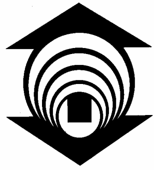


FUNDAÇÃO EDSON QUEIROZ
UNIVERSIDADE DE FORTALEZA – UNIFOR

DIORGENS MIGUEL MEIRA

UM MECANISMO DE PROCESSAMENTO DE CONSULTAS DISTRIBUÍDO EM REDES DE SENSORES SEM FIO

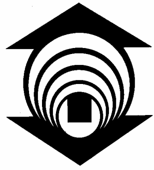
Fortaleza

2007

Livros Grátis

http://www.livrosgratis.com.br

Milhares de livros grátis para download.



FUNDAÇÃO EDSON QUEIROZ
UNIVERSIDADE DE FORTALEZA – UNIFOR

DIORGENS MIGUEL MEIRA

UM MECANISMO DE PROCESSAMENTO DE CONSULTAS DISTRIBUÍDO EM REDES DE SENSORES SEM FIO

Dissertação apresentada ao Curso de Mestrado em Informática Aplicada da Universidade de Fortaleza como requisito parcial para a obtenção do Título de Mestre em Ciências da Computação.

Orientador: Prof. Dr. Ângelo Roncalli Alencar Brayner

Fortaleza

2007

DIORGENS MIGUEL MEIRA

UM MECANISMO DE PROCESSAMENTO DE CONSULTAS DISTRIBUÍDO EM REDES DE SENSORES SEM FIO

Data de Aprovação: ____________________

Banca Examinadora:

____

Prof. Ângelo Roncalli Alencar Brayner, Dr.-ing.

(Presidente da Banca)

____

Prof. Javam de Castro Machado, Dr.

(Universidade Federal do Ceará - UFC)

____

Prof. Pedro Porfírio Muniz Farias, Dr.

(Universidade de Fortaleza - UNIFOR)

MEIRA, DIORGENS MIGUEL

Um Mecanismo de Processamento de Consultas Distribuído em Redes de Sensores Sem Fio [Fortaleza] 2007

154p., 29,7 cm (MIA/UNIFOR, M. Sc, Ciência da Computação, 2007)

Dissertação de Mestrado – Universidade de Fortaleza, MIA

1. Banco de Dados
2. Processamento de Consultas
3. Redes de Sensores Sem Fio

I. MIA/UNIFOR II. TÍTULO (série)

A meus pais, Miguel e Delfina,
a minha esposa, Edlena, a
meus filhos, Daniel e Cecília.

AGRADECIMENTOS

Ao professor Ângelo Brayner pela sua orientação, por todo conhecimento transmitido, suas valiosas sugestões, acompanhamento e paciência dedicada à concretização deste trabalho.

A todos os meus amigos e familiares, pelo apoio, incentivo e por souberem compreender minhas ausências em muitos momentos de lazer e confraternização, em especial a Edlena, minha esposa, e meus filhos, Daniel e Cecília, pelo tempo e resignação dedicados à construção deste trabalho.

Ao Banco do Nordeste do Brasil S.A., pelo apoio para a consecução deste trabalho.

A todos que contribuíram, direta ou indiretamente, para o desenvolvimento deste projeto.

A Deus, pela força e pela graça concedidas para eu chegar à conclusão de mais um ciclo da minha vida.

Resumo da Dissertação apresentada ao MIA/UNIFOR como parte dos requisitos necessários para a obtenção do título de Mestre em Ciência da Computação.

UM MECANISMO DE PROCESSAMENTO DE CONSULTAS DISTRIBUÍDO EM REDES DE SENSORES SEM FIO

Diorgens Miguel Meira

Novembro / 2007

Orientador: Dr.-Ing. Ângelo Roncalli Alencar Brayner

Programa: Ciência da Computação

Uma rede de sensores sem fio (RSSF) consiste de grupos de sensores em rede, distribuídos espacialmente e enviando dados a um nó sumidouro, denominado estação base. Estes sensores em geral são fortemente limitados em recursos. Em uma RSSF os dados coletados são passados de sensor a sensor até que a estação base seja alcançada. Considerando as larguras de bandas mais comuns disponíveis, grandes volumes de dados podem produzir congestionamento no tráfego das RSSFs. Neste trabalho, defendemos o uso de tecnologias de bancos de dados em redes de sensores sem fio para reduzir a quantidade de dados que devem ser transmitidos e, conseqüentemente, reduzir o tráfego de dados na rede. Pelo fato da comunicação através de um meio sem fio consumir grandes quantidades de energia, reduzir a quantidade de dados transmitidos pelos sensores pode melhorar significativamente o tempo de vida da RSSF. Este trabalho apresenta um mecanismo de processamento de consultas em RSSFs. O principal objetivo do mecanismo proposto é melhorar o tempo de vida útil da RSSF, e tem como característica fundamental à execução de consultas de maneira distribuída através de vários nós sensores. Os resultados experimentais apresentados neste trabalho provam a eficiência do mecanismo proposto.

Abstract of Thesis presented to MIA/UNIFOR as a partial fulfillment of the requirements for the degree of Master of Science Computer

A DISTRIBUTED QUERY PROCESSING ENGINE IN WIRELESS SENSOR NETWORKS

Diorgens Miguel Meira

November / 2007

Advisor: Dr.-Ing. Ângelo Roncalli Alencar Brayner

Department: Computing Science

A wireless sensor network (WSN) consists of groups of spatially distributed networked sensors sending data to a sink node, called base station. These sensors are usually strongly resource constrained. In a WSN, collected data are passed from sensor to sensor until the base station is reached. Considering common available bandwidths, large data volumes may produce heavy traffic congestion in WSNs. In this work, we advocate the use of database technology in sensor networks to reduce the amount of data to be transmitted by sensors and consequently the data traffic in the network. Since data communication by means of a wireless medium consumes a lot of energy, reducing data transmitted by sensors can improve significantly the sensor network lifetime. This work presents a query engine for processing queries in WSNs. The main goal of the proposed query engine is to improve significantly the sensor network lifetime. The key feature of the proposed engine is to execute queries in a distributed way through the various sensor nodes of a WSN. Experimental results presented in this work prove the efficiency of the proposed query engine.

SUMÁRIO

LISTA DE FIGURAS xii
LISTA DE TABELAS xiv
LISTA DE ACRÔNIMOS xv

Capítulo Página

1 INTRODUÇÃO
1.1 Motivação 16
1.2 Objetivo 17
1.3 Organização do trabalho 18

2 REDES DE SENSORES SEM FIO
2.1 Introdução 20
2.2 Sensores 20
2.3 Características aplicadas a redes de sensores sem fio 22
2.3.1 Métodos de sensoriamento 22
2.3.2 Conservação de energia 24
2.3.3 Comunicação 25
2.3.4 Processamento nos nós sensores 26
2.3.5 Escalabilidade 27
2.3.6 Roteamento e disseminação de dados 27
2.3.7 Gerenciamento de falhas 32
2.4 Nós sensores inteligentes 33
2.4.1 Mica2 Mote 33
2.4.2 BEAN 34
2.4.3 Medusa MK-2 35
2.4.4 Smart Dust 35
2.5 Considerações finais 36

3 PROCESSAMENTO DE CONSULTAS
3.1 Introdução 37

3.2 Processamento de consultas em bancos de dados 38
3.3 Sistemas de gerenciamento em fluxos de dados 38
3.4 Consultas contínuas 40
3.5 Consultas em RSSF 40
3.6 Considerações finais 43

4 SNQL – SENSOR NETWORK QUERY

4.1 Introdução 44
4.2 Componente 44
4.3 Componente DDL....47
4.4 Declarações para execução em lote 51
4.5 Funções de agregação....52
4.6 Considerações finais 53

5 PDCRS: UM MECANISMO PARA PROCESSAMENTO DISTRIBUÍDO DE CONSULTAS EM RSSF

5.1 Introdução....54
5.2 Processamento distribuído de consultas....55
5.2.1 Gerenciamento de esquemas....57
5.2.2 Processamento de consultas....60
5.2.3 Interpretação....61
5.2.4 Decomposição 62
5.2.5 Disseminação das subconsultas....66
5.2.6 Processamento das subconsultas 67
5.2.7 Pós-processamento para consolidação dos resultados 68
5.3 Considerações finais 69

6 SIMULAÇÃO

6.1 Introdução....70
6.2 Cenário utilizado....71
6.3 Testes realizados 73
6.4 Resultados....74
6.5 Considerações finais 79

7 CONCLUSÕES E TRABALHOS FUTUROS........80
REFERÊNCIAS BIBLIOGRÁFICAS........82

APÊNDICES

Apêndice A. Algoritmos do PDCRS........90
Apêndice B. Simulador de rede de sensores sem fio........97
Apêndice C. Modelo de consumo de energia pelo sensor........125
Apêndice D. Principais classes do PDCRS........130
Apêndice E. Estrutura sintática da SNQL........147

LISTA DE FIGURAS

Figura Página

Figura 2.1 Os componentes de um nó sensor inteligente. 21

Figura 2.2 Nó sensor Mica2. 33

Figura 2.3 Placa do sensor BEAN...34

Figura 2.4 Nós Medusa MK-2 básico e Medusa MK-2 com placa de ultra-som.........35

Figura 2.5 Nó sensor Smart Dust. 36

Figura 4.1 Sintaxe geral componente DDL da SNQL..45

Figura 4.1 Sintaxe geral componente DDL da SNQL..45

Figura 4.2 Sintaxes para gerenciamento de esquemas 48

Figura 5.1 Estratégia do PDCRS para processamento de consultas em RSSFs. 56

Figura 5.2 Esquema global materializado sob forma de um arquivo XML. 60

Figura 5.3 Consulta Q, escrita em SNQL. 61

Figura 5.4 Plano de execução da consulta Q. 62

Figura 5.5 Plano de execução gerado na fase de decomposição. 64

Figura 5.6 Fragmentação gerada pelo PDCRS..64

Figura 5.7 Consulta diretora Q0, em XML...66

Figura 5.8 Subconsultas Q1 e Q2, em XML..66

Figura 6.1 Diagrama simplificada da RSSF. 73

Figura 6.2 Tempo de resposta da rede. 74

Figura 6.3 Tempo de vida da rede. 75

Figura 6.4 Tempo de vida dos nós sensores. 76

Figura 6.5 Nós ativos por tempo e região. 77

Figura 6.6 Nós inativos por tempo e região..78

Figura 6.7 Taxa de entrega de pacotes e Taxa de descarte de pacotes 79

Figura A.1 Algoritmo gerenciamento para obtenção da consulta...............................90

