

**Augusto Born de Oliveira**

***Integração do Grid e Redes de Sensores sem Fio  
através do POP-C++***

Florianópolis – SC

2006/2

**Augusto Born de Oliveira**

***Integração do Grid e Redes de Sensores sem Fio  
através do POP-C++***

Trabalho de conclusão de curso apresentado  
como parte dos requisitos para obtenção do grau  
de Bacharel em Ciências da Computação.

Orientador:

Prof. Dr. Antônio Augusto Medeiros Fröhlich

BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO  
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA  
CENTRO TECNOLÓGICO  
UNIVERSIDADE FEDERAL DE SANTA CATARINA

Florianópolis – SC

2006/2

# *Resumo*

O t3pico da intera3o3o entre Redes de Sensores sem Fios (RSSFs) e o Grid tem recebido relativamente pouca aten3o3o cient3fica, e a interface entre a fonte dos dados e as aplica3o3es que usam esses dados continua um problema para o arquiteto de sistema. Os esfor3os de integra3o3o existentes na bibliografia inserem uma interface de consultas similar 3 de um banco de dados entre a RSSF e a aplica3o3o, fazendo com que esta fronteira n3o seja transparente.

Para permitir que m3ltiplas aplica3o3es do Grid usem os dados das RSSFs transparentemente, este trabalho estende o POP-C++, que consiste em uma linguagem e um sistema de suporte de tempo de execu3o3o para programa3o3o do Grid. A extens3o3o do POP-C++ desenvolvida neste trabalho torna poss3vel a utiliza3o3o de um s3o modelo de programa3o3o que abrange o conjunto do Grid com as RSSFs, sem incorrer em sobrecustos excessivos.

# *Sumário*

## **Lista de Figuras**

## **Lista de Tabelas**

<b>1</b>	<b>Introdução</b>	p. 8
<b>2</b>	<b>Redes de Sensores sem Fio</b>	p. 10
2.1	Comunicação . . . . .	p. 10
2.2	Sensoriamento . . . . .	p. 11
2.3	Plataformas Comerciais . . . . .	p. 12
2.3.1	Mica . . . . .	p. 12
2.3.2	BTnode . . . . .	p. 12
<b>3</b>	<b>Grid</b>	p. 14
3.1	Organizações Virtuais . . . . .	p. 14
3.2	Arquitetura do Grid . . . . .	p. 14
3.3	Plataformas Comerciais . . . . .	p. 15
3.3.1	Globus . . . . .	p. 16
3.3.2	gLite . . . . .	p. 16
<b>4</b>	<b>POP-C++</b>	p. 18
4.1	Semânticas de Método . . . . .	p. 19
4.2	Descritor de Objeto . . . . .	p. 20
4.3	Extensão do C++ . . . . .	p. 21

4.4	Geração de Código . . . . .	p. 22
4.5	Sistema de Suporte de Tempo de Execução . . . . .	p. 25
4.5.1	Job Manager . . . . .	p. 26
4.5.2	Code Manager . . . . .	p. 26
4.5.3	Comboxes . . . . .	p. 27
<b>5</b>	<b>Estendendo POP-C++ para RSSFs</b>	<b>p. 28</b>
5.1	Sistema de suporte de tempo de execução POP-C++ para RSSFs . . . . .	p. 28
5.1.1	Estratégia de Adaptação para RSSFs . . . . .	p. 29
5.1.2	Diferenças fundamentais entre os nodos do Grid e das RSSFs . . . . .	p. 29
5.1.3	Arquitetura do sistema de suporte adaptado . . . . .	p. 31
5.2	Interação entre o Grid e a RSSF . . . . .	p. 31
5.2.1	Endereçamento . . . . .	p. 34
5.2.2	Broker de Proxy . . . . .	p. 34
5.2.3	Adaptador Grid-RSSF . . . . .	p. 36
5.3	Processo de Geração de Código Objeto . . . . .	p. 36
5.4	Desafios de Projeto . . . . .	p. 39
5.4.1	Carga na Rede . . . . .	p. 39
5.4.2	Escalabilidade . . . . .	p. 39
5.4.3	Segurança/QoS . . . . .	p. 39
<b>6</b>	<b>Avaliação do POP-C++ em RSSFs</b>	<b>p. 41</b>
6.1	Teste de Avaliação de Performance . . . . .	p. 41
6.1.1	Plataforma de Hardware/Software do Teste . . . . .	p. 42
6.1.2	Medição de Performance . . . . .	p. 43
6.2	Teste de Prova de Conceito . . . . .	p. 43
6.2.1	Plataforma de Hardware/Software do Teste . . . . .	p. 44

6.2.2	Medição de Performance . . . . .	p. 44
<b>7</b>	<b>Trabalhos Relacionados</b>	<b>p. 47</b>
7.1	Cougar . . . . .	p. 47
7.2	TAG: Tiny AGgregation Service . . . . .	p. 47
7.3	TinyDB . . . . .	p. 48
7.4	Hourglass . . . . .	p. 48
7.5	Comentários . . . . .	p. 49
<b>8</b>	<b>Conclusão</b>	<b>p. 50</b>
	<b>Referências Bibliográficas</b>	<b>p. 51</b>

# *Lista de Figuras*

2.1	Nodos Mica2 . . . . .	p. 12
2.2	Nodo BTnode . . . . .	p. 13
3.1	Mapeamento das camadas da arquitetura do Grid na arquitetura da Internet . .	p. 15
4.1	Linha de tempo detalhando a ordem de execução de métodos com semântica especial. . . . .	p. 20
4.2	Exemplo de declaração de uma classe paralela POP-C++ . . . . .	p. 23
4.3	Cadeia de geração de código binário a partir de código POP-C++. . . . .	p. 23
4.4	Diagrama de localização e interação da Interface, Broker e Objeto. . . . .	p. 24
4.5	Diagrama de Sequência de uma Invocação Remota de Método . . . . .	p. 24
4.6	Diagrama de Classes do Sistema de Suporte de Tempo de Execução . . . . .	p. 25
5.1	Classe POP-C++ SensorNode Básica . . . . .	p. 29
5.2	Diagrama de localização e interação dos componentes em RSSF. . . . .	p. 30
5.3	Diagrama de Classes do Sistema de Suporte de Tempo de Execução para RSSFs	p. 32
5.4	Interação do Grid com uma RSSF. . . . .	p. 33
5.5	Interação do Grid com uma RSSF, com o Broker de Proxy. . . . .	p. 35
5.6	Processo de geração de código binário para POP-C++ em RSSFs . . . . .	p. 38
6.1	Parclass SensorTest usada no teste de performance . . . . .	p. 42
6.2	Código de aplicação do teste de performance . . . . .	p. 42
6.3	Implementação nativa do teste de performance . . . . .	p. 45
6.4	Comparação de Tamanho de Pacote . . . . .	p. 46
6.5	Comparação de Pedidos por Segundo . . . . .	p. 46
6.6	Organização dos componentes do teste de prova de conceito . . . . .	p. 46

# *Lista de Tabelas*

2.1	Comparação entre o Mica2 e o BTnode . . . . .	p. 13
-----	---	-------



# *1 Introdução*

O contínuo avanço dos processos de fabricação de hardware e a constante miniaturização dos microcontroladores e módulos de memória permite a criação de plataformas inteiras, incluindo interfaces analógicas, num só chip. A combinação do avanço tecnológico com a necessidade de extração de informação de uma área física extensa resultou na criação das Redes de Sensores sem Fio (RSSFs), compostas por nodos com dimensões e custo de fabricação reduzidos, para que possam ser distribuídos de forma pervasiva porém não-intrusiva no ambiente que devem monitorar. O requisito de distribuição em massa acarreta na necessidade de operação e comunicação sem uma conexão física com o resto da rede, ou seja, dependente em baterias. A maximização do tempo de vida da RSSF, portanto, depende de grande preocupação com gerência eficiente de energia.

Concorrente ao surgimento das RSSFs, o modelo computacional de Grid emergiu na década de 1990 para suprir a necessidade de computação de alta performance para aplicações científicas, utilizando computadores em rede para distribuir a carga de trabalho a ser realizada. No seu artigo seminal, Ian Foster [1] define o problema do Grid como “compartilhamento controlado e coordenado de recursos entre organizações virtuais dinâmicas e escaláveis”. Neste problema, as RSSFs podem ser consideradas recursos cujo compartilhamento é de grande interesse para organizações.

Apesar das Redes de Sensores sem Fio terem sido foco de muitos esforços de pesquisa nos últimos anos, o tópico da interação entre elas e outros sistemas computacionais tem recebido relativamente baixa atenção. Os esforços de pesquisa que objetivam fazer esta ponte, como o TinyDB [2] ou o Cougar [3], abstraem os nodos sensores individuais e dão acesso à RSSF como um todo, permitindo que aplicações façam consultas como fariam em um banco de dados. Enquanto esse nível de abstração possibilita a otimização de consultas, minimizando o número de mensagens a serem trocadas pela rede sem fio, ele tira poder do programador da aplicação pois uma exploração maior do poder computacional dos nodos da RSSF se torna difícil. Além disso, uma solução desse tipo não esconde a fronteira entre as RSSFs e o resto do sistema computacional.

Em contraste com essa aproximação, este trabalho integra RSSFs e o Grid de maneira transparente sem remover o poder do programador da aplicação estendendo o POP-C++ [4]. POP-C++ é uma linguagem de programação orientada a objetos para Grid, e também um sistema de suporte de tempo de execução capaz de suportar objetos paralelos e distribuídos em uma rede. Os objetivos específicos desta extensão são permitir que:

- Aplicações no Grid se comuniquem com RSSFs transparentemente: Escondendo-se toda a interação com a rede sob uma interface de chamada remota de métodos, todos detalhes da pilha de rede e do meio físico são invisíveis da aplicação;
- Uso concorrente das capacidades de sensoriamento das RSSFs por múltiplas aplicações do Grid: Permitindo-se que múltiplos objetos sejam executados em cada nodo e que cada objeto seja utilizado por múltiplas aplicações, o uso concorrente das RSSFs por múltiplas aplicações é possibilitado;
- Software de nodo das RSSFs independente de aplicação: Ao permitir que múltiplas aplicações usem um conjunto comum de métodos, a necessidade de re-programação da memória de programa dos nodos é minimizada e torna-se possível que novas aplicações sejam executadas após a instalação da RSSF.

A estrutura deste trabalho é a seguinte: Nos capítulos 2 e 3 os conceitos básicos do trabalho são apresentados, as Redes de Sensores sem Fio e o Grid, respectivamente; no capítulo 4 a linguagem de programação POP-C++ e seu sistema de suporte de tempo de execução são introduzidos; no capítulo 5 é detalhada a maneira na qual POP-C++ foi estendida para RSSFs; no capítulo 6 a implementação do POP-C++ para RSSFs é avaliada; o capítulo 7 apresenta trabalhos relacionados que compartilham o objetivo deste trabalho; finalmente, no capítulo 8 as conclusões são apresentadas.

## 2 *Redes de Sensores sem Fio*

O contínuo avanço dos processos de fabricação de hardware e a constante miniaturização dos microcontroladores e módulos de memória permite a criação de plataformas inteiras, incluindo interfaces analógicas, num só chip. A combinação do avanço tecnológico com a necessidade de extração de informação de uma área física extensa resultou na criação das Redes de Sensores sem Fio, compostas por nodos que:

- Têm dimensões e custo de fabricação reduzidos, para que possam ser distribuídos de forma pervasiva porém não-intrusiva no ambiente que devem monitorar;
- São capazes de operar e comunicar valores de sensoriamento sem uma conexão física com o resto da rede, o que acarreta no funcionamento a baterias e uma grande preocupação com gastos de energia;

Estas funcionalidades culminaram nos projetos de nodos para sensoriamentos encontrados hoje no mercado, que usam componentes de baixa potência, modulares e altamente configuráveis. Este capítulo discute os principais aspectos dos nodos das RSSFs, comunicação e sensoriamento, e também apresenta dois nodos disponíveis comercialmente; os motes Mica, da Universidade da Califórnia em Berkeley, e os BTNodes, da ETH em Zurique.

### 2.1 **Comunicação**

O transceptor de rádio é o componente que possui o maior custo energético nos nodos de sensoriamento. Para minimizar o seu efeito negativo na plataforma, é importante que eles sejam tão configuráveis quanto necessário; isto permite que os protocolos de acesso ao meio e roteamento que os utilizarão façam o menor número de transmissões possível.

A gama de parâmetros que podem ser utilizados para configurar a pilha de comunicação em um nodo de sensoriamento é muito grande; prova disso é a grande quantidade de protocolos [5–11] de controle de acesso ao meio e roteamento criados especialmente para RSSFs.

Todas classificações principais de protocolos de acesso ao meio tradicionais estão representadas na bibliografia para RSSFs: CSMA, divisão em Slots e TDMA [12]. Apesar dessas estratégias serem fundamentalmente diferentes quanto à sua maneira de permitir acesso de múltiplos nodos ao meio de comunicação, elas compartilham entre si os mesmos objetivos:

- Ligar o rádio em escuta o mínimo possível - A escuta desnecessária pode acontecer por duas razões:
  - Canal ocioso: Ligar o transceptor no modo de recepção quando não há tráfego no canal é danoso, pois no modo receptor é consumida mais energia que no modo desligado;
  - Pacotes com outro destino: O fato do meio ser compartilhado pode causar um nodo a receber pacotes que não são direcionados a ele.
- Evitar e tratar colisões: O meio compartilhado causa a possibilidade de colisões de transmissão entre dois nodos. As duas principais estratégias para este problema são:
  - Detecção de Colisões: Usando um mecanismo de garantia de entrega após os pacotes, os protocolos podem detectar colisões, e, se interessante, re-transmitir os pacotes perdidos;
  - Evitar Colisões: Os protocolos podem garantir a transmissão em tempos diferentes através de fatias de tempo, ou diminuir a quantidade de colisões através da troca de pacotes *Request-to-Send* e *Clear-to-Send*.

