
Bind Reply	0x08
List Request	0x09
List Reply	0x0A

Tabela 3.3: *Alguns códigos de operações*

Em seguida é anexado o cabeçalho comum as operações de requisição e de resposta, que contém dados utilizados por essas operações. O cabeçalho de requisição adiciona o endereço da origem (*address*) para que o servidor possa retornar a mensagem, este campo só é necessário se estivermos trabalhando com um meio de transmissão que

não ofereça estabelecimento de sessão. O cabeçalho de resposta adiciona dois campos: *status* que identifica se a operação ocorreu normalmente, caso contrário é adicionado um campo específico denominado *exception* com o código que identifica esta exceção. Os cabeçalhos estão representados na tabela 3.4.

REQUEST HEADER
Opt-Address (*String - UTF*)

REPLY HEADER
Status (*Byte*)
Opt-Exception (*Byte*)

Tabela 3.4: *Cabeçalhos comuns a operações de requisição e resposta*

Finalmente chegamos ao cabeçalho das operações, que irão anexar os valores específicos das operações. Por exemplo, a operação *Lookup Reply* adiciona dois campos: *port* e *address*, um inteiro e uma *String-UTF* que identifica a porta e endereço respectivamente do esqueleto. O cabeçalho completo da operação é mostrado na Tabela 3.5. O cabeçalho de todas as operações podem ser encontradas no Apêndice B.

Lookup Reply	
Opcode	0x07 *(Lookup Reply Opcode)*
Status	0xAA *(OK)*
Port	30
Address	“00:1b:77:40:f7:4a”

Tabela 3.5: *Cabeçalho da operação Lookup Reply*

Com o protocolo definido utilizaremos a comunicação em baixo nível do sensor, que trabalha com a especificação base IEEE 802.15.4 [2] para trocar estes dados. Essa faixa de comunicação trabalhamos com radiogramas análogo a datagramas (UDP) da rede IP. Existe também um suporte limitado a *radiostream* que fornece comunicação confiável através da adição de uma camada sobre radiogramas. Como dito anteriormente, as requisições irão empacotar meta-dados essenciais sobre a operação desejada.

3.5 Serialização

A serialização dos dados utiliza um mecanismo de reflexão estático proposto por [4]. Os objetos a ser serializados possuem um construtor vazio e implementam a interface denominada `KSNSerializable`. Esta interface possui dois métodos:

- *readObjectOnSensor(DataInput input)*
- *writeObjectOnSensor(DataOutput output)*

Os métodos são utilizados para ler e escrever as variáveis desejadas durante o processo de serialização. O processo consiste de escrever em um *buffer* o nome da classe

e os dados em ordem a ser serializados. Na deserialização é lido o nome da classe e criado uma instância dela (através do construtor vazio). Em seguida é lido os campos na ordem que foram escritos. Este processo, apesar de um pouco rudimentar, contorna os problemas da serialização em sensores.

CAPÍTULO 4

Implementação

Este trabalho produziu uma implementação livre da arquitetura descrita nos capítulos anteriores para a plataforma *Sun SPOT*. O *software* foi denominado **spotSHOUT** e está licenciado sobre a licença ALv2 (*Apache License, Version 2.0*). Como biblioteca opcional ao uso *middleware*, caso o usuário queria adicionar suporte a envio de objetos, nós utilizamos a API de Serialização de [4] para prover mecanismos de serialização estática em sensores. Esta biblioteca também está licenciada sobre a licença ALv2. O logotipo do software e da fundação Apache pode ser visto na figura abaixo.

Figura 4.1: *Logotipo do spotSHOUT e da fundação Apache*

Os módulos descritos pela arquitetura foram implementados em cima da API padrão do Sun SPOT. Para uma maior facilidade do desenvolvedor foi utilizado como interfaces e classes da RMI os mesmos nomes e pacotes da API padrão de Java. A explicação neste capítulo será de feita *bottom-up*. Será explicado primeiramente a implementação do estabelecimento de conexão confiável entre nós. Logo em seguida é feito é descrito como foi feito implementação do protocolo. Após isso é mostrado com os dados são empacotados para o protocolo e o funcionamento dos *proxies* e esqueletos.

4.1 Estabelecimento de Conexão

O estabelecimento de conexão garante uma transmissão confiável e correta de dados entre nós, requisito fundamental em um *middleware* de objetos distribuídos. Para validar este requisito foi abstraído a estrutura e operações de requisição e resposta de um servidor de RMI na classe abstrata `Server`. Esta classe lida com o protocolo de estabelecimento de conexão confiável e com o tratamento e despache de requisições. Logo as classes específicas de registro RMI, tanto o cliente como servidor, que rodam nos

dispositivos, devem somente preocupar-se em sobrescrever o método que o despachante chama na requisição. Com esta escolha obtém-se uma enorme flexibilidade quanto ao protocolo físico de comunicação, visto, que se desejado trocar o protocolo basta modificar esta classe.