Figura A.2 Algoritmo processamento da consulta na estação base. 91

Figura A.3 Algoritmo geral de processamento nos nós - Sensores. 93
Figura A.4 Algoritmo do bloco ObtemNovaConsulta - Sensores. 94
Figura A.5 Algoritmo do bloco RecebePacotes - Sensores. 95
Figura A.6 Algoritmo do bloco ProcessaMensagemRecebida - Sensores. 95
Figura A.7 Algoritmo de empacotamento de nova tupla coletada – Sensores...........96
Figura B.1 Diagrama com interação das estações da SNETSIM...............................99
Figura B.2 Diagrama com a comunicação entre os módulos do PDCRS. 100
Figura B.3 Algoritmo para descoberta e conexão com o Agente de rede. 103
Figura B.4 Algoritmo para tratamento de comunicação pelo agente de rede. 103
Figura B.5 Tela da console do Agente de Rede. 105
Figura B.6 Tela de rastreamento gráfico de comunicação na RSSF – *NetAgent*. ...106
Figura B.7 Tela principal da aplicação *Base Station Manager*. 107
Figura B.8 Tela principal da aplicação *Base Station Processor*. 111
Figura B.9 Tela principal da aplicação Sensor Node Processor. 114
Figura B.10 Opções para criação de nós sensores – SensorProc. 114
Figura B.11 Criação de nós sensors interativamente – SensorProc. 115
Figura B.12 Criação de nós sensors a partir de lista pré-definida – SensorProc. 116
Figura B.13 Exemplo de arquivo de lista de sensors – SensorProc 117
Figura B.14 Diagrama da rede de sensores conforme lista de sensores exemplo. .118
Figura B.15 Tela de propriedades de um nó sensor – SensorProc. 119
Figura B.16 Configuração exemplo para SiteConfig.xml. 120
Figura B.17 Estrutura do catálogo do esquema global 123
Figura D.1 Diagrama de classes do PDCRS. 130
Figura E.1 Comandos escritos em SNQL. 147
Figura E.2 Elementos de um filtro de seleção. 153
Figura E.3 Palavras reservadas da SNQL. 154

LISTA DE TABELAS

Tabela Página

Tabela 2.1 Exemplos de protocolos de roteamento centrado em dados. 29

Tabela 2.2 Exemplos de protocolos de roteamento hierárquico. 30

Tabela 2.3 Exemplos de protocolos de roteamento baseado em localização. 31

Tabela 3.1 Exemplos de linguagens de consulta em RSSF 42

Tabela 4.1 Descrição das cláusulas da SNQL. 47

Tabela 4.2 Funções de agregação da SNQL. 52

Tabela 5.1 Requisitos de elementos para o grupo de sensores no esquema global. 58

Tabela 5.2 Relocalização de itens de projeção, após a decomposição. 65

Tabela 5.3 Relocalização de itens de seleção, após a decomposição. 65

Tabela C.1 Parâmetros de consultas. 125

Tabela C.2 Parâmetros dependentes da plataforma de sensor..............................125

Tabela C.3 Parâmetros derivados calculados...126

Tabela C.4 Unidades de medida...129

LISTA DE ACRÔNIMOS

API : *Application Programming Interface*

DDL : *Data Definition Language*

DML : *Data Manipulation Language*

DSMS : *Data Stream Management Systems*

GPS : *Global Positioning System*

IP : *Internet Protocol*

MAC : *Medium Access Control*

MIPS : *Millions of Instructions Per Second*

PDCRS : Processamento e Distribuição de Consultas em Redes de Sensores

QEP : *Query Execution Plan*

RAM : *Random Access Memory*

ROM : *Read Only Memory*

RSSF : Rede de Sensores sem Fio

SGBD : Sistema Gerenciador de Banco de Dados

SGFD : Sistemas de gerenciamento de fluxos de dados

SNETSIM : *Sensor Network Simulator*

SNQL : *Sensor Network Query Language*

SQL : *Structured Query Language*

TCP : *Transmission Control Protocol*

TDMA : *Time Division Multiple Access*

UDP : *User Datagram Protocol*

WSN : *Wireless Sensor Network*

XML : *Extensible Markup Language*

Capítulo 1
INTRODUÇÃO

1.1 Motivação

O crescente avanço das técnicas de comunicação, especialmente na área de processamento de sinais, e de microeletrônica permitiu o desenvolvimento de pequenos dispositivos com baixo consumo de energia capazes de combinar as capacidades de sensoriamento e de comunicação sem fio. A organização de tais dispositivos microssensores em uma sofisticada arquitetura computacional e de comunicação permitiu criar um novo tipo de rede sem fio, denominada rede de sensores sem fio (RSSF). Essa nova tecnologia permite monitorar remotamente diferentes regiões geográficas e obter informações mais precisas, tirando vantagem de ser empregada bem próxima ao fenômeno de interesse.

Uma RSSF pode ser genericamente caracterizada por possuir uma grande quantidade de nós, adotar comunicação *broadcast*[1], ter uma topologia freqüentemente alterada e ser composta por sensores de arquiteturas simples, propensos a falhas, com restrições de energia, memória e capacidade de processamento [1]. Esta limitação de recursos pode existir em menor ou maior escala em vários nós de uma RSSF. Os algoritmos, protocolos e sistemas de gerenciamento projetados para estas redes devem ser sensíveis a estas restrições, de maneira a serem capacitados a adaptarem-se a cada novo cenário apresentado.

Aplicações executadas sobre redes de sensores normalmente trabalham com fluxos contínuos de dados (também conhecidos como *data streams* [5][33]). Um sensor utilizado para coleta de temperaturas, por exemplo, pode ser configurado para captar informações do ambiente continuamente. Neste caso, o volume de dados recepcionado será potencialmente infinito, inviabilizando o seu completo armazenamento. A alternativa mais viável consiste em enviar esses dados para nós com maior capacidade de processamento e armazenamento. Ainda assim, o

---

[1] Comunicação *broadcast*: forma de comunicação em que as informações transmitidas pelo nó de origem são difundidas para todos os nós que pertencem a uma determinada rede. Conforme a estratégia adotada, os nós receptores podem descartar as informações que não são de seu interesse [32][49].

volume de dados trafegado, se não sofrer nenhum tipo de racionalização, poderá onerar o canal de comunicação e os normalmente parcos recursos de energia disponíveis nos sensores.

Há muitas pesquisas atuais que tratam do acesso a dados que fluem em RSSFs [57][66]. Algumas propostas, defendidas no escopo destas pesquisas, tratam essas redes como um banco de dados [42] especial, manipulando os fluxos de dados a partir de técnicas de SGFD (Sistemas Gerenciadores de Fluxos de Dados). O Capítulo 3 desta dissertação (tópico 3.3) apresenta detalhamento acerca destas técnicas. A modelagem de dados adotada nesses casos corresponde a um modelo relacional convencional aplicado às necessidades destas redes, aproximando-se do conceito de um ambiente de banco de dados distribuídos. Isto permite organizar logicamente as informações, e oferecer suporte a linguagens de consulta baseadas em SQL (*Structured Query Language*)[56], além de dar maior flexibilidade à aplicação, capacitando-a a responder consultas do tipo *ad hoc* [4][42].

Os cenários apresentados pelas soluções de RSSF preceituam que consultas sejam submetidas à rede através de um computador com grande capacidade de processamento e armazenamento, denominado de estação base. Em resposta, dados são coletados do ambiente pelos nós sensores e transmitidos de volta à estação base, onde o resultado da consulta é finalmente disponibilizado ao usuário requisitante da informação.

No modelo de processamento de consultas supracitado é importante a existência de um mecanismo para coordenar a disseminação dinâmica de consultas para os nós sensores e a disponibilização contínua dos resultados à aplicação usuária enquanto estas ainda estão sendo executadas, além de prover técnicas para o uso racional dos recursos da rede, tais como sumarização dos dados coletados, agregação em rede [8][9][11] e outras estratégias de otimização.

1.2 Objetivo

Este trabalho apresenta um mecanismo de processamento distribuído de consultas em redes de sensores sem fio, denominado PDCRS (Processamento e Distribuição

de Consultas em Redes de Sensores), que irá permitir a execução de consultas em RSSFs de forma distribuída nos nós da rede. O PDCRS tem como ponto de partida o processamento de consultas expressas em uma linguagem declarativa no padrão SQL. Essas consultas, para serem injetadas no sistema, deverão passar por etapas de preparação de estruturas que permitirão a sua distribuição e gerenciamento, desde a solicitação pela aplicação usuária até o recebimento dos resultados gerados, de forma contínua, pelos nós sensores. O mecanismo aqui apresentado visa oferecer um padrão para o desenvolvimento destas etapas, e utiliza critérios de distribuição baseados em categoria de sensoriamento (tipo de sensoriamento ou aquisição coletado pelo dispositivo) ou por sua região de localização.

Para dar suporte à implementação dos conceitos do PDCRS, foi proposta uma extensão à linguagem SQL, a qual denominamos SNQL (*Sensor Network Query Language*), que sustentará as novas cláusulas específicas para o processamento de consultas aquisicionais em redes de sensores.

Neste trabalho também será apresentada a implementação de um conjunto de aplicações e definições que compõem um simulador para uma rede de sensores sem fio para distribuição de consultas [47], utilizado na prova de conceitos acerca da proposta do PDCRS, que se tornou a ferramenta de auxilio nas implementações das estratégias de consultas utilizadas neste trabalho.

1.3 Organização do trabalho

A presente dissertação está estruturada conforme descrito a seguir. O Capítulo 2 apresenta conceitos, componentes e desenvolvimentos relacionados a redes de sensores sem fio. O Capítulo 3 explora aspectos gerais acerca de processamentos de consultas tradicionais e dentro de cenários de redes de sensores. O capítulo 4 expõe a especificação de uma linguagem declarativa, a SNQL, que será utilizada no contexto deste trabalho. O Capítulo 5 apresenta a proposta de processamento de consultas distribuídas, através da descrição do mecanismo PDCRS (Processamento e Distribuição de Consultas em Redes de Sensores), mostrando seus conceitos, heurísticas utilizadas e cenário operacional. O Capítulo 6 descreve as experimentações realizadas com o mecanismo PDCRS, e apresenta os resultados dos testes obtidos. O Capítulo 7 apresenta as conclusões acerca deste

trabalho, referencia alguns trabalhos relacionados com os tópicos abordados, e discute trabalhos futuros. O Apêndice A mostra os algoritmos básicos relacionados com o PDCRS. O Apêndice B apresenta e detalha a implementação do simulador de rede de sensores sem fio, utilizada como prova de conceito para o mecanismo PDCRS. O Apêndice C mostra as interfaces das principais classes do mecanismo PDCRS, com seus métodos e atributos. O Apêndice D detalha a estrutura sintática da SNQL. O Apêndice E apresenta um modelo de custo de energia para nós sensores, utilizado na implementação do simulador de RSSF aqui referenciado.