## 2.2 Sensoriamento

Os sensores são os dispositivos que permitem que as RSSFs extraiam informações do ambiente no qual elas são instaladas. As suas implementações são baseadas em um componente que, através de um sinal elétrico, permite a medição de alguma propriedade física do ambiente, como temperatura, luminosidade ou intensidade e direção de campos magnéticos.

A interface que o sensor disponibiliza para o microcontrolador do nodo pode ser digital, conectada com uma porta de entrada e saída, ou analógica, conectada com um dos conversores analógico-digital do controlador. No caso da interface digital, a lógica de conversão deve estar contida no próprio sensor, o que pode aumentar o seu custo consideravelmente.

## 2.3 Plataformas Comerciais

A pesquisa dedicada à área de RSSFs culminou na produção comercial de nodos de sensoriamento. Esta seção apresenta dois desses projetos, ambos os quais tiveram suas origens na academia.

### 2.3.1 Mica

A Universidade da Califórnia em Berkeley vem desenvolvendo uma série de nodos de sensoriamento, cuja geração atual é o Mica2 [13]. Esta geração usa um rádio de canal único Chipcon CC1000, que permite alta configurabilidade em software, um microcontrolador Atmel Atmega128 de 8 bits a 8MHz, com 4KB de memória principal e 128K de memória de programa.

O Mica2 tem dois modelos, o Mica2 e o Mica2 Dot (esquerda e direita na figura 2.1, respectivamente) ambos com projeto modular para permitir a utilização de diferentes módulos de sensoriamento conforme necessário.

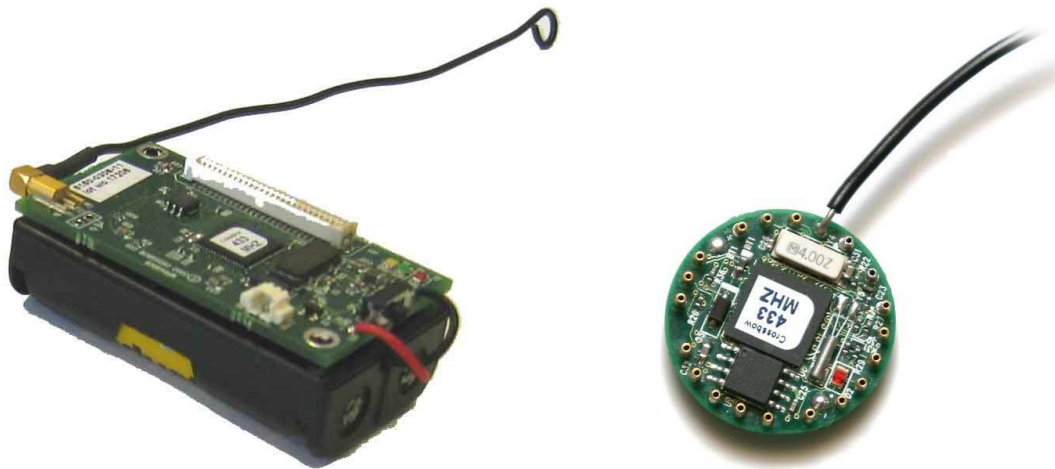


Figura 2.1: Nodos Mica2

### 2.3.2 BTnode

A plataforma BTnode [14] (figura 2.2), apesar de também utilizar o controlador Atmega128 do Mica2, utiliza um rádio Bluetooth de maior complexidade em adição ao CC1000 e estende as memórias RAM e de programa através de chips externos ao microcontrolador.

A tabela 2.1 compara os nodos Mica2 e os BTnode, e ilustra as escolhas de projeto quanto aos seus microcontroladores, quantidade de memória e interfaces de rede. Eles são represen-



Figura 2.2: Nodo BTnode

tativos da capacidade limitada de processamento e comunicação dos nodos de sensoriamento encontrados no mercado.

	Dimensões	Controlador	Mem. de Prog./RAM	Interface de Rede
Mica2	5x4x1.5cm	AVR 8MHz	128KB/4KB	Chipcon 916MHz, 40kbps
BTnode	6x4x1.5cm	AVR 8MHz	128KB/64KB	Bluetooth 2400MHz, 700kbps

Tabela 2.1: Comparação entre o Mica2 e o BTnode

## 3 *Grid*

O Grid é um modelo computacional que emergiu na década de 1990 para suprir a necessidade de computação de alta performance para aplicações científicas, utilizando computadores em rede para distribuir a carga de trabalho a ser realizada. No seu artigo seminal, Ian Foster [1] define o problema do Grid como “compartilhamento controlado e coordenado de recursos entre organizações virtuais dinâmicas e escaláveis”.

Este capítulo descreve as organizações virtuais e a arquitetura do Grid como definidas por Foster, e discute duas plataformas dedicadas à criação de Grids, o *toolkit* Globus e o *middleware* gLite.

### 3.1 Organizações Virtuais

Organizações Virtuais (OV) são compostas por participantes com interesse de compartilhar recursos para realizar alguma tarefa. Recursos são definidos não só como dados, mas também acesso direto a software, computadores e sensores, entre outros.

Limitações devem poder ser impostas no compartilhamento, como o tipo de operações permitidas num grupo de computadores. Para que isso seja possível, sistemas de autenticação e autorização consistentes entre os membros de uma OV devem ser implementados.

Dada a natureza dinâmica das OVs, também são necessárias maneiras de descobrir e descrever recursos compartilhados, para que novos membros de uma OV possam determinar que recursos podem acessar e quais políticas regem seu acesso.

### 3.2 Arquitetura do Grid

A arquitetura do Grid nasce da necessidade da interação entre as OVs; ela deve permitir a criação, manutenção e exploração do compartilhamento entre os participantes da OV. Ela foi definida em 5 camadas:

- *Fabric*: A camada mais baixa, de interface com recursos locais. Exemplos seriam armazenamento, poder computacional ou repositórios de código;
- *Conectividade*: A camada de conectividade define os protocolos de comunicação e autenticação usados nas transações de rede do Grid, e permitem a troca de dados entre recursos da camada *Fabric*;
- *Recurso*: Esta camada é responsável pelas operações de compartilhamento (por exemplo: iniciação, controle, pagamento, monitoração) sobre recursos individuais;
- *Coletivo*: Esta camada implementa operações sobre conjuntos de recursos, como serviços de diretório que permitem a descoberta de recursos;
- *Aplicações*: As aplicações operam como a última camada no ambiente de uma OV, usando os serviços providos pelas camadas inferiores.

Essa arquitetura permite que as OVs se organizem, e as aplicações operem em um ambiente capaz de suportar o compartilhamento de recursos de forma segura entre os participantes. A figura 3.1 mostra essa arquitetura de forma esquemática a relacionando com a arquitetura de protocolos da Internet. Enquanto as camadas mais baixas da arquitetura do Grid são representadas na Internet por protocolos pré-existentes da pilha TCP/IP, o fato da arquitetura da Internet extender-se até a aplicação permite que as camadas superiores do Grid sejam mapeadas a ela.

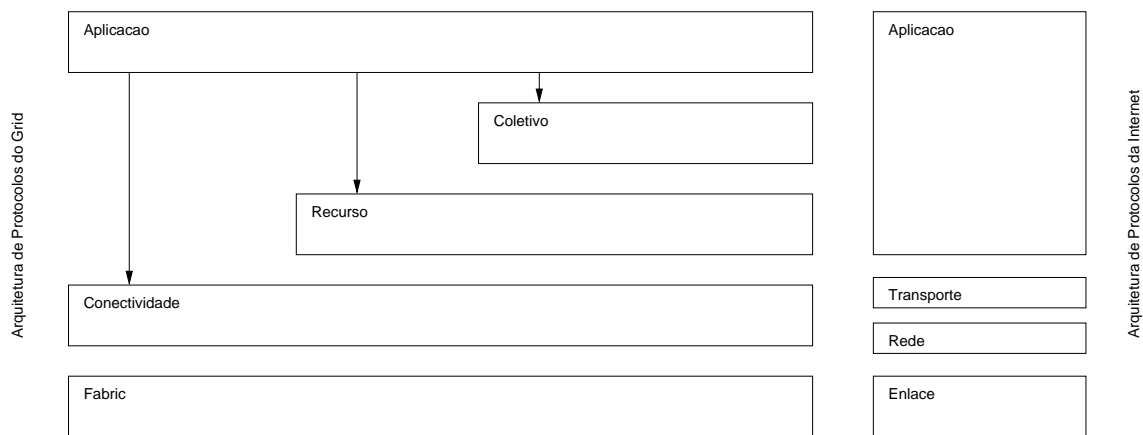


Figura 3.1: Mapeamento das camadas da arquitetura do Grid na arquitetura da Internet

### 3.3 Plataformas Comerciais

A atenção dedicada à pesquisa sobre o Grid e as tecnologias envolvidas nele levaram à criação de vários sistemas de suporte a Grids. Dois deles são discutidos nesta seção: o *toolkit*



Globus, e o *middleware* gLite.

### 3.3.1 Globus

O Globus [15] é um *toolkit* de software livre usado para a construção de Grids. Ele foi criado pela Globus Alliance, da qual Ian Foster faz parte. Ele possui componentes para suportar todas camadas determinadas na arquitetura anterior.

Para o Globus, a primeira camada (*Fabric*), é composta por componentes pré-existentes, como por exemplo um sistema de arquivos distribuído. Apenas no caso de comportamento insatisfatório pela parte destes componentes, eles são completados pelo *toolkit*. Por exemplo, o Globus implementa maneiras de um recurso desta camada se descrever caso ele não o faça; no caso do sistema de arquivos, isto incluiria uma função de consulta de espaço livre.

As funcionalidades da segunda camada, de Conectividade, são providas pela própria Internet; o Globus usa tecnologia pré-existente e bem-definida para comunicação, e implementa o *Globus Security Infrastructure* (GSI) para suprir necessidades de segurança. O GSI trata da autenticação, proteção da comunicação e autorização de acesso aos recursos.

A terceira camada, de Recurso, figura no Globus como múltiplas APIs para protocolos de informações sobre recursos, acesso e gerência de recursos e transferência de arquivos. O conjunto deles permite a realização das operações fundamentais sobre os recursos individuais compartilhados.

Finalmente, a camada de Coletivo é implementada por diversos serviços que utilizam as camadas de Recurso e Conectividade para prover as funcionalidades exemplificadas anteriormente. Um exemplo de tal implementação é o *Grid Information Index Service*, usado pelos protocolos da camada de recurso para construir um índice de informações sobre os recursos compartilhados na OV.

### 3.3.2 gLite

O gLite [16] é um *middleware* para computação em Grid criado no contexto do projeto *Enabling Grids for E-science* (EGEE), financiado pela Comissão Européia. Ele provê um *framework* para a construção de aplicações para Grid que utilizam o poder do processamento distribuído e recursos de armazenamento encontrados na Internet.

Os subsistemas do gLite são:

- Elemento de Computação: representa um recurso computacional, e tem como funcionalidade principal a gerência de tarefas;
- Gerência de Dados: trata do acesso e replicação dos dados das aplicações, e se apresenta como um sistema de arquivos global;
- Contabilidade: acumula informação sobre o uso de recursos no Grid pelos usuários e OVs. Esta contabilidade pode ser usada na detecção de abusos e para ser realizada a cobrança dos usuários de recursos;
- Logging: acompanha tarefas em maior granularidade, registrando eventos como submissão, atribuição a um elemento de computação, início de execução, entre outros;
- Informação e Monitoramento: gera informação sobre o Grid como um todo e as aplicações nele executadas. Internamente ele implementa uma arquitetura de produtor-consumidor para que entidades possam publicar e buscar informações;
- Descoberta de Serviços: provê mecanismos de localização de serviços não só para usuários mas também para outros serviços, possibilitando a sua composição. A sua implementação foi projetada para ser leve e de uso simples;
- Segurança: implementa modularmente funcionalidades de autenticação e autorização;
- Gerência de Carga de Trabalho: responsável pela distribuição e gerência de tarefas entre os recursos do Grid, de forma a permitir a execução eficiente das tarefas.

Pode-se traçar um paralelo entre a arquitetura do gLite e a definida por Foster; os subsistemas de Gerência de Carga de Trabalho, Descoberta de Serviços, Informação e Monitoramento e Logging implementam funcionalidades da camada de Coletivo. A camada de recurso está representada nos Elementos de Computação e Gerência de Dados, em conjunto com o subsistema de Contabilidade. Parte da camada de conectividade se encontra no subsistema de Segurança, junto com as tecnologias de Internet pré-existentes.

## 4 *POP-C++*

POP-C++ é uma extensão de C++ criada pelo GridGroup da University of Applied Sciences de Fribourg na Suíça. O seu objetivo é suportar objetos distribuídos e paralelos com requisitos específicos de processamento, memória e banda de rede. No modelo de objeto do POP-C++, objetos paralelos têm a habilidade de descrever seus requisitos de recurso em tempo de execução e são alocados em qualquer um dos nodos do Grid capazes de suportar sua execução. O processo de encontrar um nodo que satisfaça os requisitos é transparente para o programador. O POP-C++ também suporta semânticas especiais na invocação de métodos, apesar do que a sintaxe das invocações dos métodos não ser diferente entre invocações locais e remotas. Além disso, objetos paralelos são compartilháveis, isto é, referências para um objeto podem ser passadas em qualquer método, local ou remoto.

Neste modelo, o programador de aplicação usa objetos paralelos, instanciados de forma distribuída. A interação destes objetos com os outros da aplicação, para o programador, será semelhante à interação entre objetos tradicionais de um programa local. No POP-C++, os objetos remotos e paralelos têm todas as propriedades de um objeto tradicional mais as seguintes:

- Os objetos remotos e paralelos são compartilháveis, ou seja, referências para um objeto podem ser passadas em qualquer método, local ou remoto;
- Invocações em objetos paralelos e objetos tradicionais são sintaticamente idênticas, apesar das invocações em objetos paralelos suportarem semânticas adicionais, concorrente, sequencial e mutex;
- Objetos paralelos podem ser alocados em recursos remotos em espaços de endereçamento separados, e esta alocação é transparente para o usuário;
- Cada objeto paralelo tem a habilidade de descrever seus requisitos de recursos dinamicamente em seu tempo de vida.