O estabelecimento de conexão é realizado pela classe `RMIUnicastConnection` que é uma fábrica de conexões, encapsulando todos os detalhes de conexão entre nós. O registro cliente (`SpotRegistry`) chama o método estático `makeConnection(srvAddr, srvPort)` com o objetivo de obter uma conexão confiável. Com base nos endereços do servidor e a porta do mesmo, representados respectivamente por `srvAddr` e `srvPort` a fábrica inicia a construção da conexão. O estabelecimento de uma conexão confiável é feito através de um pequeno *handshake* entre o `Client` e o `Server`. O `Client` primeiramente envia um radiograma, não confiável, para o `Server` com a operação de abertura de conexão (`RMI_REQUEST`) e o seu endereço. O `Server` obtém a mensagem, aloca uma porta e retorna uma mensagem assíncrona ao `Client` com o número da porta alocada. Logo em seguida cria um `Tunnel` que representa a conexão confiável com o `Client` e inicia está `Thread`. O diagrama de sequência desta operação é representado na figura 4.2.

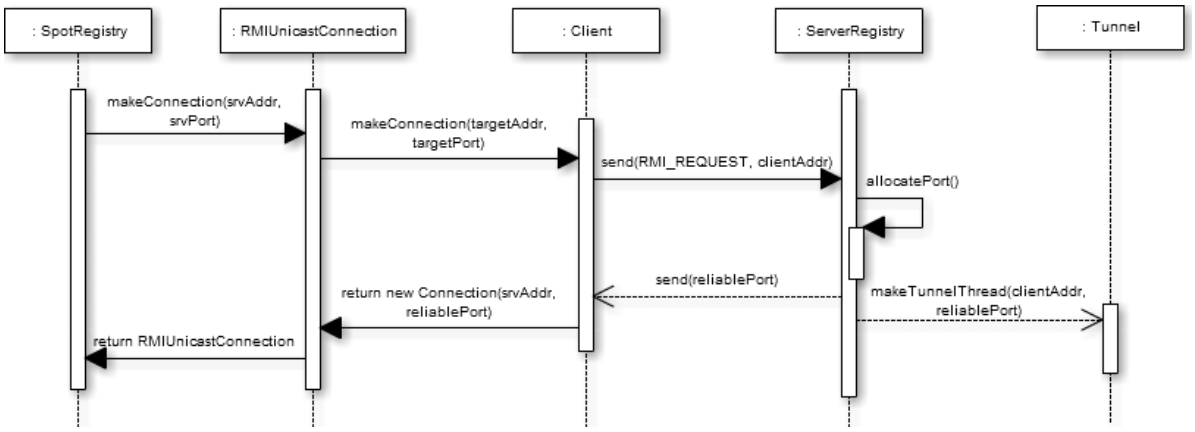

Figura 4.2: *Diagrama de sequência para o estabelecimento de conexão entre nós*

Após o estabelecimento da sessão a classe despacha as requisições para para um método específico `public RMIReply service(RMIRequest request)`, semelhante ao processo adotado na especificação de `Servlet` de J2EE. As classes que queira especificar um servidor (seja de nome ou registro) irão apenas sobrescrever este método - que tem como parâmetro uma requisição RMI (`RMIRequest`) e como saída a requisição RMI de retorno (`RMIReply`). O registro cliente então envia uma requisição RMI (`RMIRequest`) e fica esperando a resposta RMI (`RMIReply`). O registro servidor identifica a operação com base no *opcode* e então constrói o objeto da requisição através de uma pseudo-reflexão. O detalhamento das operações de requisição e resposta serão vistas na próxima seção.

4.2 Operações

Para representar as requisições como mostrado em 3.1, e representar o anexo recursivo de cabeçalhos, foi utilizado extensivamente a propriedade de herança. O cabeçalho comum de uma operação RMI é representado pela classe abstrata `RMIOperation`. O corpo de requisições e respostas de operações RMI são representados respectivamente pelas classes abstratas `RMIRequest` e `RMIReply` que herdam `RMIOperation`. Cada operação de requisição ou resposta então específica seus campos desejados herdando da classe adequada. O diagrama de classe representado pela figura 4.4 demonstra esta hierarquia.

A classe `RMIOperation` possui os métodos abstratos `readData(DataInput input)` e `writeData(DataOutput output)` aonde as requisições irão escrever os dados do protocolo (cabeçalhos + dados específicos). Cabendo a requisição específica apenas a escrita dos dados específicos desta, visto que o cabeçalho é adicionado "automaticamente" devido a característica de herança das operações. Um exemplo desta sequência pode ser vista na figura 4.3 que mostra a escrita detalhada de uma operação *LookupRequest*.