Capítulo 2
REDES DE SENSORES SEM FIO

2.1 Introdução

Uma rede de sensores consiste de um ou mais grupos de dispositivos de sensoriamento ou de aquisição contínua de dados, interligados em rede, a qual é utilizada para disseminar os dados coletados de um determinado ambiente. Uma especificidade de uma rede de sensores trata-se das Redes de Sensores Sem Fio (RSSF) [1].

Neste capítulo, além da exposição dos conceitos básicos aplicados a redes de sensores sem fio, também são apresentados os componentes básicos das arquiteturas mais estudadas de nós sensores inteligentes (Seção 2.2), as principais características aplicadas às redes de sensores sem fio (seção 2.3), soluções disponíveis atualmente e projetos desenvolvidos em dispositivos de nós sensores (Seção 2.4).

2.2 Sensores

Um sensor é um dispositivo tecnológico com capacidade de detectar eventos ou determinadas condições físicas em ambientes ou substâncias, gerando um sinal elétrico em resposta ao sensoriamento realizado. Um sensor inteligente tem a capacidade de traduzir este sinal elétrico em informação digital e realizar algum tipo de processamento sobre ele para, em seguida, enviar a informação gerada através de uma rede, para ser utilizada por uma aplicação.

O modelo de um nó sensor inteligente[2] é constituído de quatro componentes básicos [1]: uma unidade de sensoriamento, uma unidade de processamento, uma unidade de rádio para comunicação sem fio e uma unidade de energia (Ver Figura 2.1). Dependendo da arquitetura implementada ou das necessidades das aplicações usuárias, os nós sensores também podem contar com componentes

---

[2] Doravante, neste documento, referências ao termo *nó sensor* terá significado equivalente a *nó sensor inteligente sem fio*.

adicionais, a exemplo de sistema para localização do nó, como GPS (*Global Positioning System*), sistema de mobilidade do dispositivo e gerador autônomo de energia.

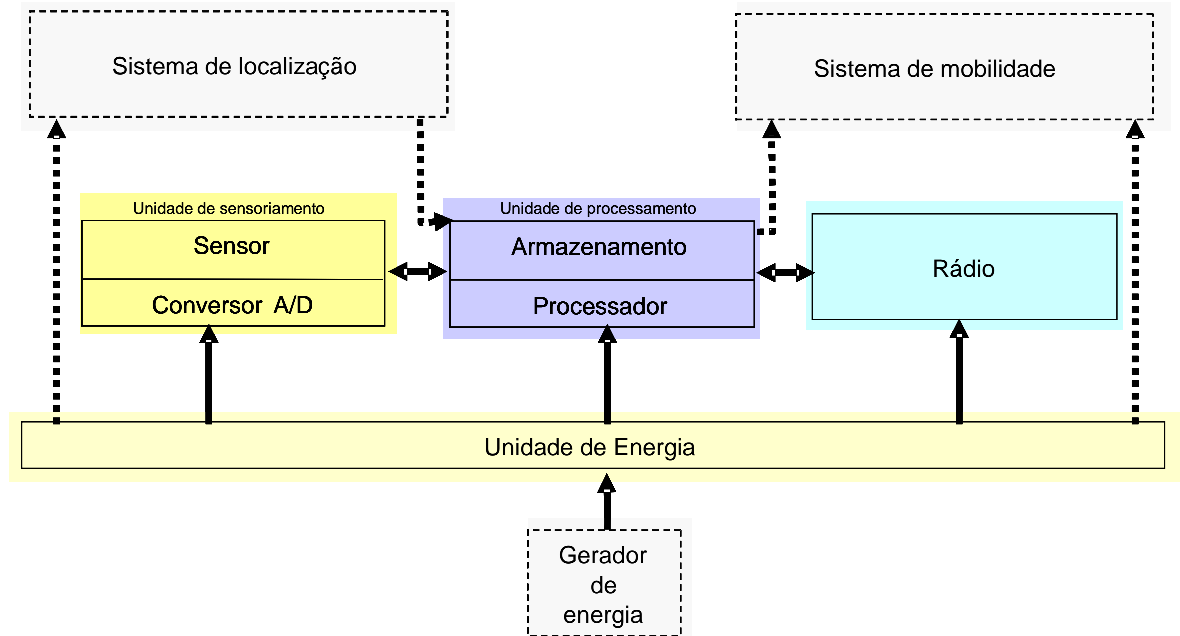


Figura 2.1 Os componentes de um nó sensor inteligente.

A unidade de sensoriamento tem a função de coletar leituras de eventos físicos no ambiente, e é composta pelo componente sensor propriamente dito e mais um conversor de analógico para digital, que transforma as leituras em dados binários.

A unidade de processamento possui um microprocessador, além de dispositivos de armazenamento (memória RAM, ROM, além de poder contar com outros tipos de memórias permanentes, como memória *Flash*[3]). Desta forma é possível a programação customizável das funções deste dispositivo.

O componente rádio implementa a comunicação entre os nós de uma RSSF, sendo a interface utilizada pela unidade de processamento para realizar a transmissão e recepção de dados com os outros nós. No modelo de rede de sensores aqui discutido, os circuitos de comunicação entre os nós funcionam num meio sem fio. Estes circuitos podem ser formados por técnicas de modulação sobre

rádio freqüência, infravermelho ou outros meios óticos. A utilização de radiofreqüência mostrou-se ser a abordagem mais utilizada neste tipo de rede [1].

A unidade de fornecimento de energia é composta por baterias, que podem ser de modelos que vão de pilhas comuns ou recarregáveis a outros modelos mais específicos, tais como a utilização de células solares. A maior parte das estratégias de processamento de consultas leva em conta a utilização de energia pelos nós sensores. Dessa forma, o conhecimento das características intrínsecas da unidade de fornecimento de energia é um aspecto importante em projetos de RSSFs.

2.3 Características aplicadas a redes de sensores sem fio

Redes de sensores sem fio diferem das redes tradicionais em vários aspectos, tais como: podem ser formadas por um grande número de nós; os nós sensores possuem fortes limitações de energia, capacidade de processamento e memória; muitas aplicações de RSSFs demandam características de auto-organização, ou seja, capacidade de se ajustar autonomamente às possíveis mudanças estruturais devido a intervenções externas, tais como mudanças topológicas (causadas por falhas, mobilidade ou inclusão de nós), reação a um evento de sensoriamento ou a uma solicitação feita por uma entidade externa (usuário ou sistema fixo).

Os algoritmos e protocolos definidos para as RSSFs devem levar em conta as particularidades deste tipo de rede, principalmente aquelas relacionadas com restrições dos recursos disponíveis nos nós sensores e também outras características definidas dentro do escopo da aplicação usuário. Algumas destas características são apresentadas nos subtópicos a seguir.

2.3.1 Métodos de sensoriamento

A escolha da categoria de sensoriamento dos dispositivos sensores dependerá das necessidades das aplicações demandantes. Dependendo da plataforma de nó sensor, um dispositivo pode oferecer mais de um tipo de sensoriamento ao mesmo

---

[3] Memória Flash: chip de memória que pode ser regravável e que, ao contrário de memória RAM, preserva o seu conteúdo sem a necessidade de fonte de alimentação.

tempo, a exemplo de determinadas placas de aquisição de múltiplas funções (Ver exemplo na arquitetura de nós sensores mostrada na subseção 2.4).

Independente da plataforma de nó sensor utilizada, o modo como os dispositivos realizam as coletas de dados dependem, por definição, da aplicação para a qual eles servem. Este aspecto, de acordo com a tipificação proposta em [52], classifica a RSSF conforme seu método de sensoriamento, como mostrado a seguir:

- RSSF de Tempo Real: os nós sensores coletam os dados no momento em que a atividade associada acontece, buscando realizar maior amostragem possível dos dados num menor intervalo possível exigida pela aplicação. São utilizados por aplicações que envolvem risco para vidas humanas ou aplicações onde a rapidez da disponibilização do dado coletado é importante na tomada de decisão e definição de estratégias. Exemplos: monitoração de áreas sujeitas a desastres naturais e aplicações militares;

- RSSF Periódica: os nós sensores coletam dados sobre os fenômenos em intervalos regulares e pré-definidos. Um exemplo são as aplicações que monitoram animais em florestas. A coleta de dados é feita durante o período em que a espécie monitorada estiver em atividade, e são desligados durante o período em que os animais estiverem repousando;

- RSSF Contínua: os nós sensores coletam os dados continuamente, sem interrupção. Um exemplo são as aplicações de exploração interplanetária ou sondas espaciais que coletam dados continuamente, até a extinção da atividade do dispositivo, contribuindo para a formação de base de dados para pesquisas;

- RSSF Reativa: os nós sensores coletam dados em resposta ao disparo de eventos pré-programados ou quando solicitado pela aplicação de monitoração. Exemplos são as aplicações de detecção de presença ou gatilhos para determinadas condições ambientais, a exemplo de detectores de calor ou fumaça.

- RSSF Híbrida: rede que adota mais de uma abordagem para coleta de dados, sendo que a aplicação implementada avalia em que circunstância o método mais adequado será utilizado.

2.3.2 Conservação de energia

O tempo de vida operacional de cada nó sensor é função da quantidade de energia disponível em sua unidade de fornecimento de energia. Assim, a adoção de estratégias que otimizam a utilização da carga da sua bateria é fator preponderante para a vida útil da rede como um todo, considerando que nem sempre se utilizam dispositivos com unidades geradoras de energia independentes, além do fato de que muitas vezes a localização dos sensores em pontos de difícil acesso dificulta tarefas de manutenção de suas baterias, corroborado pela grande quantidade de nós que normalmente compõem essas redes [1].

O consumo de energia deve ser analisado sob três aspectos de operação do nó sensor: sensoriamento, comunicação e processamento dos dados. Destes três aspectos, o que demanda maior atenção é a comunicação, pois a energia gasta para o processamento dos dados ou sensoriamento é muito menor que a energia necessária para se transmitir ou receber dados. O exemplo mostrado em [67] ilustra essa diferença: o custo em energia para se transmitir 1 KB por uma distância de 100 metros pode chegar a aproximadamente o mesmo que executar 3 milhões de instruções em um processador com capacidade de processamento de 100 MIPS/W. Dessa forma, explorar o processamento local e distribuído, implementando estratégias que diminuam a necessidade de enviar grande volume de dados para a estação base da rede, é crucial para minimizar o consumo de potência em uma rede de sensores sem fio.