## 4.1 Semânticas de Método

Apesar de sintaticamente as invocações dos métodos serem idênticas às de objetos sequenciais tradicionais, os objetos paralelos POP-C++ possuem várias opções de Semântica de Invocação a serem definidas pelo desenvolvedor, e podem ser classificadas em dois tipos: Semânticas de Interface e Semânticas de Objeto:

- **Semânticas de Interface:** As semânticas de interface regem em que momento da execução a chamada de método retorna:
  - Assíncrona: Na semântica assíncrona, a invocação retorna imediatamente após a chamada do método é enviada para o objeto, permitindo que o invocador continue execução concorrentemente com o método invocado;
  - Síncrona: Na semântica síncrona, o invocador espera a execução completa e valores de retorno (se existirem) do método. Esta semântica é análoga à execução de métodos em objetos sequenciais tradicionais.
- **Semânticas de Objeto:** As semânticas de objeto regem o nível de paralelismo que deve ser permitido na execução dos métodos no objeto invocado:
  - Sequencial: Na semântica sequencial, a execução dos métodos se dá em exclusão mútua, seguindo a ordem de chegada das invocações. Quando várias invocações de um método com semântica sequencial são feitas num mesmo objeto paralelo, elas serão tratadas de forma sequencial (ver figura 4.1). Apesar disso, invocações pré-existentes a métodos concorrentes continuarão a ser executadas normalmente. Métodos sequenciais garantem a consistência serializável de todas as invocações sequenciais no mesmo objeto.
  - Mutex: Invocação a métodos de semântica mutex são tratadas em completa exclusão mútua com todas as outras invocações no mesmo objeto. Uma invocação só é executada uma vez que todas as invocações anteriores sejam terminadas e é bloqueada até que isto aconteça (novamente, ver figura 4.1). As invocações Mutex são importantes para a sincronização de concorrência e assegurar a correção do estado dos dados compartilhados dentro do objeto paralelo.
  - Concorrente: Na semântica concorrente o método é executado concorrentemente (em uma nova thread dedicada à sua execução) a não ser no caso de algum método Mutex estar em execução ou esperando execução. Esta semântica de objeto permite

paralelismo num objeto e proporciona concorrência entre comunicação e processamento.

A figura 4.1 detalha a ordem de execução de métodos de valores semânticos diferentes num mesmo objeto. Dado um objeto chamado “objeto” que possui os métodos de semântica concorrente  $\text{Conc1}()$ ,  $\text{Conc2}()$  e  $\text{Conc3}()$ , os métodos de semântica sequencial  $\text{Seq1}()$  e  $\text{Seq2}()$  e o método de semântica mutex  $\text{Mutex1}()$ , a ordem de chegada de invocações de métodos mostrada acarreta na seguinte ordem de execução: O método  $\text{Seq1}()$  é executado concorrentemente com o método  $\text{Conc1}()$ , apesar de sua invocação ter chegado depois; isso se deve ao fato de nenhum outro método sequencial nem mutex estar sendo executado no momento. Depois disso, o método  $\text{Conc2}()$  é executado antes de  $\text{Seq2}()$ , apesar deste ter chegado antes. Isso se deve à execução de  $\text{Seq1}()$  ainda não ter terminado, o que é condição necessária para a execução de qualquer outro método sequencial. Quando a invocação de  $\text{Mutex1}()$  chega, ela deve esperar o término da execução de todos métodos correntes, e ao mesmo tempo bloqueia a execução de todos métodos cuja invocação chegar posteriormente, no caso, o  $\text{Conc3}()$ , até que sua execução seja completada.

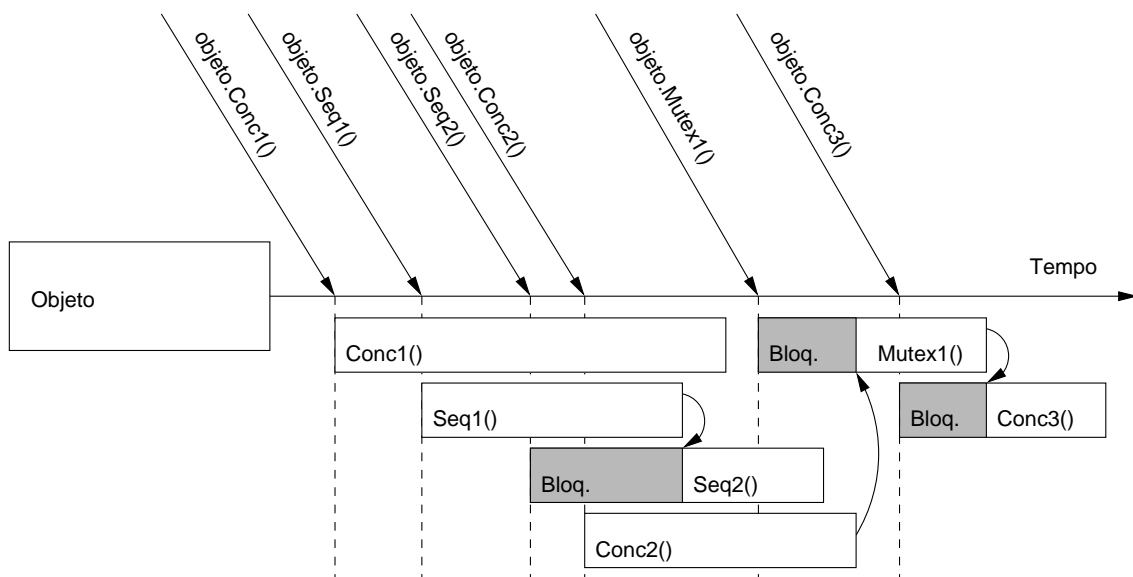


Figura 4.1: Linha de tempo detalhando a ordem de execução de métodos com semântica especial.

## 4.2 Descritor de Objeto

Para permitir que os objetos descrevam seus requisitos de recursos, cada objeto tem associado a si um Object Descriptor (OD), que possui campos para guardar os requisitos mínimos nos seguintes termos:

- Nome do Nodo: Este campo permite que o objeto escolha explicitamente em que nodo do Grid ele será hospedado;
- Poder de Processamento: Este campo permite que o objeto especifique em MFlops o poder de processamento mínimo para que a sua tarefa seja executada a tempo;
- Quantidade de Memória: Neste campo o objeto pode especificar em MB a quantidade de memória consumida por suas estruturas de dados;
- Largura da Banda de Rede: Neste campo o objeto define a largura de banda mínima que deve estar disponível entre ele e suas interfaces.

As descrições de requisitos mínimos do OD podem ser de dois tipos: estritos e não-estritos. Quando do tipo estrito, o objeto só poderá ser executado em um nodo que satisfaça completamente os requisitos descritos. Em contraste, os campos não-estritos abrem a seleção de nodos para uma gama maior, incluindo nodos que se encaixam apenas parcialmente nos requisitos descritos. Por exemplo, uma linha de descrição de requisitos pode ser da seguinte forma: “od.memory(50)”, que define o campo de memória estritamente num mínimo de 50 MB, ou “od.memory(50, 25)”, que define uma quantidade preferencial de 50 MB e o mínimo absoluto para 25 MB.

### 4.3 Extensão do C++

Para refletir as extensões do modelo de objeto, a linguagem C++ foi estendida para representá-las. As adições ao C++ são três: Declaração de Classes Paralelas, Descrições de Requisitos e Semânticas de Invocação.

- Declaração de Classe Paralela: Para diferenciar os objetos a serem distribuídas dos objetos de execução normal, a palavra chave “parclass” foi introduzida. Isso permite que o sistema de suporte de tempo de execução do POP-C++ dê suporte a distribuição e paralelismo nas classes paralelas sem incorrer em sobrecusto nas classes sequenciais tradicionais;
- Descrições de Requisitos: Usando um descritor associado ao Objeto, o desenvolvedor pode expressar requisitos de recursos na forma de um endereço de rede, o número de MFlops necessário, a quantidade de memória necessária e a banda de rede mínima entre o Objeto e suas Interfaces. No exemplo da figura 4.2, a classe paralela possui dois construtores; o primeiro toma como parâmetro o poder de processamento ideal (“wanted”) e

o poder de processamento mínimo (“minp”), o segundo toma como parâmetro o endereço de rede no qual executar os objetos paralelos desta classe (“machine”). A parametrização destes requisitos nos construtores permite a atribuição dinâmica de valores a eles, o que é interessante na implementação de programas distribuídos cujos conjuntos de dados de entrada, ou requisitos funcionais como limite de tempo para execução mudam dinamicamente;

- Semânticas de Invocação: As opções de Semântica de Invocação são definidas em tempo de compilação pelo desenvolvedor, e podem ser classificadas em dois tipos: Semânticas de Interface e Semânticas de Objeto:
  - A Semântica de Interface pode ser Síncrona ou Assíncrona; ela controla quando o método da Interface retorna. Pode-se ver na figura 4.2 que o `set()`, que por não retornar valores e se tratar de uma operação cujo término não é preciso aguardar, tem um valor semântico assíncrono, simbolizado pela palavra chave “`async`”. Já no método `get()`, está sub-entendida a semântica síncrona, simbolizada pela palavra chave opcional “`sync`”;
  - Semânticas de Objeto podem ser Mutex, Sequencial ou Concorrente. Na figura 4.2 temos exemplo das três: apenas uma invocação ao método `set()` será executada por vez (semântica sequencial, de palavra chave “`seq`”). Múltiplas chamadas ao `get()` serão executadas paralelamente, possivelmente em paralelo a uma chamada ao `set()` (semântica concorrente, palavra chave “`conc`”). O método `add()`, por sua vez, será executado em exclusão mútua em relação aos outros, por ter uma semântica mutex, palavra chave “`mutex`”.

## 4.4 Geração de Código

Para possibilitar a interação remota entre os objetos paralelos, três entidades são geradas de cada classe paralela POP-C++ que é implementada: a Interface, o Broker e o Objeto real. O parser POP-C++ toma a declaração das parclasses do programador e gera uma classe C++ para cada uma dessas entidades, processo ilustrado na figura 4.3. Depois que o código C++ é gerado, o compilador C++ para a arquitetura alvo (aqui exemplificado pelo GNU G++) gera código binário.

A figura 4.4 mostra a relação entre essas entidades em tempo de execução; a Interface fica em um nodo separado do Broker e do Objeto real, e dela originam as chamadas de função. A

```

parclass Integer {

    public :

        Integer(int wanted, int minp) @{ od.power(wanted, minp); };
        Integer(const char machine[256]) @{ od.url(machine); };

        seq async void set(int val);
        conc int get();
        mutex void add(Integer &other);

    private :

        int data;

};

```

Figura 4.2: Exemplo de declaração de uma classe paralela POP-C++

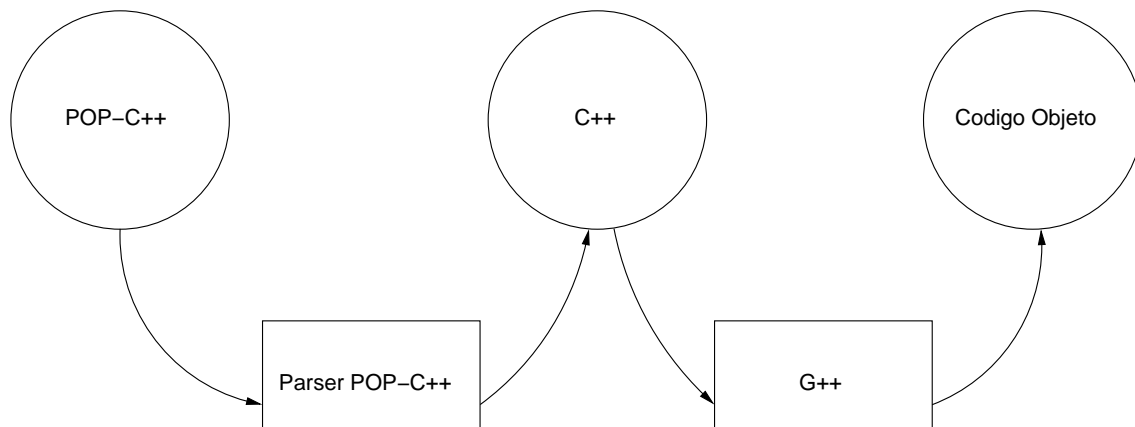


Figura 4.3: Cadeia de geração de código binário a partir de código POP-C++.

aplicação (que pode ser composta por outros objetos paralelos) invoca métodos na Interface da mesma forma que faria a um objeto sequencial tradicional localmente.

Estas chamadas de função são transmitidas pela rede com todos dados associados a ela, como qual objeto deve executá-la e os parâmetros da função, além de informação adicional para o próprio sistema de suporte de tempo de execução POP-C++.

Para substituir transparentemente o Objeto real na aplicação, a Interface compartilha a assinatura de todos seus métodos. As tarefas da Interface são empacotar os dados de chamada de método num buffer a ser transmitido pela rede, enviá-lo para o Broker, e esperar um buffer contendo os valores de retorno da chamada de função. Após a retirada dos dados desse buffer, o valor é retornado para o invocador.