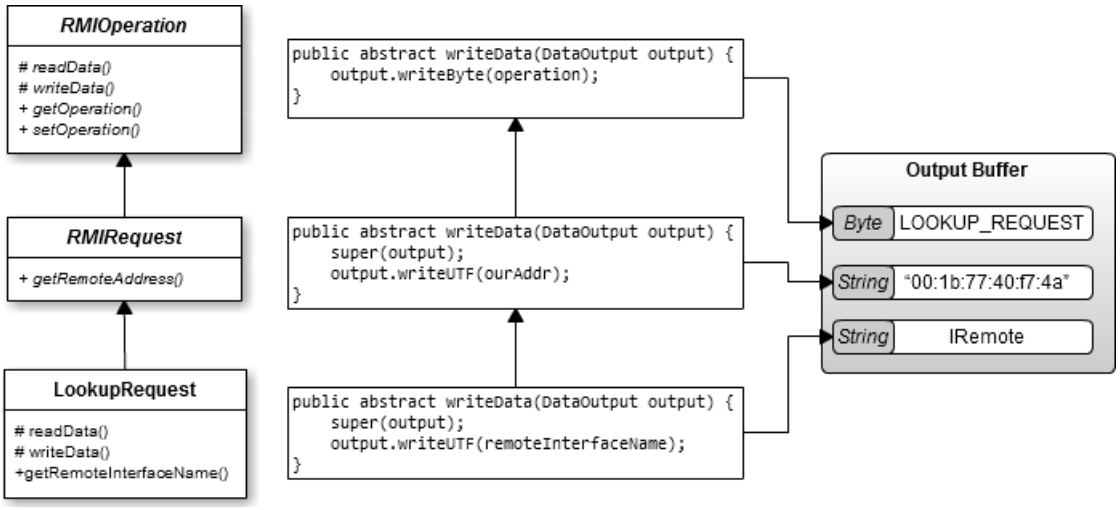

Figura 4.3: *Exemplo de escrita da operação LookupRequest*

Uma operação interessante em ser mostrada é a *InvokeRequest* que representa uma chamada a um objeto remoto. Esta operação encapsula dois atributos: o nome do objeto remoto (para se localizar o esqueleto responsável no registro alvo) e um objeto denominado `TargetMethod` que contém metadados do método a ser chamado. Esse contém um `id`, gerado na criação dos *proxy* e esqueleto, que identifica o método e também um vetor de dados serializáveis que servirão de argumento ao método. Foi criado um conjunto de classes que mapeiam para um tipo primitivo com o intuito de permitir o encapsulamento de tipos primitivos em objetos, para uniformizar o mecanismo de serialização e deserialização que será visto nas próximas seções.