Lamentavelmente, a tecnologia das baterias não está progredindo no mesmo ritmo da tecnologia de processamento do silício, mas o desenvolvimento de padrões mais eficientes destes dispositivos de armazenamentos é esperado nos próximos anos [35]. Isso demonstra a importância dos investimento em pesquisas para aperfeiçoar os algoritmos responsáveis pela comunicação entre os nós de

uma RSSF [50], além do desenvolvimento de novos tipos de transmissores, mais eficientes quanto a utilização de energia.

2.3.3 Comunicação

Comunicação talvez seja o aspecto mais complexo em uma RSSF, por causa da natureza *ad hoc*[4] deste tipo de rede e a limitação de recursos dos dispositivos envolvidos. A tarefa de comunicação pode ser dividida em subtarefas representadas em diferentes camadas [2], tal como numa rede tradicional de computadores. Uma subdivisão mais comum inclui uma camada de acesso ao meio físico (*Medium Access Control* – MAC) e uma camada de roteamento. Além de permitir que os nós participantes realizem processamento colaborativo, a função de comunicação também permite as interações entre um usuário e toda a rede de sensores [52].

Em qualquer rede de comunicação sem fio, os nós devem compartilhar um único meio para comunicação. O desempenho da rede depende de quão eficientemente os nós podem compartilhar este meio comum. Note que o pacote de transmissão é manuseado diretamente pela camada de acesso ao meio. Comparado com as redes com fio, uma significante parcela da energia do nó é despendida em transmissões de rádio ou escutando a recepção de eventuais pacotes a receber. Por outro lado, redes de sensores sem fio convivem com as inerentes restrições de energia. Desta maneira, é necessária a adoção de esquemas otimizados pelos protocolos de acesso ao meio para otimizar no que for possível a vida útil da rede.

No contexto de RSSFs, o requerimento apontado acima é extremamente crítico. De acordo com as características supracitadas, os nós de uma rede de sensores sem fio possuem recursos limitados de energia. E estes nós são largados para cumprir a sua atividades de forma desassistida, ou seja, após a sua entrada em operação, provavelmente não haverá suporte local para cada nó. Portanto, o

[4] Categorias de redes de comunicação em que os nós se comunicam diretamente uns com os outros, sem a interferência ou gerenciamento a partir de um nó principal. Termo freqüentemente associado a redes sem fio.

tempo de vida deste tipo de sensor depende quase que inteiramente de como a sua energia é utilizada durante as rotinas de comunicação.

Em relação a forma de entrega dos dados para a aplicação alvo da RSSF, essas redes podem ser classificadas [52] como:

- Contínuas: os nós sensores coletam dados e enviam à aplicação destino de forma contínua, durante todo o tempo de vida dos nós;
- Sob demanda: os nós sensores coletam dados e enviam à aplicação destino em resposta a uma consulta dessa aplicação;
- Orientadas a eventos: os nós sensores podem coletar e enviar dados quando detectam eventos que podem ocorrer no ambiente monitorado, ou conforme determinados critérios pré-estabelecidos na aplicação;
- Híbridas: quando mais de uma abordagem coexistem na mesma rede.

É Importante notar que a adoção de abordagens de envios contínuos de dados faz o consumo de energia crescer rapidamente, o que influencia negativamente no tempo de vida dos nós sensores.

2.3.4 Processamento nos nós sensores

Processamento de dados é uma função básica em nós sensores inteligentes, que pode significar desde tarefas simples, tal como detecção de valores anormais em dados coletados no ambiente, como procedimentos mais complexos, tais como decisão de movimentar o sensor dentro da rede em direção a uma melhor localização de sensoriamento. Projetos de redes de sensores sem fio normalmente definem procedimento para consolidar resultados das múltiplas saídas de nós que se agrupam, conforme a estratégia para topologia adotada. Ao optar por esse comportamento colaborativo, naturalmente somente o resultado agregado deverá ser enviado através da rede até os nós principais relacionados com a aplicação solicitante, e conseqüentemente o consumo de energia será menor do que a simples transmissão de qualquer valor coletado por nós individualmente.

Geralmente, nós sensores não são desenhados para processamento mais pesado, por causa das limitações de seus recursos computacionais. Tarefas de agregação, por exemplo, em nós sensores que não sejam *gateways* de grupos (ver tópico 2.3.6.2), não passam da execução de funções simples, tal como contagem de registros, somas, média, valores mínimos e máximos, aplicadas sobre os dados coletados. Um exemplo é apresentado em [41]. Esses nós sensores menores ficam executando uma espécie de pré-processamento dos resultados obtidos. Um exemplo disso são as tarefas de filtragem de resultados genéricos obtidos numa coleta de dados, como forma de reduzir a quantidade de eventos irrelevantes [61]. Após coletar o valor desejado, os sensores executam a redução dos dados, extraem os valores de interesse e os enviam para um outro nó com maior capacidade computacional. Esse pré-processamento diminui de maneira significativa a carga do processamento posterior, além de reduzir o volume de dados a ser transmitido.

2.3.5 Escalabilidade

Escalabilidade refere-se a habilidade de sistemas de comunicação conservar seus padrões de desempenho a despeito da alteração no tamanho da rede ou do número de nós ativos [55]. Em RSSFs, o número de nós sensores pode ser muito grande, na ordem de milhares de nós ou até maior. Neste tipo de rede o aspecto escalabilidade é um fator crítico, e torna-se um desafio, especialmente em ambientes dependentes de sincronização, como é o caso das redes sem fio.

Uma abordagem comum para conseguir escalabilidade é tratar dinamicamente o estado da rede, na medida em que esta for sendo alterada, fazendo as devidas adaptações na utilização e liberação dos recursos. Outra abordagem é tratar as interações entre os nós de forma compartilhada, a partir do desenvolvimento de estruturas hierárquicas para agrupar nós em conjuntos (*clusters*) e/ou utilizar estratégias de agregação de informações. A forma de implementar as abordagens citadas acima pode ser pela adoção de protocolos de roteamento apropriados para o modelo da rede que estiver sendo projetada.

2.3.6 Roteamento e disseminação de dados

Redes que possuem uma única camada, tendo um nó único que sirva como ponto de comunicação para que diversos outros nós possam entregar resultados a uma estação de processamento podem sofrer sobrecarrega neste ponto principal, principalmente quando a densidade de sensores da rede aumenta, e, conseqüentemente, incrementa o retardo das entregas de resultados. A utilização de protocolos de roteamento que permitam o particionamento da carga de trabalho entre grupos de nós sensores é uma maneira racional de garantir a disponibilidade dos resultados no tempo exigido pela aplicação, além de estar em conformidade com requisitos de conservação de energia das RSSFs.

Conceitos de agregação podem ser aplicados sobre protocolos de roteamento, pela combinação de dados que chegam de diferentes fontes durante o seu caminho pela rede. Isto permite eliminar redundância de informação, diminuir o número de pacotes transmitidos e, conseqüentemente, também racionalizar o consumo de energia pelos nós. O uso deste tipo de estratégias altera a forma mais tradicional de roteamento, que deixa de ser baseado em endereçamento ou localização (estilo encontrar a rota mais curta entre um par de nós endereçáveis), e passa a ser centrado nos dados, quando se deve procurar rotas a partir de múltiplas origens para um único destino, no qual se permite a consolidação em rede de dados redundantes [34].

Tendo em mente as características estudadas de uma RSSF, é muito importante neste tipo de rede considerar os mecanismos de sumarização durante a fase em que os dados estão sendo propagados. O custo de comunicação em RSSFs pode ser alto (em termos dos recursos utilizados), a largura da banda pode ser limitada, a disponibilidade de energia dos sensores pode ser pequena, além do suporte a transmissões a grandes distâncias neste tipo de dispositivo também ser baixo. Sendo assim, é bastante racional executar o máximo de processamento de dados na origem dos dados, e somente transmitir pela rede resultados sumarizados, obviamente guiados pela lógica das aplicações usuárias.

Não obstante existirem várias propostas de classificação de mecanismos de roteamento e propagação de dados para redes sem fio de grande escala, no escopo de RSSFs a classificação dos protocolos de roteamento pode ser

sumarizada em três grupos [54]: centrado em dados, hierárquicos e baseados em localização.

Os subtópicos a seguir detalham as característica de cada um dessas categorias de roteamento, e adicionalmente exemplifica com algumas propostas de algoritmos que as implementam no contexto de RSSFs.

2.3.6.1 Roteamento centrado em dados

Nesta categoria de roteamento, a estação base envia consultas para regiões específicas da RSSF e aguarda por dados dos sensores localizados nas regiões selecionadas. Quando da transmissão dos resultados pelos sensores, esses nós agregam os dados ainda enquanto são direcionados para a estação base, e os enviam em nome da região, eliminando redundâncias. Dependendo da lógica da aplicação, o conceito de região pode ser ampliado, englobando o significado de uma categoria de sensoriamento de nós sensores.

Pelo fato dos dados estarem sendo requisitados através de consultas, nomes de atributos são necessários para especificar as propriedades dos dados para a agregação. Assim, o roteamento torna-se dependente do esquema dos dados utilizado.

Haja vista a possibilidade de existir um grande número de nós sensores envolvidos, em muitas RSSFs não é prático associar estaticamente identificadores globais para cada nó. Assim, a heurística do tipo de roteamento aqui descrito considera que a seleção dos nós que farão parte da consulta será feita para uma região ou categoria de sensores da rede. A Tabela 2.1 descreve exemplos de protocolos desta categoria.

Tabela 2.1 Exemplos de protocolos de roteamento centrado em dados.

Protocolo	**Sumário**
SPIN - *Sensor Protocols for Information via Negotiation* [29]	Usa informações sobre a quantidade de energia disponível em cada sensor para fazer o roteamento. O nó se adapta à disseminação dos dados conforme o limite de energia pré-estabelecido. Quando o nó possuir dados para compartilhar, ele adverte os nós vizinhos. Ao receber uma advertência, outros nós transmitem a informação caso já a detenham, ou requisitam a informação de outros nós. Esta negociação garante que somente os pacotes úteis para um nó serão enviados a ele, conseguindo assim maior conservação de energia pela redução do número de pacotes enviados.