O Broker é o correspondente da Interface do lado do invocado; ele recebe as chamadas



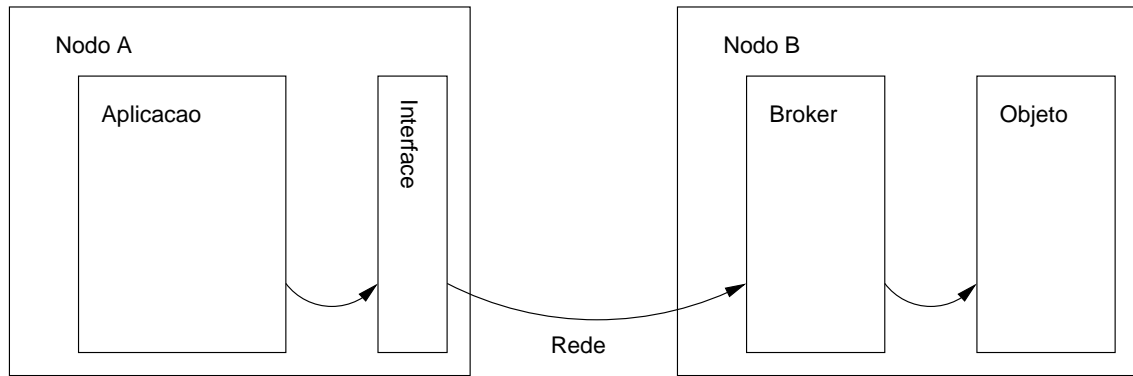


Figura 4.4: Diagrama de localização e interação da Interface, Broker e Objeto.

de método da rede invoca o método no Objeto real e então envia os valores de retorno para a Interface. O Objeto real consiste na implementação do usuário, com o código que deve ser distribuído.

O diagrama de sequência UML da figura 4.5 mostra este processo de forma simplificada numa invocação do método `get()` da parclass `Integer` vista anteriormente. A classe `Integer` mostrada ali é na verdade a Interface gerada pelo parser; ela toma o nome da parclass para fins de transparência ao programador. As classes `Integer__parocobjBroker` e `Integer__parocobj` são o Broker gerado e o Objeto real, respectivamente. Os buffers usados pela Interface e pelo Broker são encarregados de guardar os dados referentes às chamadas de métodos e também usar as funcionalidades de comunicação das bibliotecas POP-C++ para enviar e receber invocações pela rede.

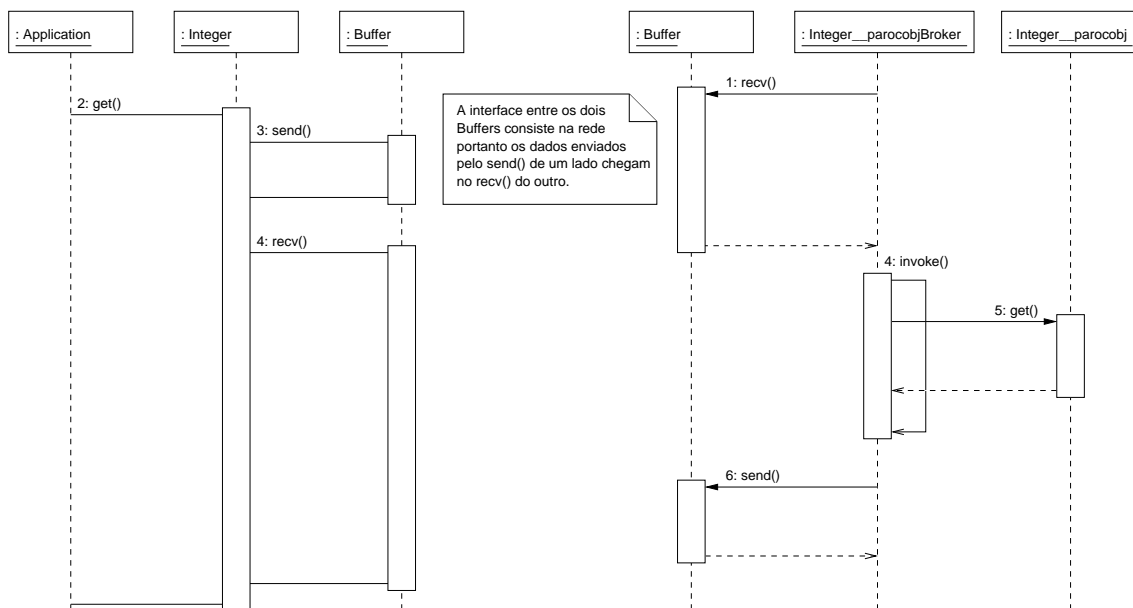


Figura 4.5: Diagrama de Sequência de uma Invocação Remota de Método

Seguindo a ordem do diagrama, a primeira ação tomada é da parte do Broker, que manda o seu buffer receber uma invocação da rede (1). O `get()` invocado pela aplicação na Interface (2) acarreta em um `send()` no buffer do seu lado (3), que envia a invocação pela rede. Assim que o buffer do lado do Broker recebe a invocação, o método de recepção retorna, o Broker analisa os dados contidos no buffer (4) e realiza a chamada ao método `get()` do objeto real (5). Paralelamente a isso, o buffer do lado da interface entra em modo de recepção (4), esperando o retorno do método pela rede (6). Assim que o retorno é recebido da rede, o método de recepção do buffer da Interface retorna, a Interface analisa os conteúdos do buffer e retorna o valor para a aplicação, que durante esse procedimento todo acredita ter invocado um simples método `get()` local.

## 4.5 Sistema de Suporte de Tempo de Execução

A figura 4.6 mostra o diagrama de classes geral do sistema de suporte POP-C++, com as classes geradas (`Integer`, `Integer__parocobjBroker` e `Integer__parocobj`) estendendo as classes do sistema (`Interface`, `Broker` e `Object`, respectivamente). Ela também mostra a interação da Interface com o Job Manager, que o utiliza para fazer a alocação dinâmica de recursos, e da Interface e do Broker com o Buffer (e indiretamente, `Combox`) para realizar a comunicação pela rede.

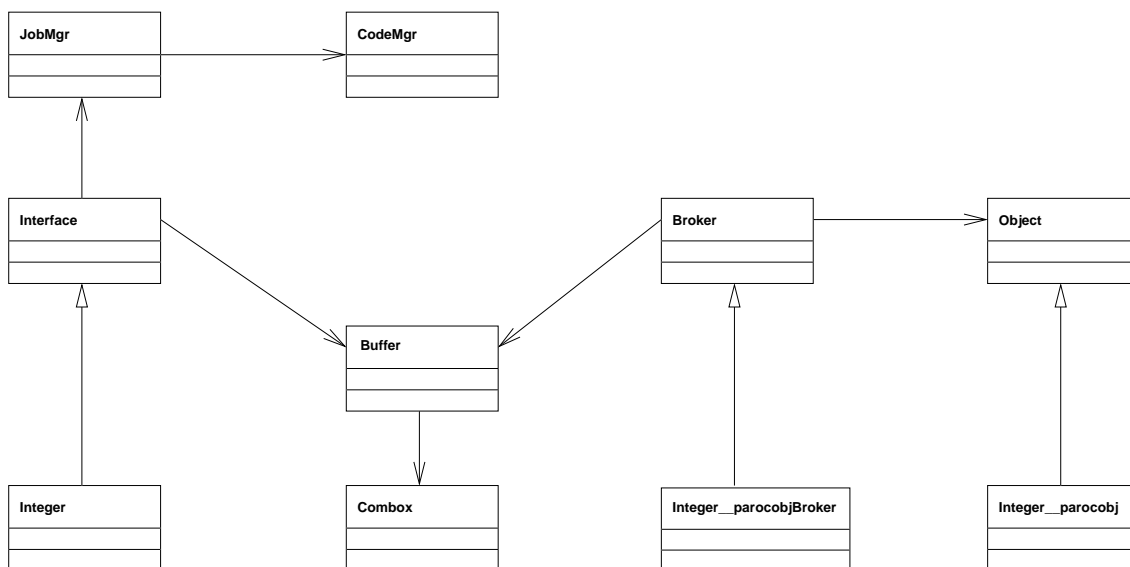


Figura 4.6: Diagrama de Classes do Sistema de Suporte de Tempo de Execução

### 4.5.1 Job Manager

Além da interação entre objetos distribuídos, o POP-C++ é capaz de realizar a alocação dinâmica de recursos dentro do Grid, através do Job Manager. O Job Manager é um objeto paralelo POP-C++, executado em todas máquinas que devem receber os objetos da aplicação. As funções do Job Manager são:

- Avaliar a capacidade de processamento, memória e rede do nodo no qual ele se encontra para fins de comparação futura;
- Conectar-se com os outros Job Managers conhecidos do Grid;
- Receber pedidos locais e remotos de execução de Objetos e tomando a decisão de:
  - Aceitar aqueles que tem requisitos menores do que a capacidade local;
  - Passar para os Job Managers em nodos vizinhos aqueles que requerem mais do que a capacidade local em algum dos parâmetros possíveis.

Assim que a aplicação instancia a Interface para um Objeto, o Objeto real para qual ela vai enviar as chamadas de método ainda não foi instanciado. Isto acarreta no registro de um pedido de alocação com o Job Manager local que, usando o OD associado ao Objeto a ser instanciado, avalia a aptidão do nodo no qual está sendo executado. Caso esteja apto, ele cria um novo processo com o Objeto; caso não esteja, ele encaminha o pedido para outro Job Manager que faz o mesmo. Este processo continua até que um Job Manager apto seja encontrado ou a árvore de Job Managers seja esgotada.

Vale citar que este processo dos Job Managers organizados em árvore executando chamadas POP-C++ recursivamente configura um algoritmo distribuído de alocação de recursos.

### 4.5.2 Code Manager

Para suportar Grids heterogêneos, o sistema de suporte de tempo de execução do POP-C++ faz o uso do Code Manager. Esta entidade, programada em POP-C++ como o Job Manager, é responsável por manter listas das arquiteturas e sistemas operacionais suportadas por cada Objeto. Desta forma, quando um nodo capaz de executar um determinado Objeto é encontrado, um binário compatível com aquele nodo é carregado da rede e um processo é instanciado.

O processo de compilação do binário para cada uma das arquiteturas a ser suportada fica a cargo do programador da aplicação, mas para facilitar o processo de distribuição num Grid de

grande escala, o Code Manager suporta protocolos como HTTP e FTP para a transferência dos arquivos binários dos Objetos.

### 4.5.3 Comboxes

Devido à grande heterogeneidade das tecnologias de interconexão dos Grids, suporte a múltiplas delas com a possibilidade de integração posterior de suporte a novas tecnologias é uma necessidade para um sistema como o POP-C++. Para que isso se torne possível, os buffers que tratam da codificação e transmissão dos dados usam objetos Combox para realizar qualquer transmissão na rede.

Cada tecnologia de interconexão (como por exemplo, Gigabit Ethernet, Myrinet ou Infini-band) tem a sua implementação de Combox, e a decisão de qual usar é feita numa negociação com o Job Manager que instanciou o Objeto. Após a instanciação do Objeto, a Interface deve criar uma Combox do tipo correto e tentar comunicar-se com o Broker no nodo onde ele foi instanciado.

## 5 *Estendendo POP-C++ para RSSFs*

Para prover um modelo uniforme com o qual programar uma aplicação do Grid que utilize RSSFs, este trabalho estende o modelo POP-C++ para RSSFs. Isto significa que não só o programador deve ser capaz de instanciar Interfaces em nodos de sensores para Objetos em execução em outros nodos, mas também instanciar Interfaces para estes objetos a partir do Grid. Não deve existir diferença entre chamadas de métodos remotas onde o invocador e invocado estejam ambos no Grid e chamadas realizadas do Grid para as RSSFs.

A figura 5.1 ilustra a declaração de um Objeto que é executado na RSSF e recebe chamadas de função do Grid. Este Objeto tem métodos para realizar leituras do sensor de temperatura e receber um valor a ser mostrado nos LEDs do nodo da RSSF. Qualquer nodo do Grid deve poder instanciar uma Interface para este Objeto e transparentemente invocar seus métodos.

### 5.1 **Sistema de suporte de tempo de execução POP-C++ para RSSFs**

Para permitir que os objetos paralelos POP-C++ programados para o Grid pudessem ser executados em RSSFs, todo código de suporte teve de ser adaptado para os nodos de sensoriamento. O esforço inicial de adaptação foi direcionado para dentro das RSSFs, ou seja, para possibilitar que Interfaces, Brokers e Objetos paralelos fossem gerados e executados nos nodos de sensoriamento. Posteriormente, a ligação entre o POP-C++ pré-existente, para Grids, foi realizada.

A figura 5.2 mostra a relação entre Interface, Broker e Objeto sendo executados nos nodos de sensoriamento; pode-se fazer uma relação direta desta disposição com a mostrada no capítulo que descreve o POP-C++, onde a Interface se encontra em um nodo e Broker e Objeto se encontram em outro, e entre os dois se encontra a rede sobre a qual eles se comunicam.

```
parclass Sensor {  
  
    public:  
  
        Sensor(string machine, uint16 n) @od.url(machine);};  
        seq async void set(unsigned char val);  
        conc sync unsigned char get();  
        uint16 getNode();  
  
    private:  
  
        uint16 node;  
        unsigned char data;  
        classuid(1);  
  
};  
  
Sensor * s = new Sensor("wsnproxy.eif.ch", 99);
```

Figura 5.1: Classe POP-C++ SensorNode Básica

### 5.1.1 Estratégia de Adaptação para RSSFs

Para manter compatibilidade do código gerado pelo parser POP-C++ no novo sistema de suporte para RSSFs, a sua arquitetura deveria ser fundamentalmente a mesma, implementada de forma o mais leve possível devido aos recursos escassos encontrados em nodos de sensoramento.

O primeiro passo da adaptação foi analisar o sistema de suporte existente, identificando sua estrutura principal, e encontrando pontos onde a implementação de funcionalidades poderia ser simplificada ou funcionalidades que não são interessantes nas RSSFs.

### 5.1.2 Diferenças fundamentais entre os nodos do Grid e das RSSFs

Devido à natureza de baixos recursos e homogênea dos nodos de RSSF, a implementação para RSSF do sistema de suporte de tempo de execução do POP-C++ tem de ocupar tão pouca memória principal e de programa quanto possível. Isto obrigou a realização de algumas concessões na implementação:

#### Protocolo de Comunicação Constante

Quanto num ambiente de Grid, é não só interessante mas necessário suportar múltiplos protocolos de comunicação, com a possibilidade de escolher entre eles em tempo de execução.