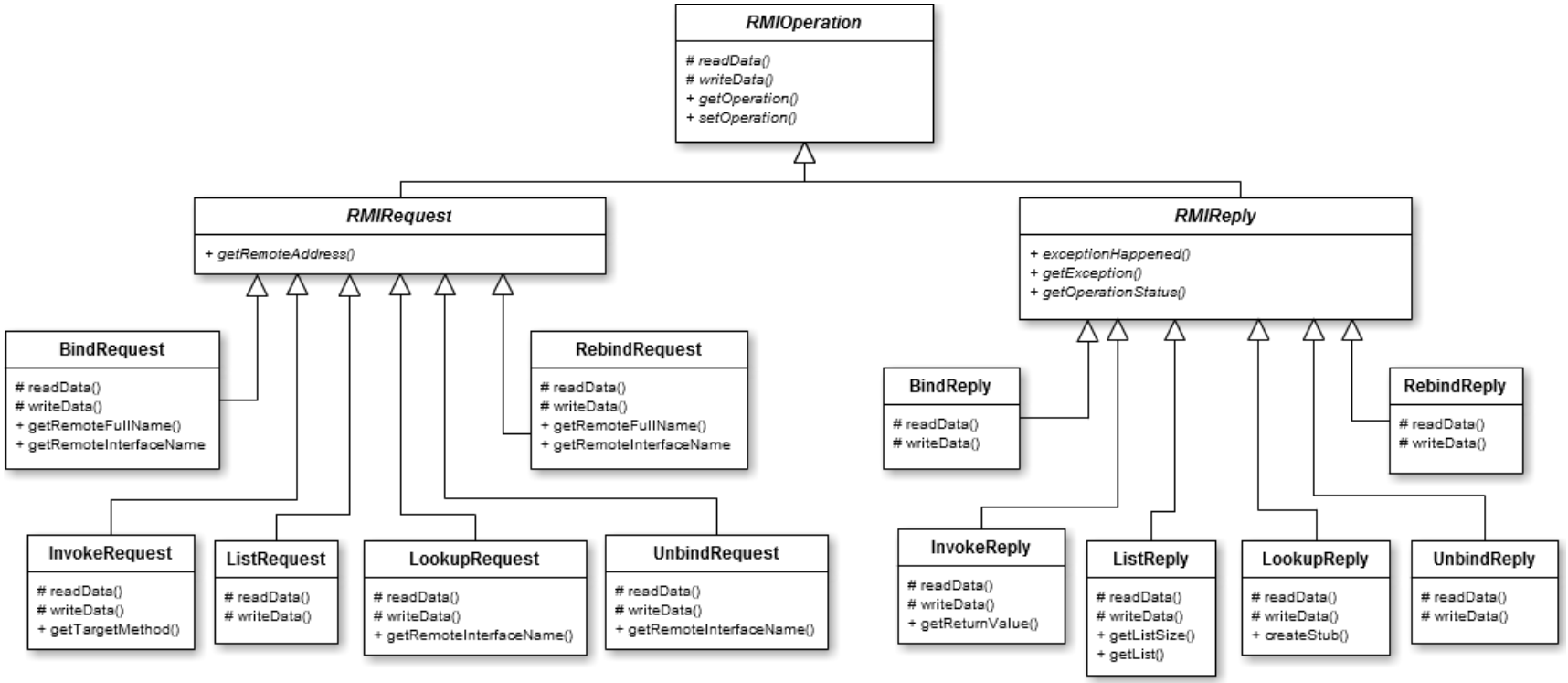

Figura 4.4: *Hierarquia de operações*

4.3 Empacotamento e Armazenamento de Dados

Vimos anteriormente, que para uniformizar a serialização e deserialização dos argumentos é necessário empacotar os dados primitivos em objetos. Foi feito várias classes empacotadoras com esse intuito, como podemos ver na figura 4.5. Cada uma destas implementam os métodos `readObjectOnSensor(DataInput input)` e `writeObjectOnSensor(DataOutput output)` que irão mapear a leitura e escrita ao tipo de dado específico. Fazendo que, na hora da escrita dos argumentos no *buffer* de entrada/saída basta que itere pela lista de argumentos chamando o método de leitura ou escrita.

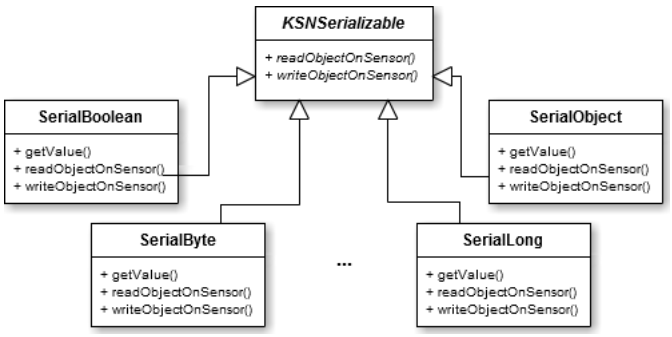

Figura 4.5: *Hierarquia de classes de empacotamento*

As tabelas que guardam os metadados de referências remotas foi representados como um mapa (*HashMap*) de vetores. A tabela de registro nos servidores de nome tem como chave o nome anunciado e referencia um vetor contendo o nome completo da interface remota e o endereço MAC do objeto desta. O mesmo raciocínio é aplicado na tabela local dos registros.

4.4 Esqueletos e *Proxies*

Como vimos em 2.1, *proxies* e esqueletos são componente com o intuito de tornar transparente o funcionamento do *middleware*. O *proxy* implementa a interface remota porém redirecionando as chamadas remotamente ao servidor. Este empacota metadados do método e seus respectivos argumentos. A reflexão é quem determina a ordem e os dados a serem empacotados, que devido a limitação da plataforma SunSPOT é feita estaticamente. Para exemplificar isto vamos mostrar a estrutura desta reflexão. Suponha a interface remota `Calculator` que contenha o método `add` descrito no Programa 4.1.

Código 4.1 `Interface Remota Calculator`

```
1 public interface Calculator extends Remote {
2     public int add(int x, int y) throws RemoteException;
3 }
```

O *proxy* irá primeiramente empacotar os argumentos primitivos em objetos e adicioná-los ao vetor de argumentos. Em seguida é criado o objeto `TargetMethod` com o identificador do método (em ordem decrescente de declaração) e o vetor de argumentos. Este objeto guarda o `id` e o vetor de argumentos. É criado uma requisição de chamada (*InvokeRequest*) com o nome do objeto anunciado junto com os metadados da chamada (*TargetMethod*).