Difusão direcionada [31]	Neste protocolo uma requisição de sensoriamento é propagada pela rede na forma de um interesse, que representa uma consulta emitida pela aplicação observadora na estação base. Este interesse é difundido pela rede usando interações locais. Uma vez que um nó sensor que satisfaz a consulta (nó que coletará os dados) é alcançado, aquele nó começa a transmitir dados para a estação base, novamente usando interações locais. Os nós são endereçados por dados, e estes dados são nomeados por pares de atributo e valor. A ausência de noção de um identificador global (por exemplo, endereços estáticos pré-atribuídos) torna a difusão orientada eficiente para redes com mobilidade e também com um grande número de nós.

2.3.6.2 Roteamento hierárquico

Os protocolos desta categoria trabalham com agrupamento de nós (“*clusters*”) e comunicação multiponto dentro do escopo de cada grupo. Cada agrupamento possui um nó principal, denominado *gateway*, que faz a comunicação com os nós desse mesmo nível, com os outros grupos da rede [58]. Isto permite a operacionalização de RSSFs que possam monitorar uma grande área de interesse, e necessariamente possuam uma grande quantidade de nós sensores.

O objetivo do roteamento hierárquico é gerenciar eficientemente o consumo de energia nos nós sensores ao estabelecer comunicação com múltiplos trajetos direcionada à estação base, e ao executar agregação e consolidação de dados para reduzir o número de pacotes a serem transmitidos para esse nó principal. A Tabela 2.2 descreve exemplos de protocolos desta categoria.

Tabela 2.2 Exemplos de protocolos de roteamento hierárquico.

Protocolo	**Descrição**
LEACH (*Low-energy-adaptive clustering hierarchy*) [28]	Este protocolo define ciclos durante os quais são formados agrupamentos de nós (clusters), onde um nó é escolhido como cabeça de cada grupo (coordenador) pelos demais nós, em função do menor custo de comunicação. Conforme a sua arquitetura de consolidação, os nós secundários (não cabeças) transmitem seus dados para o nó cabeça do grupo. Este nó coordenador agrega dados de cada sensor e envia esta informação para a estação base, por onde estes serão entregues à aplicação observadora. Existem rodízios dos nó cabeças de grupos, visando balancear a carga de energia de todo o sistema. O nó coordenador de cada grupo deverá transmitir os seus dados consolidados para a estação base com um único salto, o que limita o tamanho da rede em função do alcance entre os nós cabeças de grupos e a estação base.
TEEN (*Threshold-sensitive energy-efficient sensor network*) [46]	Este protocolo assemelha-se ao LEACH no que se refere a formação de agrupamentos (*clusters*). No entanto, adota uma estratégia diferente para transmissão de dados. Há a definição de

	dois parâmetros, informados na fase de definição de um novo coordenador para o cluster: *Hard Threshold* (HT), que define o limite de valor até o qual o dado coletado continuamente deve ser transmitido, e *Soft Threshold* (ST), que é a tolerância mínima para o valor que fará com que o dado seja transmitido novamente. Se o valor do dado exceder HT pela primeira vez, ele é armazenado em uma variável e transmitido durante o intervalo de tempo alocado a transmissão do nó. Posteriormente, se houver coleta com valor monitorado que ultrapasse o valor armazenado fora dos limites definido em ST o nó transmite o dado imediatamente. O valor enviado é guardado para futuras comparações. O comportamento definido para este protocolo reduz o consumo de energia pelo nó sensor, por reduzir o número de transmissões que é realizado. Note que, se o valor do dado coletado nunca ultrapassar o limite definido para HT, o nó jamais iniciará a transmissão dos dados, o que pode, dependendo das características da aplicação observadora, provocar erros no seu processamento.
APTEEN (*Adaptive threshold-sensitive energy-efficient sensor network*) [45]	Este protocolo é baseado no protocolo TEEN, com algumas melhorias. Ao procedimento de rodízio de nós coordenadores dos clusters foram acrescentados os seguintes parâmetros: *Schedule* – reserva um canal TDMA para cada nó. Isto faz com que não haja colisões entre as transmissões dos nós de um cluster; e *CountTime* – define o tempo máximo entre duas comunicações sucessivas.

2.3.6.3 Roteamento baseado em localização

Neste tipo de protocolo, as informações sobre a localização dos nós sensores fazem parte dos dados do roteamento, de modo a ajudar na otimização do consumo de energia dos nós. Estas informações são utilizadas para calcular a distância entre dois nós, para que o consumo de energia possa ser determinado ou estimado. Por exemplo, se a região a ser monitorada é conhecida, a consulta pode ser disseminada somente para essa região específica, limitando e/ou eliminando o número de transmissões para fora do espaço definido. Os protocolos baseados em localização são ideais para redes *ad hoc* móveis, mas eles podem ser utilizados praticamente em qualquer RSSF (observando que alguns protocolos desenvolvidos para redes *ad hoc* sem fio, mas não orientados a otimização de consumo de energia, não são ideais para RSSFs [54]). A Tabela 2.3 descreve exemplos deste tipo de protocolo.

Tabela 2.3 Exemplos de protocolos de roteamento baseado em localização.

Protocolo	**Descrição**
MECN (*Minimum energy communication network*) [51]	Este protocolo calcula o melhor consumo de energia em redes de sensores, ao identificar regiões contendo nós que podem ser utilizados para transmissão de dados dos outros nós por gastarem menos recursos de energia do que se todas as transmissões

	fossem realizadas diretamente pelos nós origens. Esta identificação é feita utilizando-se GPS de baixo consumo. A idéia principal do MECN é encontrar um caminho que seja um subconjunto (sub-rede) com poucos nós que requerem pouca energia para transmissão entre dois nós específicos. Desta maneira, caminhos globais que consomem pouca energia são encontrados sem envolver todos os nós da rede. Isto é realizado usando uma pesquisa localizada para cada nó e considerando sua região identificada.
SMECN (Small minimum energy communication network) [37]	Este protocolo é uma extensão do MECN. O SMECN considera possíveis obstáculos entre dois pares de nós que fazem parte de um caminho global construído, e assim promove a sua otimização, tornando a sub-rede construída pelo SMECN menor que uma construída pelo MECN.

2.3.7 Gerenciamento de falhas

Em uma RSSF as falhas são possíveis e aceitáveis e a rede deve prover mecanismos para lidar com elas de maneira automática e natural. Os dispositivos sensores podem apresentar falhas de diversas naturezas, a exemplo de falta de energia, interferência de comunicação, quebra por vandalismo ou qualquer outro dano físico. Considerando a natureza da sua distribuição, que podem ser dispersos em grande quantidade em grandes áreas a serem monitoradas, a falha de alguns nós não deve causar mau funcionamento no resto da rede [1][32]. Isto pode ser garantido pela implementação de aspectos de tolerância à falhas.

Diferentes níveis de tolerância à falhas podem ser implementados a partir de diferentes algoritmos de controle, cada um mais adequado para determinada situação. Uma RSSF na qual é mais fácil se dar manutenção (e.g., uma rede no ambiente controlado de uma empresa), deve ter um nível de tolerância à falhas menor do que uma rede de sensores instalada num ambiente hostil, a exemplo de sensores deixados em locais externos, sujeitos a vandalismos ou convulsões da natureza.

O gerenciamento de falhas é um componente essencial em qualquer sistema de redes, e no cenário de RSSF torna-se crucial. Na maioria das aplicações em cima desta plataforma, a detecção de falhas é vital não apenas para garantir a tolerância a estas, mas também por questões de segurança, pois, ao ser detectada uma falha de comunicação, pode-se descobrir tratar-se de causa maliciosa, e assim possibilitar que a aplicação seja notificada de eventuais tentativas de ataque à rede.

2.4 Nós sensores inteligentes

Nesta seção serão descritas as características dos principais dispositivos de nós sensores inteligentes, desenvolvidos no contexto de pesquisas acadêmicas e comerciais.

2.4.1 Mica2 Mote

Os nós sensores Mica Motes [30] foram desenvolvidos pelos pesquisadores da Universidade de Berkeley. Esta é a terceira geração desta família de equipamentos modulares usados na construção de projetos de redes de sensores, e é comercializada pela empresa Crossbow [19].

Figura 2.2 Nó sensor Mica2.

O Mica2 segue o modelo básico de um nó sensor inteligente (Ver seção 2.2), possuindo unidade de processamento (baseada em Atmel ATmega 128L), unidade de armazenamento (4 KB de memória RAM, 128 KB de memória ROM e 512 KB de memória *Flash* para armazenamento de dados), unidade de rádio multicanal embutida no processador (taxas de 38,4 Kbps a 150 metros) e unidade de energia, esta composta por duas baterias “AA” (Figura 2.2). Uma característica da arquitetura Mica2 é a modularidade, permitindo, por exemplo, que se customize a categoria de sensoriamento que cada nó irá realizar, a partir do acoplamento de módulos de aquisição conforme a demanda da aplicação em uso.

Um sistema operacional denominado TinyOS (*Tiny Microthreading Operating System*) [30], específico para esta arquitetura de sensores, é utilizado no modelo aqui citado. O TinyOS, na realidade uma biblioteca API (*Application Programming*

Interface), foi desenvolvido num ambiente de programação semelhante à linguagem C, visando facilitar a elaboração de aplicações, ao levar a programação das rotinas de sensoriamento, comunicação e controle dos dispositivos para o patamar das linguagens de alto nível, além de disponibilizar uma série de procedimentos básicos para gerenciamento do nó sensor. Esse sistema tem como premissa a adequação ao poucos recursos disponíveis neste tipo de dispositivo (e.g., 8 Kbytes de memória para programas e 512 bytes de memória RAM). A linguagem de desenvolvimento do TinyOS, o “nesC”, é uma extensão da linguagem de programação C que incorporou os conceitos estruturais e o modelo de execução desta plataforma.

2.4.2 BEAN

O Projeto SensorNet, hospedado no Departamento de Ciências de Computação da Universidade de Minas Gerais (DCC/UFMG), tem como objetivo desenvolver o nó sensor (Figura 2.3) BEAN (*Brazilian Energy-Efficient Architectural Node*) [59]. Uma das premissas para o seu desenvolvimento é a utilização de componentes encontrados no mercado. O microcontrolador utilizado é um MSP430 de 16 bits, que tem baixo consumo de energia, com desempenho de 8 MIPS, e é equipado com um conjunto conversor analógico-digital. O rádio utilizado é o CC1000 (o mesmo que é utilizado em algumas montagens do Mica2 Mote). Este nó utiliza uma memória serial *Flash* externa (STM25P40) como memória secundária.