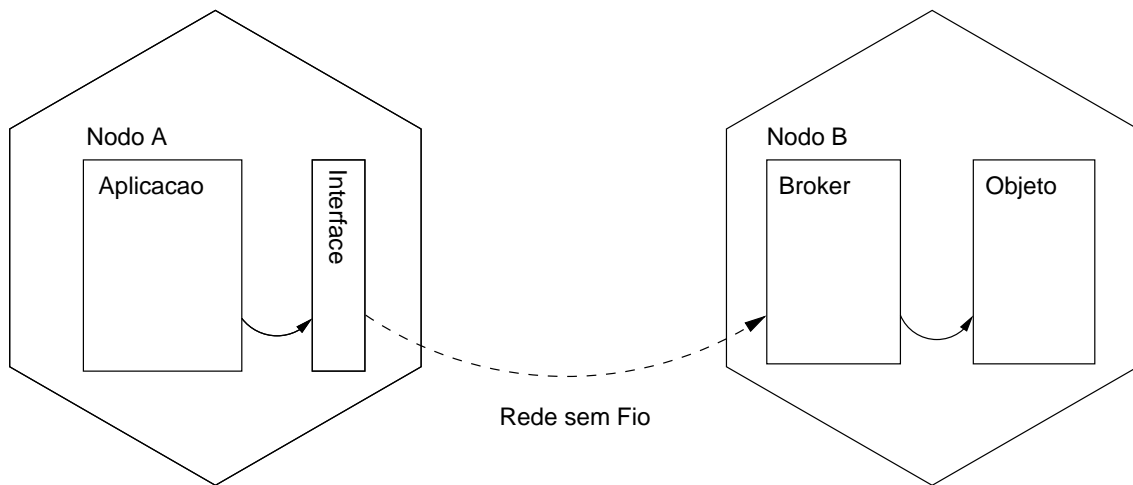


Figura 5.2: Diagrama de localização e interação dos componentes em RSSF.

No POP-C++ original estas funcionalidades são implementadas pela Combox. Apesar disso, nas RSSFs o protocolo de comunicação tem grande probabilidade de ser constante e global, devido ao hardware homogêneo e protocolos bastante específicos que devem ser empregados. Por isso, a implementação deste trabalho não suporta explicitamente múltiplos protocolos de comunicação em tempo de execução.

Esta funcionalidade, caso venha a ser interessante, pode ser atingida através de uma pilha de comunicação mais avançada no sistema operacional usado para suportar o POP-C++ para RSSFs.

Na implementação do sistema de suporte para RSSFs, esta simplificação acarretou na remoção das Combox por completo, que figuram no código apenas por motivos de compatibilidade. Para refletir esta mudança, os Buffers agora usam as funcionalidades de comunicação do sistema operacional diretamente.

### **Alocação de recursos estática**

No Grid o custo de um processo de alocação dinâmica de recursos é apenas fazer a carga e execução de um binário da rede. Para isso, o conjunto Job Manager e Code Manager mantém listas de nodos e arquivos de código com suas respectivas arquiteturas. Na instanciação de uma Interface, o algoritmo de alocação dinâmica de recursos é iniciado, gerando várias trocas de mensagens e culminando na execução do Objeto apropriado.

Nas RSSFs, não só o custo energético desta troca de mensagens seria altíssimo, mas a reescrita de memória de programa é um procedimento mais caro ainda [17]. Portanto, na versão para RSSFs, foi feita a escolha de não suportar a alocação dinâmica de recursos. Isto permite

que a imagem a ser gravada na memória de programa dos nodos de sensoriamento seja constante através da vida útil da RSSF.

As consequências principais desta simplificação são duas:

- O Object Descriptor perde a maior parte de sua funcionalidade, e as Interfaces estão limitadas à alternativa de instanciação através de endereço de rede;
- Com a etapa de criação de um novo processo pelo Job Manager removida, os Brokers devem ser iniciados pelo próprio sistema operacional no tempo de inicialização dos nodos.

Para diminuir a necessidade de reprogramação, os programadores são encorajados a utilizarem funções independentes de aplicação para possibilitar que novas aplicações sejam executadas na RSSF após a sua instalação.

### **Paralelismo limitado**

Devido à quantidade muito pequena de memória principal encontrada em nodos de RSSF, a quantidade de threads concorrentes que podem ser executadas em um nodo também é bastante pequena. Isto significa que a implementação de POP-C++ para RSSF suporta uma quantidade limitada de chamadas de método concorrentes, e que chamadas em excesso terão de ser postas em uma fila de espera, não importando sua semântica.

Esta limitação também é encontrada no POP-C++ original, porém pelo fato dos nodos do Grid normalmente terem quantidade de memória RAM medida nas centenas de megabytes, raramente vem a ser um problema.

### **5.1.3 Arquitetura do sistema de suporte adaptado**

Após as simplificações, a arquitetura de software do sistema de suporte de tempo de execução para RSSFs tomou a forma mostrada pelo diagrama na figura 5.3. Sem o Job Manager, Comboxes e com um Buffer simplificado, a implementação foi pequena o suficiente para permitir a execução dos Objetos paralelos POP-C++ em nodos de sensoriamento.

## **5.2 Interação entre o Grid e a RSSF**

Uma vez que um sistema de suporte de tempo de execução POP-C++ tenha sido implementado nas RSSFs, o segundo passo foi permitir a instanciação de Interfaces de dentro do Grid para



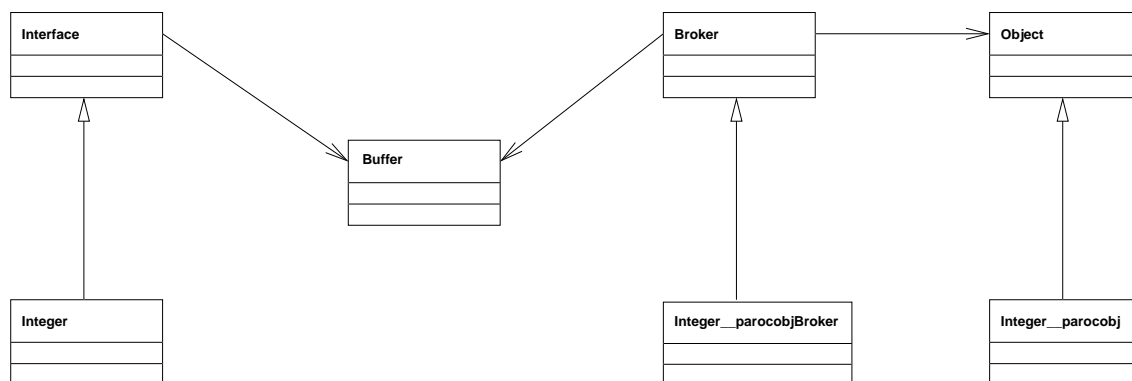


Figura 5.3: Diagrama de Classes do Sistema de Suporte de Tempo de Execução para RSSFs

Objetos em execução nas RSSFs. Para isso, foi expandida a metodologia de endereçamento já encontrada no POP-C++ original, além do desenvolvimento de um Broker de Proxy, cuja função é garantir a transparência da interação para a Interface.

A figura 5.4 mostra a disposição tradicional dos nodos do Grid em relação às RSSFs, e a relação entre Interface, Broker e Objeto. Esta figura também mostra que são necessários nodos específicos no Grid capazes de comunicar-se com a RSSF. Isso se deve ao fato que, em geral, é necessário hardware específico para a comunicação entre um nodo do Grid e um nodo da RSSF. Estes nodos, munidos do transceptor encontrado nos nodos de sensoriamento, devem então fazer a ponte entre o Grid, conectado com alguma tecnologia de maior velocidade como Ethernet, e a RSSF, usando rádio-frequência com protocolos de acesso ao meio cujo principal objetivo é economizar energia. Vale notar que podem existir mais de um ponto de contato entre o Grid e cada RSSF.

Na figura, o traço entre a Interface e o Broker é direta, não levando em conta o caminho que os pacotes de invocação de método devem realmente seguir. Realizar esta ponte trouxe dois obstáculos principais:

- Como endereçar nodos individualmente nas RSSFs, sem estender o POP-C++ existente;
- Como permitir que a Interface, sendo executada no sistema de suporte POP-C++ original, interagisse transparentemente com o Objeto, sendo executada no sistema de suporte POP-C++ para RSSFs.

A solução para os dois problemas foi o Broker de Proxy; não só ele permitiria desfrutar do endereçamento existente no POP-C++ tradicional, mas também serviria para criar a ilusão para a Interface de que ela estivesse interagindo com um Objeto normal do Grid.

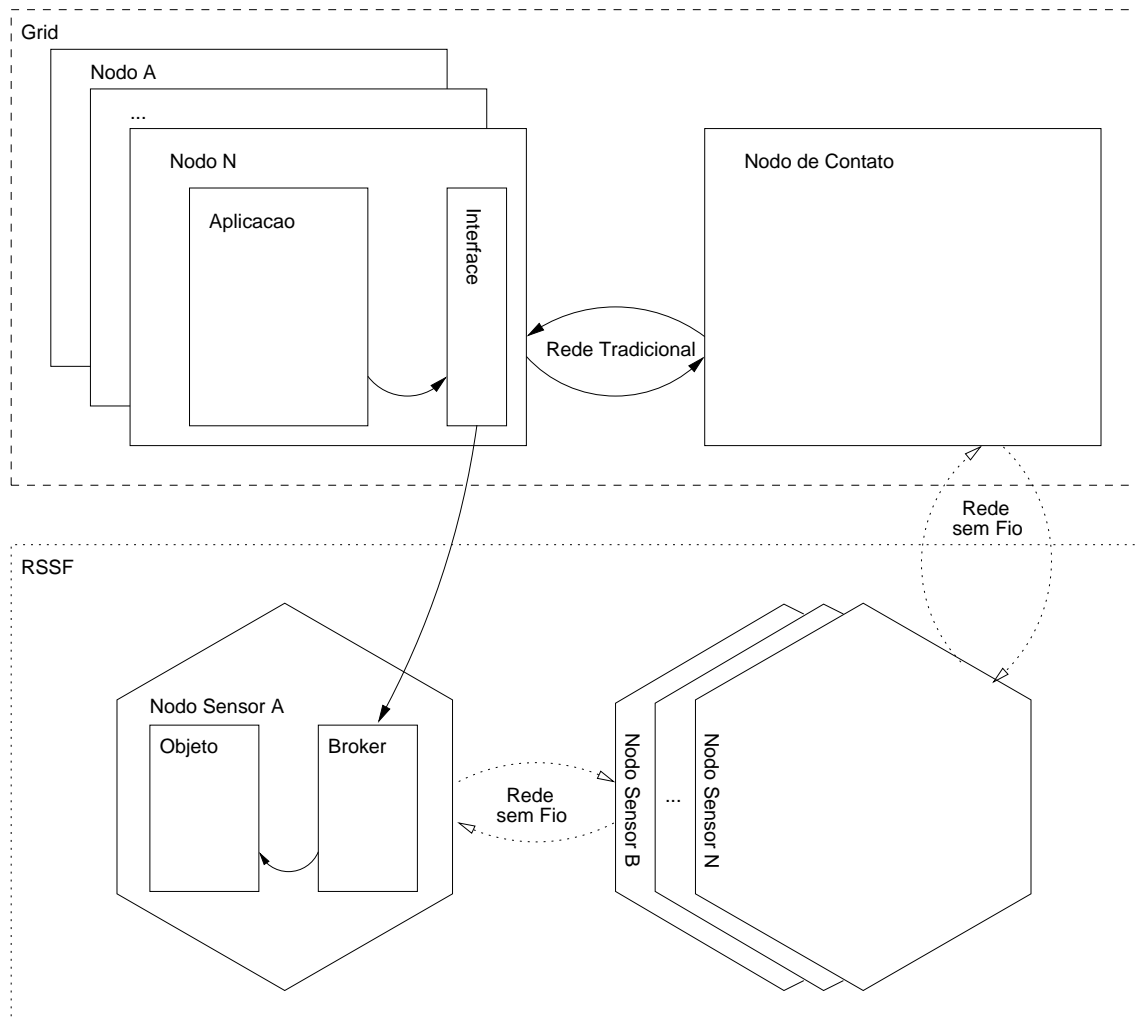


Figura 5.4: Interação do Grid com uma RSSF.

### 5.2.1 Endereçamento

Para permitir acesso direto a cada nodo sensor individualmente, este trabalho estendeu o método de endereçamento dos objetos POP-C++. Como mencionado no capítulo de apresentação do POP-C++, existe a possibilidade de Interfaces serem instanciadas com um parâmetro de endereço de rede, forçando o Objeto a ser alocado no nodo deste endereço. Foi feita a extensão deste método para os nodos da RSSF, usando dois endereços:

- **Endereço 1 - Ponto de contato entre o Grid e a RSSF:** Toda RSSF deve ter pelo menos um ponto de contato com o Grid. Este ponto de contato deve ser capaz de comunicar-se tanto no protocolo usado pelo Grid quanto pelo protocolo usado na RSSF, portanto provavelmente ele deve possuir hardware específico como o transceiver de rádio encontrado nos nodos sensores. Ao receber esse parâmetro, torna-se possível instanciar um Broker de Proxy no endereço do Grid apropriado, criando a ponte lógica que encaminha as chamadas de método direcionadas para a RSSF.
- **Endereço 2 - Nodo da RSSF:** Este endereço permite que o Broker de Proxy direcione as chamadas de método para o nodo sensor correto. O seu formato é deixado em aberto para que métodos de endereçamento diferentes sejam utilizados para diferentes RSSFs.

Na última linha da figura 5.1 pode-se ver a instanciação de um Objeto paralelo usando dois endereços. O primeiro, “wsnproxy.eif.ch” é o endereço da máquina que faz contato entre o Grid e a RSSF. O segundo, “99” é o endereço do nodo ao qual a Interface enviará suas chamadas de método.