Código 4.2 `Calculator Proxy`

```
1  public int add(int p0, int p1) throws RemoteException {
2          try {
3                  Serializable[] args = new Serializable [2];
4                  args[0] = new SerialInt(p0);
5                  args[1] = new SerialInt(p1);
6
7                  TargetMethod m = new TargetMethod(0, args);
8                  InvokeRequest invReq = new InvokeRequest("Cal", m);
9                  RMIUnicastConnection conn = RMIUnicastConnection.
10                         makeClientConnection(ProtocolOpcode.INVOKE_REQUEST,
11                         getTargetAddr(), RMIProperties.RMI_SPOT_PORT);
12                 conn.writeRequest(invReq);
13
14                 InvokeReply invReply = (InvokeReply) conn.readReply();
15                 conn.close();
16                 if (invReply.exceptionHappened()) throw new RemoteException();
17                 return ((SerialInt)invReply.getReturnValue()).getValue();
18         } catch (IOException ex) {
19                 throw new RemoteException("Remote Exception on add()");
20         }
21  }
```

É realizado a conexão entre os registros e o envio da requisição através da operação `writeRequest()` de `RMIUnicastConnection`. Este por sua vez irá chamar o método `writeData()` do objeto `InvokeRequest` que irá escrever os dados do cabeçalho e em seguida escrever os seus dados específicos. Estes dados são: o nome do objeto

anunciado, o tamanho do vetor de argumentos e o vetor propriamente dito. Como o vetor de argumentos está todo em objetos empacotadores, será chamado internamente uma função das classes empacotadas denominada writeObjectOnSensor(DataOutput output) que escreve o tipo de dado primitivo no *buffer* de saída. O código 4.2 mostra o conteúdo deste *proxy*.

Após o envio da requisição, se o método tiver retorno, é lido o retorno por meio da conexão. Esta operação (`readReply()`) basicamente realiza o processo inverso da escrita. A operação `InvokeReply` contém um único valor (de retorno) encapsulado. Fazendo com que caso não ocorra exceções durante a execução do método seja retornado o valor através do método `getValue()` das classes empacotadoras.

O funcionamento do outro lado, i.e. no esqueleto, é um pouco diferente porém simples. O servidor irá receber a requisição de chamada de método e irá através do nome anunciado localizar o esqueleto responsável. Logo, o esqueleto deve apenas tratar esta requisição.

Código 4.3 `Calculator Skel`

```
1  public RMIReply service(RMIRequest request) {
2          try {
3                  TargetMethod method = ((InvokeRequest) request).getMethod();
4                  KSNSerializableInterface returnValue = null;
5                  Calculator remoteObj = (Calculator) remote;
6
7                  if (method.getMethodNumber() == 0) {
8                          int p0 = ((SerialInt)method.getArgs()[0]).getValue();
9                          int p1 = ((SerialInt)method.getArgs()[1]).getValue();
10                         returnValue = new SerialInt(remoteObj.add(p0, p1));
11
12                         return new InvokeReply(returnValue);
13                 }
14                 return null;
15         } catch (Exception ex) {
16                 InvokeReply reply = new InvokeReply();
17                 reply.setOperationStatus(ProtocolOpcode.OPERATION_NOK);
18                 reply.setException(ProtocolOpcode.EXCEPTION_REMOTE);
19                 return reply;
20         }
21 }
```

Primeiramente o esqueleto irá recuperar o metaobjeto (`TargetMethod`) para em seguida para escolher o método apropriado através do identificador. Após escolha é basicamente desempacotado os argumentos para os tipos primitivos necessário e chamado o método. Caso o método não tenha retorno, é retornado o valor `null` ao registro que invocou o esqueleto e este não irá retornar a mensagem ao *proxy*. No caso da existência do retorno é feito o empacotamento do dado e retornado uma operação *InvokeReply* que recebe como um argumento um tipo de dado empacotado. O código 4.3 mostra o conteúdo deste esqueleto.

4.5 Geração de Esqueletos e *Proxies*

Para gerar o código dos *proxies* e esqueletos foi desenvolvido uma ferramenta que a partir da biblioteca padrão de reflexão da linguagem Java gera os arquivos-fontes destes. A ferramenta recebe um aplicativo (jar), o pacote da interface remota, o nome da interface remota e por último o nome desejado a ser vinculado no servidor de nomes [1] A partir destes parâmetros o programa carrega a interface remota e começa o processamento reflexivo para gerar o código fonte do *proxy* e esqueleto conforme a estrutura destes mencionada na seção anterior. Com o código fonte em mão o programa executa a compilação do *proxy* e esqueleto e empacota estes em um jar, bastando o usuário adicionar o jar ao caminho de construção (*build path*) do projeto. A figura 4.6 mostra a interface gráfica do programa.