Figura 2.3 Placa do sensor BEAN.

O sistema operacional deste projeto também está sendo desenvolvido pelos pesquisadores da UFMG e foi batizado de YATOS (*Yet Another Tiny Operating System*). Ele é dedicado ao nó sensor BEAN e tem arquitetura dirigida a eventos. Uma das vantagens em relação ao TinyOS utilizado pelos Mica Mote é que o YATOS possui prioridade entre tarefas.

2.4.3 Medusa MK-2

O Medusa MK-2 [11] é um nó sensor desenvolvido no Laboratório de Engenharia Elétrica da Universidade da Califórnia (Figura 2.4).

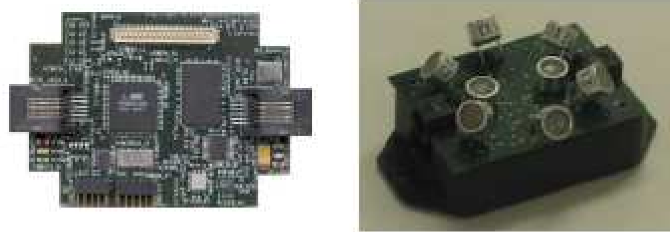


Figura 2.4 Nós Medusa MK-2 básico e Medusa MK-2 com placa de ultra-som.

Um nó Medusa MK-2 usa dois microcontroladores. O primeiro é um Atmel ATMega128L com 32 KB *Flash* e 4 KB de RAM rodando a 4 MHz. Este microcontrolador é dedicado a tarefas que não exigem muito poder computacional, como a interface com rádio e a coleta de dados do sensor. O segundo processador, um Atmel AT91FR4081 processa as tarefas mais pesadas, demandadas pela aplicação instalada. Este processador roda a 40 MHz e tem 1 MB de memória *Flash*, além de 136KB of RAM.

O seu rádio, um TR1000, possui uma potência de transmissão de 0,75 mW, com alcance de até 20 metros. A taxa de transferência pode variar de 2,4 Kbps a 115 Kbps. O consumo de energia quando o nó está totalmente operacional é de 200mW, e de 100µW quando está em modo de espera.

2.4.4 Smart Dust

O Smart Dust [60] é um projeto desenvolvido pela Universidade de Berkeley. Este projeto tem por objetivo reduzir o tamanho dos nós sensores para que estes apresentem as dimensões de um grão de poeira, ou seja, um cubo de aproximadamente um milímetro (Figura 2.5), daí o seu nome. Os componentes disponíveis para este dispositivo serão um sensor, uma bateria, um circuito analógico, um dispositivo de comunicação ótica bidirecional e um microprocessador programável.

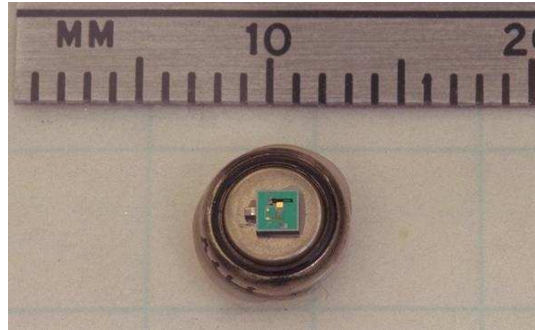


Figura 2.5 Nó sensor Smart Dust.

A escolha do dispositivo de comunicação ótica se deu pelo fato da comunicação através de transceptores de rádio freqüência (RF) não ter se mostrada adequada para os nós deste tipo, devido a diversos aspectos, tais como: as antenas seriam muito grandes para os Smart Dust, e o consumo de energia para a comunicação via rádio seria alto para a disponibilidade do nó.

2.5 Considerações finais

Este capítulo apresentou conceitos básicos relativos a redes de sensores sem fio e os seus componentes, e explorou algumas das características mais importantes ligadas a esta tecnologia. As RSSF têm potencial de utilização por um amplo espectro de aplicações, desde domínios civis a militares. Contudo, como foi visto na exposição de cada aspecto ligado a este tipo de rede, a sua operacionalização encontra várias limitações ligadas a seus recursos, principalmente aquelas relacionadas ao armazenamento e consumo de energia. Também verificamos aqui que eficiência na utilização do recurso de energia é um objetivo crítico na elaboração de protocolos de comunicação voltados para as RSSFs, visando aumentar o tempo de vida dos dispositivos que implementam os nós sensores.

Capítulo 3
PROCESSAMENTO DE CONSULTAS

3.1 Introdução

Sistemas gerenciadores de bancos de dados (SGBDs) consideram que os dados a serem manuseados são armazenados em algum meio. A propriedade de existência prévia da persistência dos dados é intrínseca aos SGBDs, e é naturalmente adequada a aplicações que utilizam consultas freqüentes em um grande conjunto de dados armazenados, enquanto que atualizações ou inserção de novos dados não são realizadas continuamente.

Em contraste, surge um novo conceito para sistemas de gerenciamento de dados, denominado DSMS: *Data Stream Management Systems* [5], ou Sistemas de Gerenciamento de Fluxos de Dados (SGFDs). Define-se fluxo como uma corrente de dados, tal como fila de registros, sendo gerada continuamente por uma aplicação ou dispositivo, sem definição estática de início ou fim desse fluxo. Em contraste com aplicações que utilizam SGBDs, as aplicações voltadas para geração de fluxos de dados realizam continuamente inserção de novos dados no final do fluxo que está sendo produzido, inexistindo atualizações sobre este. Enquanto isso, aplicações de consultas podem executar operações seqüenciais sobre os novos dados gerados.

Independente da abordagem aplicada, se SGBD ou SGFD, as interações do usuário com um gerenciador de dados deve ocorrer através de uma consulta declarada. Por isto, tal gerenciador deve oferecer mecanismos eficientes para obter dados de suas fontes através de seu processador de consultas.

O objetivo deste capítulo é caracterizar o que é um processamento de consultas nos SBGDs, e em especial, ao processamento de consultas contínuas sobre fluxos de dados implementado em redes de sensores sem fio (RSSFs). Na Seção 3.2 são ditados conceitos relacionados com processamento de consultas em bancos de dados. A Seção 3.3 apresenta aspectos relacionados aos sistemas de gerenciamento de fluxos de dados. A Seção 3.4 estende os conceitos da seção

anterior, a partir da demonstração de conceitos relacionados com consultas contínuas sobre fluxos de dados e mostra exemplos de desenvolvimento de aplicações sobre esta tecnologia, e a Seção 3.5 categoriza processamento de consultas em redes de sensores sem fio e traz alguns exemplos de linguagens desenvolvidas para este tipo de tratamento.

3.2 Processamento de consultas em bancos de dados

Num processador de consultas dois grupos de atividades se destacam: a interpretação (ou compilação) e a execução [21][22]. Nas atividades de interpretação/compilação são definidas três tarefas principais: a análise das instruções em linguagem declarativa, a geração do plano lógico e a geração do plano de execução. A análise supracitada consiste em construir uma árvore de análise para representar a estrutura da consulta. A geração do plano lógico converte a árvore de análise num plano de consulta inicial, em formato algébrico. Na geração do plano de execução, operações algébricas (por exemplo, junções), são transformadas em operações físicas (por exemplo, *hash join* ou *merge join*). Nas atividades de execução, as operações do plano de execução são processadas e os resultados da consulta são entregues à aplicação usuária.

Geralmente, os possíveis planos de execução gerados para uma consulta podem ser numerosos e dois fatores influenciam essa geração: a quantidade de operações na consulta e o número de artifícios que podem ser usados para analisar cada operação. Exemplos dessas estratégias podem ser vistos em [26]. O número de planos de execução ainda pode ser incrementado se a consulta for executada em um ambiente distribuído porque devem ser considerados outros fatores relacionados com a localização dos dados. Descobrir o melhor plano de execução de consultas não é uma tarefa trivial, e, deste modo, técnicas de otimização destes procedimentos têm sido debatidas [14].

3.3 Sistemas de gerenciamento em fluxos de dados

Fluxos de dados (ou *data streams*), no contexto corrente, é definido como uma seqüência de itens, classificada conforme a ordem destes itens.

Os novos desafios em utilizar os sistemas gerenciadores de fluxos de dados (SGFDs) em vez de SGBDs originam-se de duas diferenças fundamentais:

(i) um SGFD deve manusear múltiplos fluxos de dados contínuos, ilimitados quanto ao tempo de geração, e obtidos em tempo real de sua geração;

(ii) por causa da natureza contínua dos dados, um SGFD tipicamente deve ser capaz de executar consultas contínuas de longa execução, das quais é esperada a produção de resultados também respectivamente de forma contínua, e também sem tempo para serem concluídos.

O tratamento de dados como um fluxo é apropriado quando os dados são gerados continuamente a partir de novas inserções [48] e é desnecessário ou impraticável armazená-los sem um tratamento prévio. Como exemplo, citamos aplicações que geram registros de transações de mercado financeiro (*tickers*), medições de tráfico e performance de redes, registros de rastreamento em navegação de páginas Web, bilhetagem de chamadas de centrais telefônicas, mensagens de e-mails e dados coletados de dispositivos sensores [24].

Os SGBDs atuais não possuem funções específicas para realizar o gerenciamento de fluxos de dados. Assim, neste contexto, no máximo os SGBDs são utilizados para armazenamento, após as informações passarem por aplicações especializadas, pois muitos aspectos ligados a gerenciamento de dados precisam ser reconsiderados na presença destes fluxos.

Dentre trabalhos recentes de pesquisadores acerca do processamento de fluxos de dados, destacamos o projeto STREAM (*STanford stREam datAManagement*) [6][3], da Universidade de Stanford, que é dirigido às demandas impostas pelo gerenciamento de fluxos de dados e suas técnicas de processamento. Este projeto, um protótipo de um SGFD, teve o foco inicialmente nas seguintes funções:

- possuir uma interface flexível para obter e armazenar fluxos de dados (ou sinopses de fluxos), como parte de um gerenciador hierárquico de armazenamento;

- ser um processador de consultas contínuas especificadas a partir de uma linguagem declarativa padrão SQL;
- disponibilizar uma interface de programação para o desenvolvimento de aplicações (*Application Programming Interface* - API) para o controle de consultas contínuas e o recebimento de seus resultados.