Com esse artifício, foi possível a integração de duas tecnologias de comunicação completamente diferentes, tanto no seu meio físico, quanto nos seus protocolos e meios de endereçamento, de uma forma pouco invasiva no código da aplicação.

### 5.2.2 Broker de Proxy

Para permitir que chamadas de métodos sejam encaminhadas para os nodos da RSSF, um objeto Broker especial foi criado. Este é um Broker genérico (não específico para uma parclass) que recebe chamadas de método do Grid como se ele fosse o Broker real do Objeto, e então as encaminha para o nodo da RSSF. Quando o nodo da RSSF retorna da chamada de método, este Broker encaminha o valor de retorno para o invocador original. Isto cria o efeito de transparência para as Interfaces desse Objeto; para elas, as chamadas de métodos nunca deixam o Grid.

Apesar do Broker de Proxy não ser específico para parclass, antes de propagar os pedidos na rede sem fio ou os valores de retorno no Grid ele precisa realizar operações de adaptação nos pedidos de invocação para que o sistema de suporte de RSSF possa recebê-los. A principal adaptação é do cabeçalho, onde o Broker de Proxy reduz o tamanho dos campos de Objeto, Método e Semânticas utilizadas no POP-C++ original.

A figura 5.5 mostra o Grid conectado a uma RSSF através de Brokers de Proxy. Comparando esta com a figura 5.4, pode-se ver que agora as redes a serem utilizadas são duas, e há a necessidade de uma entidade lógica para fazer a interface entre as duas pontas.

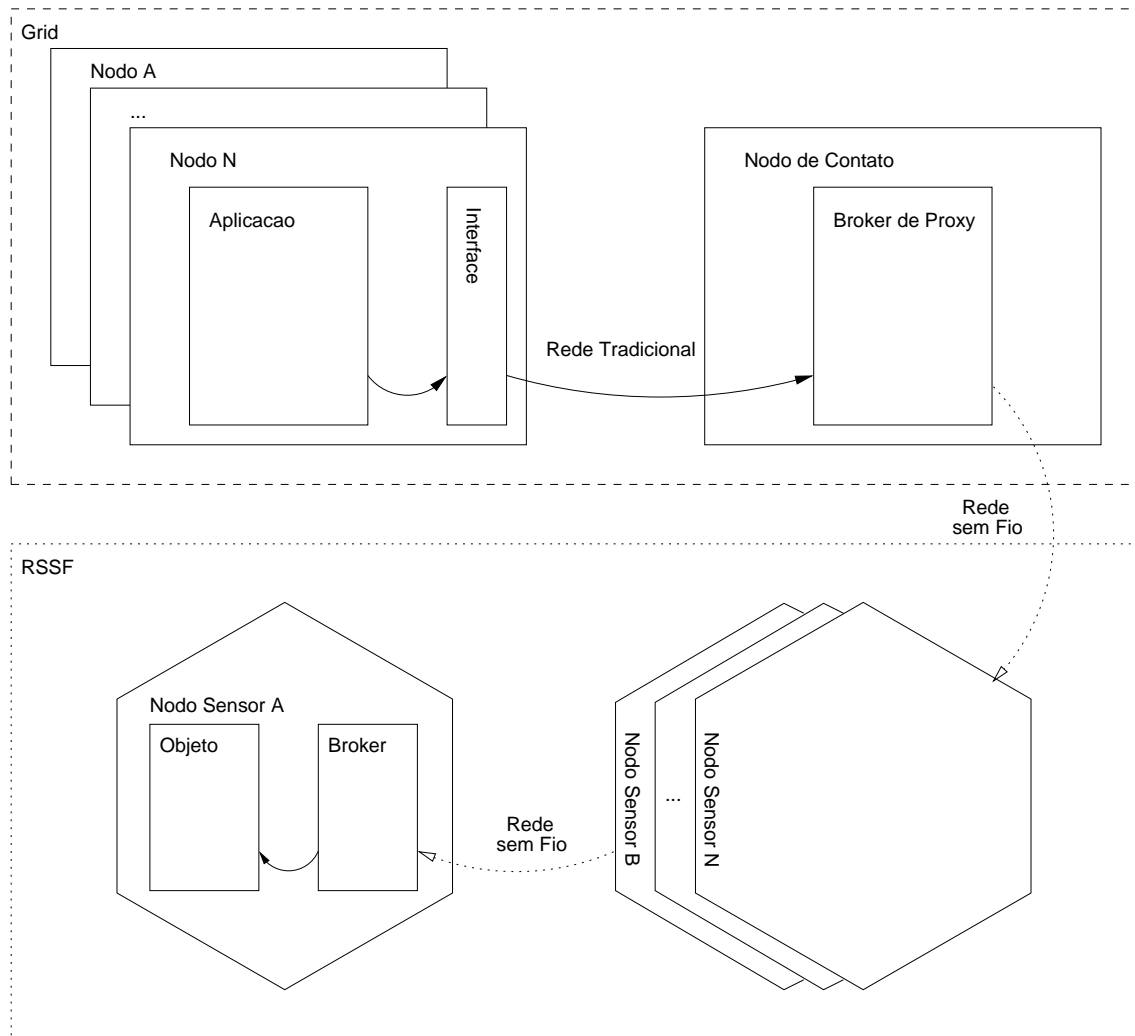


Figura 5.5: Interação do Grid com uma RSSF, com o Broker de Proxy.

Para instanciar o Broker de Proxy no nó correto, o sistema de suporte original usa o primeiro endereço, o que é incluído no Object Descriptor. A instanciação da Interface para um Objeto a ser executado no Grid ocasiona então duas instanciações remotas:

- O Broker de Proxy no nó do Grid de endereço 1;

- O Objeto paralelo no nodo de sensoriamento de endereço 2.

O primeira é realizada pelo Job Manager e Code Manager, que realizam a carga e iniciam o processo com o código do Broker de Proxy, e a segunda é feita através do Broker real, pré-existente no nodo de sensoriamento. Para a Interface e para o Objeto paralelo, toda interação é realizada com o Broker de Proxy.

### 5.2.3 Adaptador Grid-RSSF

Como mencionado anteriormente, os nodos que servirão de hóspede para os Brokers de Proxy devem ser capazes de comunicar-se no mesmo meio e protocolo das RSSF. Para fins de prova de conceito e teste, neste trabalho foi implementado um adaptador Grid-RSSF na forma de um nodo de sensoriamento dedicado, que faz a propagação no meio sem fio de toda invocação de método recebida pela sua porta serial, e envia pela porta serial as respostas dessas invocações. Esta alternativa a uma peça de hardware mais fortemente acoplada ao nodo do Grid (como uma placa PCI, por exemplo) que facilita bastante a replicação do hardware de rede e da pilha de comunicação usada na RSSF.

## 5.3 Processo de Geração de Código Objeto

No capítulo de descrição do POP-C++, foi mostrada a cadeia de ferramentas e a ordem de execução delas para chegar do código original POP-C++ no código binário a ser executado nos nodos do Grid. Para gerar o código para os nodos de RSSF, entretanto, este processo teve de ser modificado.

A reutilização do parser POP-C++ original se mostrou a escolha lógica, por duas razões:

- Compatibilidade com o sistema de suporte original: Usar o mesmo parser assegura que o código escrito para o sistema de suporte de tempo de execução para RSSFs fosse compatível com o original e vice-versa, evitando a criação de uma “segunda” linguagem POP-C++;
- Reuso de Código: O parser original já tinha se mostrado funcional e testado em larga escala. Um parser novo aumentaria bastante a complexidade do trabalho, apesar de permitir que a implementação do sistema de suporte para RSSFs fosse mais limpa, com menos código de compatibilidade.

Para gerar as três classes C++ para as parclasses implementadas pelo programador da aplicação, o parser POP-C++ analisa o código após a passagem do pré-processador C (no caso do Linux, por exemplo, o GNU CPP). Isso acarreta num arquivo C++ de saída que inclui todos os cabeçalhos usados não só pelo código do usuário, mas também pelo sistema de suporte POP-C++ que também é anexado a este código.

Como o sistema de suporte de tempo de execução para os nodos RSSF será implementado em um sistema operacional especial e deve ser o mais pequeno possível, a solução encontrada foi a extração das três classes geradas deste arquivo. Desta forma, pode-se incluir os cabeçalhos do novo sistema de suporte de tempo de execução e do novo sistema operacional, e, após a re-execução do pré-processador C, compilá-lo para a nova plataforma alvo, usando os compiladores da arquitetura usada nos nodos de sensoriamento.

A figura 5.6 mostra o novo processo de geração de código binário. Até a obtenção do código C++, o processo é idêntico à geração do código para nodos do Grid. Porém após disso, é usado uma ferramenta de expressões regulares para realizar a extração de código do arquivo (no exemplo, o GNU Sed). Este código extraído em conjunto com a re-implementação do sistema de suporte para RSSFs e o sistema operacional são dados como entrada no compilador C++, que gera a imagem a ser gravada na memória de programa dos nodos de sensoriamento.

Apesar da extração do código dos arquivos C++, o sistema de suporte de tempo de execução para RSSFs ainda precisou contar com artifícios de compatibilidade. Vários dos métodos invocados pelo código gerado pelo parser têm implementações vazias, alguns dos parâmetros e atributos originais das classes são ignorados na nova implementação, e outros tiveram sua implementação simplificada. Um exemplo de método vazio é o método `allocate()`, da classe `Interface`, que na implementação original inicia o processo de interação com o `Job Manager` para a procura de um nodo capaz de executar o Objeto paralelo. Outro exemplo é a classe `Object Descriptor`, que na implementação original precisa armazenar diversos requisitos de performance, na implementação para RSSFs trata-se apenas de um valor “int” que simboliza o nodo onde o Objeto deve ser executado. Ambas simplificações usadas como exemplo são fruto da ausência de um sistema de alocação dinâmica de recursos na versão para RSSFs do sistema de suporte de tempo de execução.

A alternativa a adicionar estes artifícios de compatibilidade seria corrigir o código C++ gerado pelo parser, porém isto acarretaria em ainda mais uma ferramenta complexa para fazer diversas operações no código original. Ao escolher implementar este código esta ferramenta se torna desnecessária, e como todos artifícios são resolvidos em tempo de compilação, não causam nenhum sobrecusto de memória ou processamento.

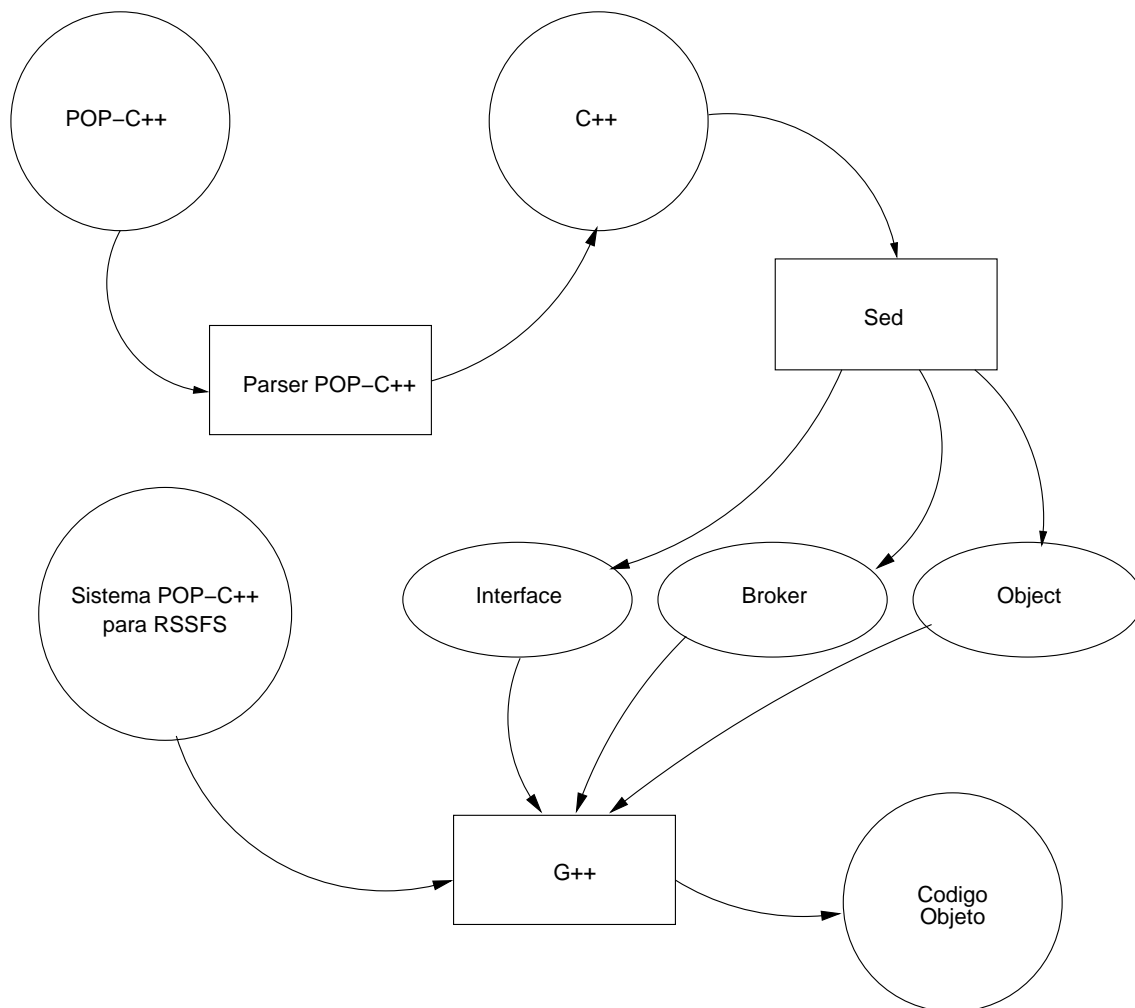


Figura 5.6: Processo de geração de código binário para POP-C++ em RSSFs

## 5.4 Desafios de Projeto

Nesta seção são discutidos os novos desafios de projeto introduzidos pelas RSSFs, o ambiente novo no qual POP-C++ trabalharia.