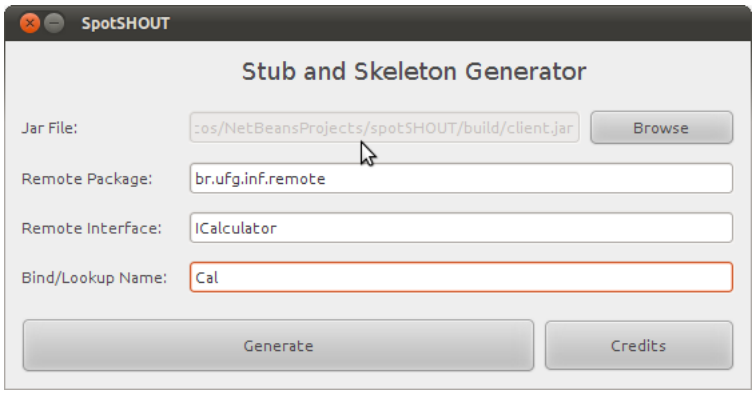

Figura 4.6: *Interface gráfica para geração de esqueletos e proxies*

[1] O grande número de parâmetros fornecido é devido a incapacidade de trabalhar-se com reflexão no sensor, tornando necessário este processamento prévio.

4.6 Avaliação

Nesta seção iremos mostrar alguns dados obtidos pela experimentação do *middleware*. Para realizar os experimentos foi criado um aplicativo simples de calculadora remota e para completar utilizamos dados de programas *demo* presentes no SDK do SunSPOT. Primeiramente iremos comparar o tamanho da implementação do *middleware* com outros da plataforma SunSPOT. No gráfico 4.7 vemos que o spotSHOUT possui um tamanho extremamente pequeno (apenas 0,33% da memória total do sensor). Este tamanho permite a utilização deste tranquilamente pelas diversas aplicações da plataforma. Também é notavél que apesar dos pequenos tamanhos, o spotSHOUT é menor do que os outros *middlewares*.

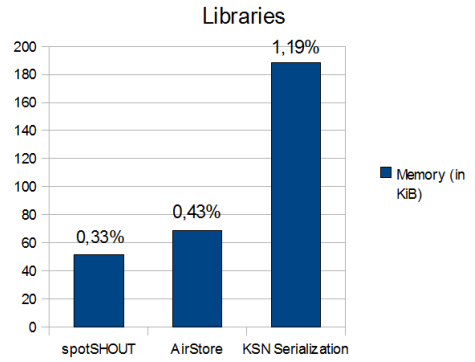

Figura 4.7: *Comparação por tamanho (KiB)*

O uso de memória por uma aplicação que utiliza o SunSPOT é muito similar a utilização de uma aplicação sem o *middleware*. A diferença de memória é bastante pequena entre os dois casos. Em geral as aplicações testadas tiveram um rendimento excelente utilizando apenas em média 17% da memória RAM disponível do sensor (512 KiB RAM). No gráfico 4.8 observamos este comportamento.

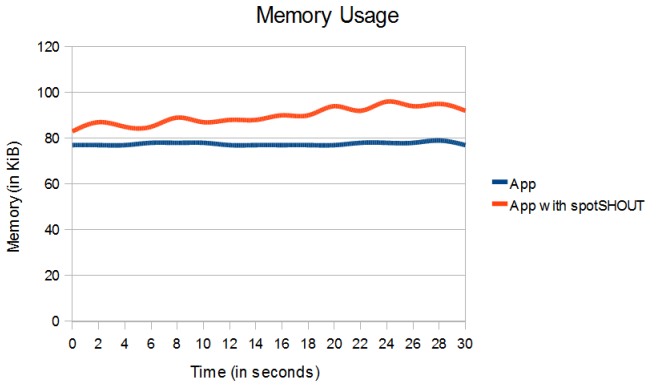

Figura 4.8: *Comparação por uso de memória*

O número de linhas escritas em média por programas é bem menor quando utilizado o *middleware*, como podemos ver no gráfico 4.9. Isto é explicado devido aos vários serviços providos pelo *middleware* que devem ser realizado manualmente na ausência dele.

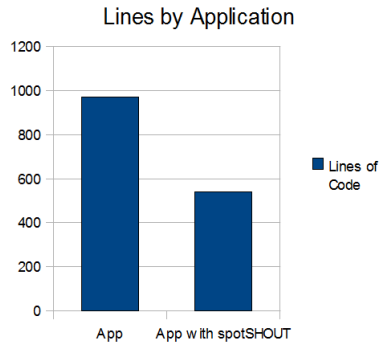

Figura 4.9: *Comparação por linhas de código*

CAPÍTULO 5

Trabalhos Relacionados

Vários *middlewares* foram desenvolvidos para RSSF, porém a maioria foca em novos mecanismos de utilização de comunicação. [8] propõe um projeto de RPC transparente e leve para o sistema TinyOS. Infelizmente a linguagem nesC, do sistema TinyOS, dificulta a criação de aplicações devido a limitação de abstração. Por isso o *middleware* requer muito intervenção manual do programador nele para o seu funcionamento. O projeto apresenta as mesmas restrições quanto a questão de *proxies* e esqueletos, estes devem ser gerados estaticamente e acoplado na aplicação. Infelizmente esta foi a única referência que encontramos de uma aproximação de RPC/RMI para RSSF. Outras abordagens 'semelhantes' serão descritas abaixo.