3.4 Consultas contínuas

A proliferação da Internet, a Web, e redes de sensores catalisaram o desenvolvimento de aplicações que tratam dados como fluxos em vez de considerá-los armazenados. Já foram propostos sistemas para tratar as necessidades de processamento de consultas sobre fluxos de dados [13]. Nesse contexto, foi sistematizado o conceito de consulta contínua, que é a consulta definida sobre um fluxo contínuo de dados de entrada e que gera resultados continuamente, conforme os parâmetros de projeção e seleção definidos, além de outros eventuais filtros.

O acesso a dados em redes de sensores sem fio pode se dar através de sistemas de gerenciamento baseados em consultas contínuas. Como exemplo, citamos os projetos descritos a seguir. O NiagaraCQ [15][39] permite múltiplas consultas relacionadas compartilhando o processamento de partes comuns em seus planos de execução. Já o mecanismo CACQ (*Continuously Adaptive Continuous Queries*) [61] estende a noção de consultas contínuas ao habilitar facilidades de adaptação conforme a carga de processamento.

3.5 Consultas em RSSF

Em relação aos sistemas computacionais tradicionais, onde os dados são armazenados previamente para que sejam habilitados acessos a estes, as redes de sensores têm as suas peculiaridades quando da geração de seus dados e a sua disponibilização para aplicações de consultas sobre estes fluxos. De certa forma, uma rede de sensores sem fio pode ser comparada a um banco de dados distribuído [7][16], quando disponibiliza uma camada de processamento de consultas para aplicações usuárias obterem dados produzidos pelos sensores

disseminados. E pela natureza deste tipo de rede, formas mais adequadas de realizar consultas sobre seus dados podem ser adotadas.

Como exemplo de classificação das formas de consultas que são utilizadas em redes de sensores, a seguir são descritos cinco tipos [67] que tipicamente são implementados por aplicações observadoras nestas plataformas:

(i) Consultas Históricas: os dados gerados pela rede são consolidados, e armazenados sob forma de informações históricas em sistemas de bancos de dados tradicionais, para permitir consultas posteriores. Um exemplo seria uma aplicação que necessitasse de informações sobre a média de precipitação chuvosa sobre uma dada região, num determinado período;

(ii) Consultas Instantâneas: as consultas são preparadas para obter os dados na rede em um momento definido, tal como uma fotografia da situação num ponto específico da região monitorada. Um exemplo deste tipo de consulta seria uma aplicação que necessitasse obter os valores atuais de umidade do ar em todos os sensores de uma determinada região;

(iii) Consultas de Longa Duração: as consultas são preparadas para obter continuamente dados da rede durante um período de tempo definido. Um exemplo seria uma demanda para se informar os valores médios de temperatura, a cada hora, numa dada região, sempre a partir de um determinado horário do dia;

(iv) Consultas disparadas por eventos: os dados são coletados por procedimentos executados na ocorrência de certos eventos pré-definidos, que podem ser relacionadas à lógica da aplicação ou a eventos no ambiente monitorado [40]. Um exemplo é uma aplicação de monitoração ambiental [44], que deve medir a temperatura de ninhos num determinado local, sempre que fosse identificada a presença dos pássaros;

(v) Consultas em intervalos multidimensionais: envolve mais de um atributo de dados do nó sensor na especificação do filtro da consulta [38]. Por exemplo, determinar as coordenadas dos sensores que apresentarem

valores de leitura de fumaça maior que 5% e que tenham leituras de temperatura maior que 30 graus Celsius.

Ao mapear as origens de dados de uma RSSF como uma abstração de um banco de dados [25], tornou-se natural a busca por ferramentas para extração de informações que possuam interfaces semelhantes aos modelos tradicionais. A utilização de linguagens de alto nível para acesso aos dados dispensa o usuário de conhecer detalhes acerca de como esses dados estão armazenados, além de prover formas otimizadas para acessá-los [65].

Consultas escritas em linguagens declarativas, baseadas no padrão SQL, tornam transparente ao usuário os detalhes físicos da execução de consultas, como a geração de planos de consulta e otimização, ou a forma de distribuição das consultas através dos nós sensores envolvidos. A Tabela 3.1 mostra exemplos de linguagens declarativas propostas junto a estratégias de consultas no escopo de redes de sensores e processamento de fluxos de dados.

Tabela 3.1 Exemplos de linguagens de consulta em RSSF

Nome	**Descrição**	**Projeto associado**
Acquisitional Query Language [43]	Foi proposta para dar suporte ao processador de consultas (*Acquisitional Query Processing* - ACQP) do projeto TinyDB. Segue o padrão SQL, com extensões específicos que atendem às estratégias deste projeto. Nessa abordagem, existe apenas uma tabela virtual, denominada *sensors*, que é fisicamente particionada, representada pelos sensores distribuídos pela rede. Os registros nesta tabela são materializados (ou seja, adquiridos) somente quando requeridos para satisfazer a consulta, e são geralmente armazenados por um breve período de tempo ou entregues para transmissão pela rede. As consultas suportam cláusulas de projeção, seleção, agrupamento, junção e agregação, com sintaxe e semântica semelhantes ao SQL.	TinyDB [42]
CQL – *Continuous Query Language* [4]	Orientada ao padrão SQL, foi construída para dar suporte a consultas em fluxos de dados e também a tabelas físicas, definida nesta linguagem como sendo relações. Disponibiliza operadores para realizar conversões entre relações e fluxos de dados, conforme as classes a seguir: - s*tream-to-relation*: produz uma relação a partir de um fluxo de dados, utilizando conceitos de *sliding windows*[23] *sobre o fluxo*; - *relation-to-relation*: produz uma relação a partir de uma ou mais relações (como no SQL padrão); - *relation-to-stream*: produz um fluxo de dados a partir de uma relação.	STREAM [6][3]

GSQL – *Gigascope Query Language* [18]	Baseada no padrão SQL, é orientada puramente para fluxos de dados: todas suas entradas e saídas são fluxos de dados, e as operações são feitas sobre janelas temporais (*sliding Windows)* [23]. Suporta operações de seleção, junção, agregação e combinação de fluxos. Todas as consultas operam sobre fluxos de dados, que podem ser de um dos seguintes tipos: - *Protocols*: um fluxo de dados gerado após interpretar uma seqüência de pacote de dados que chegam ao Gigascope. Este é interpretado como uma coleção de campos; - *Streams*: é a saída de uma consulta do Gigascope. Os campos destas tuplas são empacotados de uma forma padrão.	Gigascope [18]
SNQL – *Sensor Network Query Language*	Extensão à linguagem SQL proposta no contexto da presente dissertação (Detalhes no Capítulo 4), que dá suporte aos conceitos em foco neste trabalho. Adiciona cláusulas específicas para tratamento pelas estratégias de otimização do uso de recursos de energia e memória dos nós sensores.	PDCRS

3.6 Considerações finais

Este capítulo apresentou considerações acerca de processamento de consultas, focando as características e estratégias voltadas ao processamento de consultas em ambiente de RSSFs. Os conceitos abordados neste capítulo serão utilizados como base para a especificação do mecanismo de distribuição de consultas em RSSFs, a ser exposta com tema do Capítulo 5 desta dissertação.

Capítulo 4
SNQL – SENSOR NETWORK QUERY LANGUAGE

4.1 Introdução

No escopo da presente dissertação é proposta[5] a SNQL (*Sensor Network Query Language*), linguagem que dá suporte ao mecanismo de distribuição de consultas proposto no Capítulo 5 deste trabalho. A SNQL é uma extensão ao padrão SQL, acrescida de cláusulas específicas para processamento de consultas em redes de sensores sem fio (RSSF). O seu interpretador foi implementado como parte do processador de consultas do mecanismo PDCRS, objeto desta dissertação.

As cláusulas chaves do comando padrão SQL para manipulação de dados (DML – Data Manipulation Language) foram mantidas (*Select, From, Where, Group By e Having*), enquanto que outras, especialmente adequadas às necessidades do processamento de consultas em RSSF, foram incluídas. Tais cláusulas dão à aplicação flexibilidade ao permitir que sejam definidos intervalos de tempo para captação, empacotamento e envio dos dados, e especificação da validade e a freqüência de submissão da consulta. Os comandos relacionados com definição de dados (DDL – Data Definition Language), se comparados àqueles do padrão SQL, sofreram adaptações para tratar especificamente o objeto foco das RSSFs, que são os dispositivos sensores envolvidos.

Nas seções a seguir, será apresentado o detalhamento acerca das sintaxes das declarações SNQL, por categoria de utilização.

4.2 Componente DML

O comando iniciado pela cláusula *SELECT* é utilizado para especificar consultas, e tem a seguinte sintaxe geral mostrada na Figura 4.1, abaixo.

---

[5] A SNQL foi proposta conjunta do grupo de pesquisa de RSSF, no Mestrado em Informática Aplicada, da UNIFOR, do qual fazemos parte. A implementação do seu interpretador foi parte deste nosso trabalho de mestrado.

```
SELECT  {<lista de projeção>}
FROM  {<lista de categorias de sensores>}
[WHERE  {<filtro de condições>}]
[GROUP BY {<atributos de agrupamento>}
[HAVING {<filtro para agrupamento>}]]
TIME WINDOW <tempoconsulta> | CONTINUOUS
[DATA WINDOW <número de coletas>]
SEND INTERVAL <intervalo entre transmissões>
SENSE INTERVAL <intervalo entre coletas >
[SCHEDULE  <numero de execuções> [{tempo de início}] | CONTINUOUS]
```

Figura 4.1 Sintaxe geral componente DDL da SNQL

As novas cláusulas introduzidas pela SNQL para uso específico em consultas para aplicações de RSSFs são:

- *Sense Interval*: define o intervalo de tempo entre duas coletas de dados do ambiente monitorado. Quanto maior o valor especificado nesta cláusula, menor será a precisão do resultado, pois menos captações serão realizadas no decorrer da consulta. Por outro lado, valores muito reduzidos para o *Sense Interval* irão gerar um grande volume de dados tratados pelos nós sensores, resultando em consumos maiores de energia, tanto para a captação como para o processamento e envio dos dados. Exploramos este conceito em [8][9][11].
- *Send Interval*: informa aos sensores o intervalo de tempo que deve ser completado entre dois procedimentos de envio de dados de resultados. Durante este intervalo, os dados captados do ambiente estarão sendo armazenados em estruturas mantidas nos sensores, podendo sofrer agregações, até o próximo período de envio, quando os dados resultantes serão empacotados e enviados ao nó pai. Este conceito também é explorado em [9].
- *Time Window*: esta cláusula define quanto tempo uma consulta deverá permanecer ativa na rede, após a sua submissão. Devido à dificuldade em se manter o sincronismo no envio de dados em uma RSSF, considera-se que a estação base deterá o controle sobre esta informação, ativando um temporizador de início da consulta e, quando o valor da *Time Window* for alcançado, esta estação base finaliza o processamento, mesmo que os

demais nós ainda não tenham enviado os pacotes remanescentes a serem processados. Para os nós sensores esta informação será útil para tomarem conhecimento do momento adequado de suspenderem a captação de dados e, conseqüentemente, o seu processamento e transmissão para a estação base. Opcionalmente, a aplicação pode considerar um período de tolerância, além do valor especificado na *Time Window*, como forma de permitir a recepção e processamento dos dados que ainda estejam trafegando na rede após a *Time Window* ter sido alcançada. Este mesmo conceito é explorado em [7].