### 5.4.1 Carga na Rede

- **Sobrecusto de Dados da Carga Útil:** Para transportar os dados necessários para realizar a chamada de função, campos adicionais são incluídos nos pacotes transmitidos pela rede. Este é um sobrecusto que o POP-C++ original também tem, porém ele tem um efeito negativo maior na implementação para RSSFs devido ao alto custo de transmissão de dados pelo rádio. Para minimizar este problema, o tamanho de cada um destes campos foi diminuído dos seus valores originais de 32 bits, na expectativa que os nodos de sensoriamento executem um número menor de Objetos e que seus Objetos tenham um número menor de métodos.
- **Sobrecusto de Troca de Pacotes:** Enquanto a implementação original do sistema de suporte POP-C++ usa pacotes de ACK adicionais, na implementação para RSSFs qualquer correção de erros de rede ou controle de fluxo são deixados para a pilha de comunicação do sistema operacional; desta forma, os protocolos de controle de acesso ao meio e roteamento podem tratar da comunicação da melhor maneira possível sem interferência da aplicação.

### 5.4.2 Escalabilidade

Se todas chamadas de método provenientes do Grid para a RSSF forem roteadas através de um único nodo do Grid, a falha deste nodo causaria todos Objetos executados na RSSF a se tornarem incomunicáveis. Sob cargas muito pesadas ele também pode se tornar um gargalo da rede, causando alta contenção no nível físico. Para evitar este problema são permitidos pontos múltiplos de contato entre o Grid e a RSSF simultaneamente, idealmente fisicamente separados para que todos possam transmitir simultaneamente. Quando um dos pontos de contato falha, as aplicações podem trocar para um outro e continuar a comunicação.

### 5.4.3 Segurança/QoS

Os problemas de Segurança e QoS em RSSFs ainda são relativamente inexplorados, mas a atual pesquisa nestas áreas propõem soluções no nível de rede, especialmente na camada



de roteamento. Existem grupos trabalhando em métodos para proteger RSSFs de ataques de Denial of Service [18], garantir confidencialidade dos pacotes [19] e prover QoS [20] numa RSSF compartilhada. O POP-C++ utiliza a pilha de rede do sistema operacional no qual é executado, da forma que uma aplicação qualquer a utilizaria. Desta forma, a implementação para RSSFs se baseia no EPOS para que este tipo de funcionalidade seja implementada.

## 6 *Avaliação do POP-C++ em RSSFs*

Neste capítulo são apresentados os resultados de duas avaliações do sistema de suporte de tempo de execução do POP-C++ para RSSFs, o primeiro com o objetivo de avaliar a performance do sistema, e o segundo com o objetivo de demonstrar o funcionamento do sistema inteiro com uma aplicação que faz chamadas de função do Grid na RSSF.

### 6.1 **Teste de Avaliação de Performance**

Este teste tem o objetivo de avaliar o sobrecusto que o sistema de suporte POP-C++ para RSSFs adiciona à aplicação. Este sobrecusto é medido através da comparação de duas implementações da seguinte aplicação: recuperar e atribuir um valor de 8 bits. Uma versão foi implementada sobre o POP-C++ e a outra diretamente no sistema operacional. No caso da implementação sobre o POP-C++, o cliente instancia uma Interface para um Objeto “Sensor-Test” que é executado em outro nodo, e chama os métodos `get()` e `set()` para recuperar e atribuir o dado. Na implementação nativa ao sistema operacional, o cliente envia pacotes pré-formatados que são recebidos no servidor, que conhece seu formato, e responde com o dado na carga útil do pacote.

A figura 6.1 mostra o código da parclass usada na implementação POP-C++ teste, e na figura 6.2 mostra a aplicação de teste; ela instancia a Interface para o Objeto e invoca o método `get()` uma quantidade pré-determinada de vezes. A organização dos componentes da aplicação entre os nodos é a mesma mostrada na figura 5.2.

A figura 6.3 mostra a implementação nativa da implementação. A função `sink()` faz o trabalho do lado da Interface, montando a mensagem, enviando-a e esperando a resposta do lado de sensoriamento. Assim como na versão POP-C++, este procedimento é repetido um número pré-determinado de vezes. A função `sensor()` funciona como o Objeto, esperando pedidos do outro nodo e respondendo-os de acordo com qual método deve ser invocado.

```
parclass SensorTest
{
    public:
        SensorTest(string machine, short n) @od.url(machine);};

        mutex void set(unsigned char val);
        mutex unsigned char get();

    private:
        unsigned char data;
};
```

Figura 6.1: Parclass SensorTest usada no teste de performance

```
int main(int argc, char **argv)
{
    SensorTest * s = new SensorTest("remus", 1);

    s->set(65);
    char value;

    for(int i = 0; i < TEST_ITERS; i++)
        value = s->get();
}
```

Figura 6.2: Código de aplicação do teste de performance

### 6.1.1 Plataforma de Hardware/Software do Teste

Os testes foram realizados usando dois nodos de sensoriamento Mica2 desenvolvidos em Berkeley; eles utilizam um rádio CC1000 de apenas um canal, um microcontrolador Atmel Atmega128 de 8MHz e possuem 4KB de memória principal e 128KB de memória flash para código.

Para prover suporte de comunicação, gerência de memória e concorrência que ambas implementações da aplicação necessitam, foi utilizado o Embedded Parallel Operating System [21] (EPOS). Ele consiste num framework baseado em componentes usado para gerar sistemas de suporte de tempo de execução para aplicações de computação dedicada. Para este teste, o protocolo MAC do EPOS foi configurado para confiabilidade, assegurando entrega de pacotes através de ACKs constantes e minimizando atrasos de rede utilizando um ciclo de atividade "sempre-ligado".

## 6.1.2 Medição de Performance

- **Tamanho de Pacote:** A figura 6.4 detalha o tamanho e o conteúdo dos pacotes de rede das duas implementações da aplicação. Os pacotes da versão POP-C++ são maiores por duas razões:
  - **Campo de Objeto:** Para permitir que mais que um Objeto seja executado em cada nodo, pacotes são individualmente endereçados para cada Objeto;
  - **Campo de Valor Semântico:** O Valor Semântico do método a ser chamado também é incluso no cabeçalho do pacote.

A adição de funcionalidade equivalente na implementação nativa da aplicação resultaria num tamanho de pacote similar, justificado pela informação adicional que é necessariamente transmitida quando se provê suporte para conjuntos complexos de aplicações.

- **Pedidos por Segundo:** Para avaliar o sobrecusto de execução que o sistema POP-C++ adiciona nesta aplicação, foi conduzido um teste de performance que mediu quantos pedidos por segundo puderam ser feitas do nodo cliente para o nodo servidor. A figura 6.5 mostra que a implementação POP-C++ foi capaz de realizar 6,875 chamadas remotas de método por segundo, enquanto a implementação nativa fez 7,046 pedidos por segundo. Esta diferença de 2,42% se deve ao processamento adicional feito pelo sistema de suporte de tempo de execução POP-C++ quando chamadas de método chegam da rede, e também aos bytes adicionais que são transmitidos na implementação POP-C++ da aplicação.

## 6.2 Teste de Prova de Conceito

Para demonstrar a interação entre o Grid e um RSSF, um teste de prova de conceito foi realizado. Ele consiste em um teste da cadeia completa, onde um nodo no Grid faz chamadas à funções executadas na RSSF. Este teste tem os seguintes objetivos:

- Demonstrar o funcionamento conjunto de todos elementos ;
- Demonstrar a compatibilidade dos Objetos em execução nas RSSFs com Interfaces em execução tanto no Grid quanto na própria RSSF;
- Através de uma comparação com o teste anterior, avaliar o custo da interface Grid-RSSF.

A diferença entre este teste e o teste de performance discutido anteriormente é a localização da Interface, e, por consequência, o caminho que as chamadas de método devem percorrer para alcançar o Objeto. A figura 6.6 mostra a organização dos componentes nesse teste, incluindo o Broker de Proxy que se faz necessário pela interação das diferentes redes.

Outra consequência da nova localização da Interface é a interação entre as duas implementações do sistema de suporte de tempo de execução POP-C++; a Interface usa a implementação original, e o Broker e o Objeto usam a implementação para RSSFs. Apesar desta diferença, a imagem usada no nodo que hospeda o Objeto é exatamente a mesma do teste de performance, demonstrando a compatibilidade entre as duas implementações.

### 6.2.1 Plataforma de Hardware/Software do Teste

Além do nodo de sensoriamento Mica2 utilizado para a execução do Broker e do Objeto, são utilizados dois computadores de mesa IA32, ambos com o sistema de suporte de tempo de execução POP-C++ original e conectados por uma rede Ethernet 10/100Mbit. Um deles é hóspede para a Interface, e o outro serve de nodo de contato entre o Grid e a RSSF. Este último usa um nodo Mica2 conectado através de um cabo serial comunicando-se a 57600 baud para toda comunicação com o nodo de sensoriamento.

Para prover suporte a ambos nodos do Grid, o sistema operacional Linux foi utilizado. O sistema operacional EPOS foi utilizado em ambos Mica2, provendo a funcionalidade de comunicação serial utilizada pelo nodo de contato.

### 6.2.2 Medição de Performance

A mudança de localização da Interface de dentro da RSSF para um nodo do Grid diminuiu o número de pedidos por segundo para 3,804. A diminuição de performance se dá pela soma dos vários custos trazidos pela nova arquitetura, entre eles a troca de pacotes na rede do Grid, o processamento extra realizado no Broker de Proxy em todas as invocações de método e também a comunicação serial entre o Broker de Proxy e o nodo Mica2 que faz a ponte entre o Grid e a RSSF.

O uso de uma interface com a rede sem fio melhor integrada com o nodo de contato (como um transceptor no barramento PCI) melhoraria a performance por eliminar a conexão serial e qualquer processamento realizado num nodo de sensoriamento de baixa capacidade. Os outros fatores, como a troca de pacotes no Grid, são consequências naturais do aumento da complexidade e escopo do sistema.

```
struct Message {
    unsigned char method;
    unsigned char param;
};

void sink() {

    NIC nic;

    unsigned char src, prot;
    unsigned int size;

    Message msg;

    msg.method = 0;
    msg.param = 65;

    nic.send(0, 0, &msg, sizeof(msg));           // chamada
    nic.receive(&src, &prot, &msg, sizeof(msg)); // resposta

    for(int iter = 0; iter < TEST_ITERS; iter++) {

        msg.method = 1;

        nic.send(0, 0, &msg, sizeof(msg));       // chamada
        nic.receive(&src, &prot, &msg, sizeof(msg)); // resposta

        memset(&msg, sizeof(msg), 0);

    }
}

void sensor() {

    char value;

    NIC nic;

    unsigned char src, prot;
    unsigned int size;

    Message msg;

    while(1) {

        nic.receive(&src, &prot, &msg, sizeof(msg)); // chamada

        if(msg.method == 0)
            value = msg.param;
        else if(msg.method == 1)
            msg.param = value;

        nic.send(0, 0, &msg, sizeof(msg));       // resposta

    }
}
```

Figura 6.3: Implementação nativa do teste de performance

POP-C++	Origem	Destino	Protocolo	Tamanho	Objeto	Metodo	Semantica	Valor	CRC	
Nativa	Origem	Destino	Protocolo	Tamanho	Metodo	Valor	CRC			
Bytes	0	1	2	3	4	5	6	7	8	10

Figura 6.4: Comparação de Tamanho de Pacote

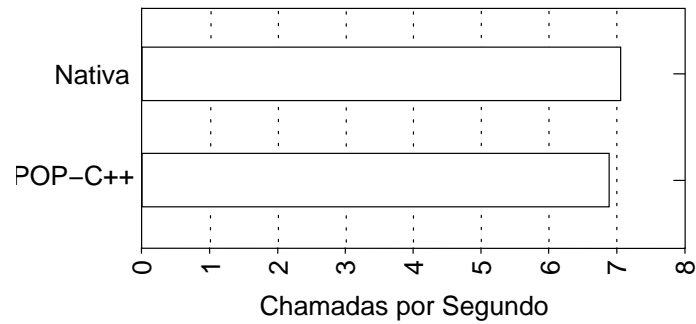


Figura 6.5: Comparação de Pedidos por Segundo

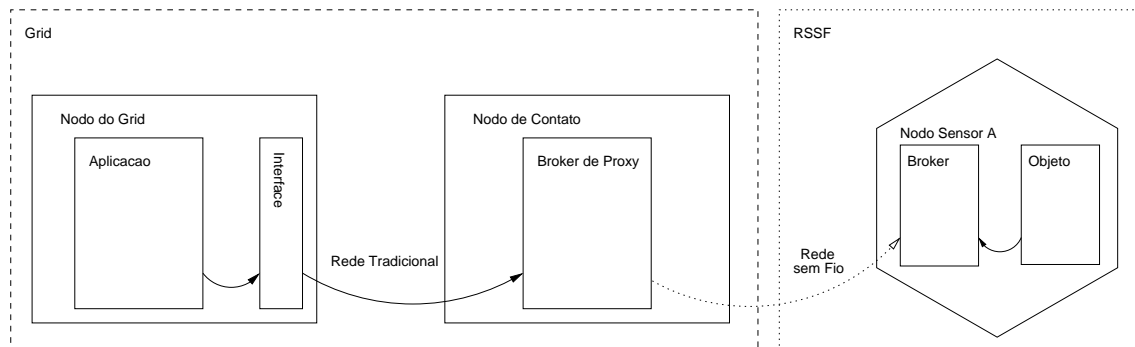


Figura 6.6: Organização dos componentes do teste de prova de conceito

## **7 *Trabalhos Relacionados***

Este capítulo discute outros projetos que compartilham o objetivo de que aplicações no Grid se comuniquem com RSSFs, e comenta as maneiras nas quais a metodologia deles se relaciona com a deste trabalho.