Mires [19] é um *middleware* orientado a eventos, implementando o paradigma *publish/subscribe*, focado na facilidade de uso e nos limites físicos da RSSF. Este trabalha com a ideia de agregação de dados. Ele especifica um algoritmo de roteamento *multi-hop* visto que o sistema não fornece esta escolha, com o objetivo de a cada nó agregar mais dado para ser disseminado.

TinyLIME [20] é um *middleware* que utiliza o conceito de espaço de tuplas, abstraindo a comunicação entre os sensores através de um modelo de memória compartilhada. Com isso em mente ele permite aplicações fora da RSSF visualizar esta como um grande banco de dados. Permitindo assim operações de SQL sobre a rede.

CAPÍTULO 6

Conclusão

O trabalho conseguiu realizar a arquitetura de um *middleware* de objetos distribuídos para RSSF de maneira que validasse os requisitos identificados de *middleware* para RSSF, como: carga computacional, facilidade de uso e localização. Infelizmente o trabalho não aborda questões mais específicas quanto a problemas físicos de RSSF, dando uma visão superficial deles. O trabalho conseguiu fazer uma implementação satisfatória e rápida para a plataforma SunSPOT.

Ao longo deste trabalho solidifiquei os meus conhecimentos em *middlewares* e principalmente em chamada de métodos remotos (RMI). Também aprendi bastante sobre RSSF, aonde pude notar a complexidade de trabalhar com um ambiente tão dinâmico como este, em que modificações constantes são regras e não exceções. Acredito que o desenvolvimento deste *middleware* irá aumentar a capacidade de abstração de programas em RSSF permitindo à construção de códigos mais sofisticados do que coleta de dados.

Como trabalho futuro quero aumentar a robustez do projeto como um todo, especialmente a parte de *bufferização* e construir abstrações em cima deste *middleware*. Uma possível vertente é a construção de um *middleware* orientado a mensagem (MOM) que usaria RMI no seu núcleo na entrega de mensagens.

Referências Bibliográficas

[1] AGUILAR, J. P.; KOFUGI, S. T. **Middleware para redes de sensores sem fio**. In: *Primeiro Workshop de Redes de Sensores Sem Fio*, 2006.

[2] ALLIANCE, Z. **Zigbee faq**. http://www.zigbee.org/About/FAQ.aspx, 06 2010.

[3] ANGERER, B. **Space-based programming - o'reilly media**. http://onjava.com/pub/a/onjava/2003/03/19/java_spaces.htmll, 03 2003.

[4] BESTEHORN, M.; KESSLER, S.; LEPPERT, A.; MEISINGER, S. **Ksn serialization**. http://www.ipd.uni-karlsruhe.de/KSN/Serialization/, 10 2010.

[5] BEVAN, D. I. **Distributed garbage collection using reference counting**. In: *Proceedings of the Parallel Architectures and Languages Europe, Volume I*, p. 176–187, London, UK, 1987. Springer-Verlag.

[6] COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. **Sistemas Distribuídos Conceitos e Projeto**. Bookman, quarta edição edition, 2007.

[7] 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 (MidSens 2006)*, p. 60–65. ACM Press, 2006.

[8] MAY, T. D.; DUNNING, S. H.; DOWDING, G. A.; HALLSTROM, J. O. **An rpc design for wireless sensor networks**. *International Journal of Pervasive Computing and Communications*, 2:384–397, 2006.

[9] MICROSYSTEMS, S. **Registry (java 2 platform se 5.0)**. http://download.oracle.com/javase/1.5.0/docs/api/java/rmi/registry/Registry.html, 09 2010.

[10] MICROSYSTEMS, S. **Sunspotworld - home**. http://www.sunspotworld.com, 2010.

[11] PERKINS, C. E.; ROYER, E. M. **Ad hoc on-demand distance vector routing**. In: *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, p. 90–100, 1999.

[12] PROCESS, J. C. **Jsr 195: Information module profile**. http://jcp.org/en/jsr/detail?id=139, 07 2003.

[13] PROCESS, J. C. **Jsr 139: Connected limited device configuration 1.1**. http://jcp.org/en/jsr/detail?id=139, 11 2007.

[14] RÖMER, K.; KASTEN, O.; MATTERN, F. **Middleware challenges for wireless sensor networks**. *Mobile Computing and Communications Review*, 6:12, 2002.

[15] SCHEIFLER, R. W.; GETTYS, J. **The x window system**. *ACM Trans. Graph.*, 5(2):79–109, 1986.

[16] SCHNEIDER, M.; ALEKSY, M.; SCHADER, M.; TAKIZAWA, M. **Leasing variants in distributed systems**. In: *CISIS '07: Proceedings of the First International Conference on Complex, Intelligent and Software Intensive Systems*, p. 68–73, Washington, DC, USA, 2007. IEEE Computer Society.

[17] SIMON, D.; DANIELS, J.; CIFUENTES, C.; CLEAL, D.; WHITE, D. **Java(tm) on the bare metal of wireless sensor devices - the squawk java virtual machine**. In: *Proceedings of the 2nd International Conference on Virtual Execution Environments*, 2006.

[18] SOHRABY, K.; MINOLI, D.; ZNATI, T. **Wireless Sensor Networks: Technology, Protocols, and Applications**. Wiley-Interscience, 2007.

[19] SOUTO, E.; GUIMAR AES, G.; VASCONCELOS, G.; VIEIRA, M.; ROSA, N.; FERRAZ, C. **A message-oriented middleware for sensor networks**. In: *MPAC '04: Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, p. 127–134, New York, NY, USA, 2004. ACM.

[20] TINYLIME. **Tinylime: Lime for sensor networks**. http://lime.sourceforge.net/tinyLime/index.html, 2010.

[21] WIKIPEDIA. **List of wireless sensor nodes — wikipedia, the free encyclopedia**. http://en.wikipedia.org/w/index.php?title=List_of_wireless_sensor_nodes&oldid=365096044, 04 2010.

[22] WIKIPEDIA. **Middleware — wikipedia, the free encyclopedia**, 2010. [Online; accessed 26-November-2010].

[23] WIKIPEDIA. **Sun spot — wikipedia, the free encyclopedia**, 2010. [Online; accessed 15-November-2010].

[24] YU, Y.; KRISHNAMACHARI, B.; PRASANNA, V. K. **Issues in designing middleware for wireless sensor networks**. *IEEE Network*, 18:15–21, 2003.

[25] ZHAO, F.; GUIBAS, L. J. **Wireless Sensor Networks**. Elsevier, 2004.

APÊNDICE A

Opcodes

Este apêndice tem como objetivo listar todos os códigos de operações do *middleware*. Estes são representados por um *byte*. Os códigos são utilizados para a identificação do tipo da mensagem no protocolo. O corpo de todas as operações do protocolo (PDU) pode ser vista no apêndice B.

Status de datagrama para operação de estabelecimento de conexão confiável:

OPERATION	CODE
ACK	0xAA
NACK	0xBB

Status de execução de uma operação RMI:

OPERATION	CODE
OPERATION_OK	0xCC
OPERATION_NOK	0xDD

Operações de anúncio entre registros:

OPERATION	CODE
HOST_ADDR_REQUEST	0x01
HOST_ADDR_REPLY	0x02
REGISTRY_REQUEST	0x03
REGISTRY_REPLY	0x04
INVOKE_REQUEST	0x05
INVOKE_REPLY	0x06

Exceções RMI:

OPERATION	CODE
EXCEPTION_ALREADY_BOUND	0x11
EXCEPTION_NOT_BOUND	0x12
EXCEPTION_NULL_POINT	0x13
EXCEPTION_REMOTE	0x14

Operações de RMI:

OPERATION	CODE
BIND_REQUEST	0x07
BIND_REPLY	0x08
LIST_REQUEST	0x09
LIST_REPLY	0x0A
LOOKUP_REQUEST	0x0B
LOOKUP_REPLY	0x0C
REBIND_REQUEST	0x0D
REBIND_REPLY	0x0E
UNBIND_REQUEST	0x0F
UNBIND_REPLY	0x10

APÊNDICE B

Unidade de dados do protocolo

Este apêndice lista a estrutura de dados do protocolo das operações especificadas no *middleware*. Campos que possuem *Opt* são opcionais e serão aplicados em caso de exceção do protocolo.

Cabeçalho comum a todas operações:

COMMON HEADER
Opcode (*Byte*)

Cabeçalho comum a operações de requisições e resposta:

REQUEST HEADER
Opt-Address (*String - UTF*)

REPLY HEADER
Status (*Byte*)
Opt-Exception (*Byte*)

Operações:

BIND_REQUEST
Remote Interface Name (*String - UTF*)
Remote Full Name (*String - UTF*)
Remote MAC Address (*String - UTF*)

BIND_REPLY

INVOKE_REQUEST
Remote Interface Name (*String - UTF*)
TargetMethod Serialized Size (Int)
TargetMethod(Serializable)

INVOKE_REPLY
Return Serialized Size (*Int*)
Return Value (*Serializable*)

LIST_REQUEST

LIST_REPLY
List Size (*Int*)
* List Element Value (*String - UTF*)

LOOKUP_REQUEST
Remote Interface Name (*String - UTF*)

LOOKUP_REPLY
Remote Interface Name (*String - UTF*)
Remote Full Name (*String - UTF*)
Remote MAC Address (*String - UTF*)

REBIND_REQUEST
Remote Interface Name (*String - UTF*)
Remote Full Name (*String - UTF*)
Remote MAC Address (*String - UTF*)

REBIND_REPLY

UNBIND_REQUEST
Remote Interface Name (*String - UTF*)

UNBIND_REPLY