### **7.1 Cougar**

O Cougar [3] é uma metodologia para extração de dados de RSSFs através do processamento distribuído de consultas, baseado num otimizador de consultas localizado no nodo de contato com a RSSF e uma camada de proxy de consultas localizado nos nodos de sensoria-mento. Esta camada interage tanto com a aplicação quanto com a camada de roteamento da pilha de rede.

A aplicação faz consultas ao otimizador da mesma forma que em um banco de dados tra-dicional. Ao receber consultas, o otimizador cria um plano de processamento distribuído que define o fluxo dos dados entre os sensores e o processamento a ser realizado em cada um de-les. Desta forma, é tomada vantagem do fato de que processamento local é menos custoso em termos de energia que a transmissão de dados pela rede sem fio.

### **7.2 TAG: Tiny AGgregation Service**

O TAG [22] consiste em um serviço de agregação de dados de redes de sensores baseados em TinyOS. Como o Cougar, ele usa uma interface de consultas para permitir que usuários extraiam dados da RSSF, e baseia-se no processamento distribuído das consultas dentro da RSSF para diminuir o consumo de energia causado pela transmissão de dados pela rede sem fio. Após uma consulta chegar no nodo de contato com a RSSF, operadores que implementam a consulta são distribuídos pela rede, e após disso os dados relevantes são retornados ao nodo de contato através de uma árvore de roteamento da qual ele é base. À medida que os dados são



transmitidos, eles são agregados de acordo com uma função de agregação relevante à consulta em processamento.

As consultas no TAG, diferentemente das consultas em bancos de dados tradicionais, têm como resultado um fluxo de valores. Esta diferença é fruto da natureza periódica das aplicações que utilizam RSSFs, como de monitoramento de ambientes, onde a análise de como os valores dos sensores se comportam ao longo de um período é necessária.

## 7.3 **TinyDB**

O TinyDB [2] é uma evolução do TAG. Uma das principais novas funcionalidades trazidas pelo TinyDB é a consulta baseada em evento, que permite que o nodo fique dormiente até que o evento a ser monitorado ocorra, diminuindo ainda mais o consumo de energia do sistema. Eventos também podem ser utilizados para causar o término de uma consulta corrente, nos casos onde o fluxo gerado por uma consulta não é mais de interesse da aplicação depois que um certo evento ocorra.

No nodo de contato, responsável pela otimização das consultas, foi adicionada a funcionalidade de ordenação de predicados e consultas. A ordenação de predicados faz com que as operações sobre sensores menos custosas em energia sejam executadas primeiro, caso possam acarretar na não-execução de outras operações mais custosas. Dada uma consulta que envolve os valores do acelerômetro (amostragem menos custosa) e do magnetômetro (amostragem mais custosa), se a amostra do acelerômetro pode causar o cancelamento da amostra do magnetômetro (por colocar o sensor fora do escopo selecionado pela consulta), ela deve sempre ser feita primeiro.

Outra extensão foi feita para que informação semântica relevante às consultas ativas fosse utilizada como auxílio no algoritmo de roteamento. Pela natureza de agregação realizada pelo TAG e pelo TinyDB, existem rotas a serem tomadas que, apesar de não serem as mais diretas à raiz, custam menos energia para a RSSF como um todo por diminuir a quantidade de mensagens trocadas entre os nodos dela.

## 7.4 **Hourglass**

O Hourglass [23] insere uma Rede de Coleta de Dados (Data-Collection Network - DCN) entre as aplicações (Application Entry Points - AEPs) e as redes de sensores das quais elas fazem aquisição de dados (Sensor Entry Points - SEPs). A DCN é composta de vários sistemas

na Internet que fazem a descoberta e consulta em múltiplas RSSFs.

O Hourglass abstrai os detalhes internos das RSSFs completamente, e provê funcionalidade tradicional de Web Services como registro e descoberta de serviços, bem como realizar o roteamento de dados das RSSFs para as aplicações. Os protocolos adotados para isso são baseados em *publish/subscribe*, onde as RSSFs publicam seus dados através dos SEPs, e as aplicações fazem a requisição desses dados através das AEPs.

Para possibilitar a interação entre o maior número possível de AEPs e SEPs, o Hourglass usa vários padrões estabelecidos na Internet, como XML, SOAP, Web Services Description Language (WSDL) e Open Grid Services Architecture (OGSA). Com eles, se espera que a adição de novos serviços de sensoriamento e a implantação de novas aplicações seja feita com facilidade.

## 7.5 Comentários

O Cougar, TAG, TinyDB e outros esforços de pesquisa [22] [24] implementam processadores distribuídos de consultas, colocando grandes esforços na otimização das consultas e roteamento eficiente. Usando estas técnicas, eles foram capazes de alcançar considerável redução no consumo de energia, além de externalizar uma interface similar a SQL, amigável para o programador da aplicação. Este trabalho compartilha estes objetivos, porém no lugar de otimização ativa, ele sai do caminho do programador de aplicação permitindo acesso completo ao hardware dos nodos sensores. Além disso, existe a possibilidade de implementar as funcionalidades de otimização de consultas do Cougar, TAG e TinyDB como código de Objeto POP-C++, provendo funcionalidades similares.

Devido ao fato do Hourglass deixar os protocolos e a arquitetura de software a serem utilizados dentro das RSSFs em aberto, a extensão do POP-C++ feita neste trabalho poderia ser utilizada pela DCN para realizar toda comunicação dentro das RSSFs e prover um fluxo de dados para ela. As aplicações então usariam a interface uniforme provida pela DCN para fazer consultas às RSSFs. Esta forma de utilização abriria mão do uso dos objetos POP-C++ por todo Grid, limitando-o à implementação de um otimizador de consultas similar aos dos outros trabalhos comentados a ser executado no SEP.

## 8 *Conclusão*

Este trabalho descreveu as Redes de Sensores sem Fio e o Grid, ambos conceitos definidos nos últimos anos, e levanta a possibilidade de utilização de um paradigma único para a programação de aplicações que fazem uso de ambas essas tecnologias. Estas aplicações são distribuídas pelo Grid, fazendo uso de seu recurso computacional, mas também precisam da capacidade de sensoriamento possibilitado pelas RSSFs.

Como solução para esse problema, foi apresentada uma maneira de utilizar objetos paralelos e distribuídos para integrar o Grid e RSSFs, através de uma extensão do sistema de suporte de tempo de execução do POP-C++ para dentro das redes de sensoriamento. Utilizando o POP-C++ para realizar esta integração torna-se possível que o programador de aplicação utilize as RSSFs para múltiplas aplicações concorrentes transparentemente, utilizando interfaces instanciadas dentro do Grid para invocar métodos em objetos executados nas RSSFs.

Ao comparar uma aplicação implementada com e sem POP-C++, o sistema de suporte de execução deste trabalho mostrou um sobrecusto pequeno que é justificado pela habilidade de suportar múltiplas aplicações concorrentemente. Também foi avaliado o funcionamento de um sistema completo, envolvendo nodos IA32 conectados a uma rede tradicional fazendo requisições a uma RSSF, demonstrando a capacidade do POP-C++, distribuído entre o Grid e uma RSSF, de prover suporte a uma aplicação implementada com um único paradigma, sem uma fronteira bem definida entre os dois ambientes onde ela é executada.

Foram discutidas também outras metodologias usadas na literatura para promover a integração entre o Grid e RSSFs, e comentada a relação entre este trabalho e elas; de forma geral, o POP-C++ pode ser utilizado como base para a implementação de uma grande quantidade de funcionalidades complexas, por não limitar o acesso do programador da aplicação ao hardware dos nodos das RSSFs.

## *Referências Bibliográficas*

- 1 FOSTER, I.; KESSELMAN, C.; TUECKE, S. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, v. 2150, p. 1–??, 2001. Disponível em: <[citeseer.ist.psu.edu/foster01anatomy.html](http://citeseer.ist.psu.edu/foster01anatomy.html)>.
- 2 MADDEN, S. et al. The design of an acquisitional query processor for sensor networks. In: *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press, 2003. p. 491–502. ISBN 1-58113-634-X.
- 3 YAO, Y.; GEHRKE, J. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, ACM Press, New York, NY, USA, v. 31, n. 3, p. 9–18, 2002. ISSN 0163-5808.
- 4 NGUYEN, T.-A.; KUONEN, P. Paroc++: A requirement-driven parallel object-oriented programming language. In: *Proceedings of the Eighth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'03)*. [S.l.]: IEEE Computer Society, 2003. ISBN 0-7695-1880-X.
- 5 EL-HOIYDI, A. Aloha with preamble sampling for sporadic traffic in ad hoc wireless sensor networks. In: *IEEE International Conference on Communications (ICC)*. New York: [s.n.], 2002. Disponível em: <<http://dx.doi.org/10.1109/ICC.2002.997465>>.
- 6 HOESEL, L. v.; HAVINGA, P. A lightweight medium access protocol (LMAC) for wireless sensor networks. In: . Tokyo, Japan: [s.n.], 2004. Disponível em: <<http://wwwhome.cs.utwente.nl/hoesel/publications/VanHoesel-INSS2004.pdf>>.
- 7 LU, G.; KRISHNAMACHARI, B.; RAGHAVENDRA, C. An adaptive energy-efficient and low-latency MAC for data gathering in sensor networks. In: *Int. Workshop on Algorithms for Wireless, Mobile, Ad Hoc and Sensor Networks (WMAN)*. Santa Fe, NM: [s.n.], 2004. Disponível em: <<http://dx.doi.org/10.1109/IPDPS.2004.1303264>>.
- 8 WOO, A.; CULLER, D. A transmission control scheme for media access in sensor networks. In: *7th ACM Int. Conf. on Mobile Computing and Networking (MobiCom)*. New York, NY: ACM Press, 2001. p. 221–235. ISBN 1-58113-422-3. Disponível em: <<http://dx.doi.org/10.1145/381677.381699>>.
- 9 DAM, T. v.; LANGENDOEN, K. An adaptive energy-efficient MAC protocol for wireless sensor networks. In: . Los Angeles, CA: [s.n.], 2003. p. 171–180. Disponível em: <<http://dx.doi.org/10.1145/958491.958512>>.
- 10 POLASTRE, J.; HILL, J.; CULLER, D. Versatile low power media access for wireless sensor networks. In: *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. New York, NY, USA: ACM Press, 2004. p. 95–107. ISBN 1-58113-879-2.

- 11 YE, W.; HEIDEMANN, J. Medium access control in wireless sensor networks. Kluwer Academic Publishers, Norwell, MA, USA, p. 73–91, 2004.
- 12 LANGENDOEN, K.; HALKES, G. Embedded systems handbook. In: \_\_\_\_\_. [S.l.]: CRC press, 2005. cap. Energy-Efficient Medium Access Control.
- 13 TECHNOLOGY, C. *MPR/MIB Mote Hardware User's Manual*. San Jose, CA, September 2005.
- 14 PROJECT, B. *BTnode rev3 Hardware Reference*. jan. 2006. Em <http://www.btnode.ethz.ch/Documentation/BTnodeRev3HardwareReference>. Acessado em 25 de Janeiro de 2006.
- 15 FOSTER, I.; KESSELMAN, C. The Globus project: a status report. *Future Generation Computer Systems*, v. 15, n. 5–6, p. 607–621, 1999. Disponível em: <[citeseer.ist.psu.edu/foster98globus.html](http://citeseer.ist.psu.edu/foster98globus.html)>.
- 16 EGEE. *gLite: Lightweight Middleware for Grid Computing*. jan. 2007. Em <http://glite.web.cern.ch/glite/default.asp>. Acessado em 12 de Janeiro de 2007.
- 17 DUNKELS, A. et al. Run-time dynamic linking for reprogramming wireless sensor networks. In: *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*. New York, NY, USA: ACM Press, 2006. p. 15–28. ISBN 1-59593-343-3.
- 18 DENG, J.; HAN, R.; MISHRA, S. Defending against path-based dos attacks in wireless sensor networks. In: *SASN '05: Proceedings of the 3rd ACM workshop on Security of ad hoc and sensor networks*. New York, NY, USA: ACM Press, 2005. p. 89–96. ISBN 1-59593-227-5.
- 19 BANERJEE, S.; MUKHOPADHYAY, D. Symmetric key based authenticated querying in wireless sensor networks. In: *InterSense '06: Proceedings of the first international conference on Integrated internet ad hoc and sensor networks*. New York, NY, USA: ACM Press, 2006. p. 22. ISBN 1-59593-427-8.
- 20 OUFERHAT, N.; MELLOUCK, A. Qos dynamic routing for wireless sensor networks. In: *Q2SWinet '06: Proceedings of the 2nd ACM international workshop on Quality of service & security for wireless and mobile networks*. New York, NY, USA: ACM Press, 2006. p. 45–50. ISBN 1-59593-486-3.
- 21 FRÖHLICH, A. A.; SCHRÖDER-PREIKSCHAT, W. EPOS: an Object-Oriented Operating System. In: *2nd ECOOP Workshop on Object-Oriented and Operating Systems*. Lisbon, Portugal: [s.n.], 1999. (Chemnitzer Informatik-Berichte, CSR-99-04), p. 38–43.
- 22 MADDEN, S. et al. Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, ACM Press, New York, NY, USA, v. 36, n. SI, p. 131–146, 2002. ISSN 0163-5980.
- 23 GAYNOR, M. et al. Integrating wireless sensor networks with the grid. *IEEE Internet Computing*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 8, n. 4, p. 32–39, 2004. ISSN 1089-7801.

- 24 BONNET, P.; GEHRKE, J.; SESHADRI, P. Towards sensor database systems. In: *MDM '01: Proceedings of the Second International Conference on Mobile Data Management*. London, UK: Springer-Verlag, 2001. p. 3–14. ISBN 3-540-41454-1.