- *Data Window:* complementa a definição de tempo de ativação de uma consulta, ao impor limite à quantidade de coletas a serem realizadas.
- *Schedule*: define a freqüência de submissão da consulta, além de permitir opções de agendamento. O usuário pode definir que uma mesma consulta seja submetida várias vezes à aplicação, por exemplo: diariamente, semanalmente ou continuamente. Uma consulta especificada com agendamento contínuo estará continuamente sendo ressubmetida aos sensores, após a finalização de cada instância da consulta, sem a necessidade de interação do usuário. Dependendo dos parâmetros do Schedule, algumas informações extras serão necessárias para o agendamento da execução, como por exemplo, hora de execução da consulta ou dia da semana.

A Tabela 4. apresenta o sumário das funcionalidades das cláusulas da SNQL. Os parâmetros de tempo de vida da consulta (*Time Window*) e intervalos de sensoriamento (*Sense Interval*) e envio de dados (*Send Interval*), por padrão, serão dados em milissegundos, mas também podem ser passados em unidades maiores (segundos, minutos, etc.), se informadas, em seguida ao valor pretendido, as constantes referentes à unidade desejada[6].

[6] Unidades de tempo: MILISECOND, SECOND, MINUTE, HOUR, DAY, MONTH e YEAR.

Tabela 4.1 Descrição[7] das cláusulas da SNQL.

Cláusulas	Descrição
SELECT {<lista de projeção>}	<listaprojeção>: lista de atributos de categorias de sensores ou itens constantes, dos quais os valores farão parte do resultado da consulta.
FROM {<lista de categorias de sensores>}	<lista de categorias de sensores>: lista das relações que serão consultadas, representando as categorias de sensores distribuídos.
[*WHERE* {<filtro de condições>}]	<filtro de condições>: conjunto de condições da cláusula de seleção da consulta.
[*GROUP BY* {<atributos de agrupamento>} [*HAVING* {<filtro para agrupamento>}]]	<atributos de agrupamento>: conjunto de atributos pelos quais o agrupamento e agregação dos resultados da consulta serão realizados. <filtro para agrupamento>: conjunto de condições pelas quais o agrupamento será realizado.
TIME WINDOW <tempoconsulta> | CONTINUOUS	<tempoconsulta>: tempo de vida da consulta no sistema, em milissegundos, ou outra unidade especificada. Obs.: a cláusula *TimeWindow* tem preferência sobre a cláusula *Data Window*. *CONTINUOUS:* parâmetro que especifica uma consulta sem tempo de vida especificado.
[*DATA WINDOW* <número de coletas>]	<número de coletas>: define o tempo de vida da consulta a partir da especificação do número máximo de coletas que o nó sensor irá realizar.
SEND INTERVAL <intervalo entre transmissões>	<intervalo entre transmissões>: tempo entre cada transmissão de dados pelo nó sensor.
SENSE INTERVAL <intervalo entre coletas >	<intervalo entre coletas>: tempo entre cada atividade de sensoriamento (coleta de dados) no ambiente.
[*SCHEDULE* <numero de execuções> [{tempo de início}] | CONTINUOUS]	<numero de execuções>: número de vezes em que a consulta será submetida pela estação base à rede de sensores. Cada vez tem o tempo de vida definido em *Time Windows* ou *Data Window*. <tempo de início>: define a data e horário que cada execução começará. O parâmetro *CONTINUOUS* indica que a consulta será ressubmetida sempre que o tempo de vida de uma submissão anterior for atingido.

4.3 Componente DDL

A criação de relações define visões lógicas sobre os dados gerados continuamente pelos nós sensores de uma RSSF, sendo essas definições armazenadas no nó central, relacionados com a visão definida. O gerenciamento dessas visões é papel

[7] Convenções: “{}”: parâmetros obrigatórios; “[]”: parâmetros opcionais; “<>”:uma expressão ou lista de parâmetros; e, “|” especifica que um ou outro parâmetro, somente, deve ser fornecido.

do PDCRS, e faz parte das funções de manipulação do esquema global de relações deste mecanismo.

Este tópico mostra como definir, utilizando comandos da linguagem SNQL, os esquemas das relações que mapeiam as categorias de sensores, criando assim um esquema global de relações de categorias de sensoriamento. Também aqui será explicado como as relações são criadas e modificadas, e as funcionalidades disponíveis para controlar os dados coletados a partir do conjunto de sensores conforme suas categorias.

As declarações para definição de dados (*Data Definition Language* - DDL) contêm os componentes da linguagem para criação, alteração e exclusão de esquemas lógicos das relações geradas pelos sensores.

A criação de categorias de sensores (*CREATE SENSORS*) define visões lógicas sobre os dados gerados continuamente pelos nós sensores de uma mesma categoria de sensoriamento na rede monitorada, sendo essas definições armazenadas no nó central e nos nós sensores respectivos. O gerenciamento dessas visões é papel do PDCRS, e faz parte das funções de manipulação do esquema global de relações deste motor. A Figura 4.4 apresenta um sumário das sintaxes dos comandos para gerenciamento dos esquemas (DDL) tratados pelo PDCRS.

```
CREATE SCHEMA <schemaname> ;
CREATE SENSORS <sensorcategname> (<fieldname> <fieldtype>[, ...]);
DROP SCHEMA <schemaname>;
DROP SENSORS <sensorcategname>;
USE <schemaname>;
IF [NOT] EXISTS {SCHEMA|SENSORS } <componentname> <declaration>;
```

Figura 4.2 Sintaxes para gerenciamento de esquemas

Os subtópicos a seguir expõem as sintaxes relacionadas com cada tipo de declaração para definição de dados, dentro dos esquemas utilizados pela SNQL.

4.3.1 Definição do esquema global

O esquema global é o catálogo utilizado pelo PDCRS para manter as definições das relações mapeadas referentes às categorias de sensoriamento de sensores distribuídos na rede. A criação de um esquema global novo, utilizando a SNQL,

começa com o comando *CREATE SCHEMA*, onde é especificado o nome do arquivo que conterá o catálogo. Neste arquivo, em formato XML[8], serão gravadas as definições geradas com os demais comandos de definição de dados.

Declaração de criação do esquema global

CREATE SCHEMA <schemaname> ;

<schemaname> ::= Nome do repositório de esquemas de relações a ser criado. Este nome será utilizado na denominação do arquivo XML correspondente.

Escolha do esquema a ser utilizado

USE <schemaname>;

<schemaname> ::= Nome do esquema de relações a ser "aberto" para utilização. A partir da inserção desta declaração no script, todas as declarações subseqüentes utilizarão o esquema aqui informado.

Exclusão de um esquema global

DROP SCHEMA <schemaname>;

<schemaname> ::= esquema a ser excluído, caso não exista nenhum grupo sensor definido dentro deste.

O Exemplo a seguir cria o esquema como o nome GlobalSchema:

```
CREATE SCHEMA GlobalSchema;
```

Se a definição um esquema global não for mais necessária, poderá ser removida utilizando o comando *DROP SCHEMA*. Por exemplo:

```
DROP SCHEMA GlobalSchema;
```

O exemplo acima remove o esquema global, suprimindo seu arquivo relacionado.

4.3.2 Definição de categoria de sensores

Para definir uma categoria de sensores utiliza-se o comando *CREATE SENSORS*. Neste comando são especificados o nome do novo grupo sensor, os nomes das colunas, e o tipo de dado de cada coluna. Por exemplo:

---

[8] XML (Extensible Markup Language) [64]: padrão de codificação de dados em que as estruturas e os

```
CREATE SENSORS minha_categoria_sensor (
    primeira_coluna text,
    segunda_coluna integer);
```

Este comando cria uma categoria de sensores chamado *minha_categoria_sensor* contendo duas colunas. A primeira coluna chama-se *primeira_coluna*, e possui o tipo de dado *text*; a segunda coluna chama-se *segunda_coluna*, e possui o tipo de dado *integer*. Deve ser observado que a lista de colunas é envolta por parênteses, e os elementos da lista são separados por vírgula.

Normalmente são dados nomes para as categorias de sensores e para as colunas são condizentes com as informações que são geradas pelos dispositivos sensores respectivos que estão sendo mapeados conforme essa relação. Sendo assim, a seguir é mostrado um exemplo mais próximo da realidade:

```
CREATE SENSORS temperatura (
    RegionId   integer,
    collectedValue float,
    timestamp  time);
```

Se a definição uma categoria de sensor não for mais necessária, poderá ser removida utilizando o comando *DROP SENSORS*. Por exemplo:

```
DROP SENSORS temperatura;
```

Tentar remover uma categoria de sensor não existente no esquema global produz um erro. Tentar criar uma categoria de sensor já existente no esquema global também produz um erro. Entretanto, há o recurso para tentar remover uma relação condicionalmente antes de criá-la, evitando assim um erro na execução de um lote de comandos SNQL, conforme descrito em na subseção 4.4, a seguir.

Definição de categoria de sensores

CREATE SENSORS <sensorscategname> (<fieldname> <fieldtype>[, ...]);

<sensorscategname> ::= Nome para a categoria de sensoriamento dentro do esquema global;

<fieldname> ::= Nome de um atributo da lista de atributos da categoria de sensor;

---

dados armazenados ficam no mesmo documento.

<fieldtype> ::= Tipo de um atributo da lista de atributos da categoria de sensor.

Exclusão da definição de uma categoria de sensores

DROP SENSORS <sensorscategname>;

<sensorscategname> ::= categoria de sensores a ser removido do esquema global.

4.4 Declarações para execução em lote

O PDCRS provê recursos para executar série de declarações a partir de um lote de comandos, permitindo a aplicação usuária controlar o fluxo de execução entre cada um dos comandos processados. Dentro de um lote de comandos pode conter mais de um comando DDL, e no máximo um comando DML. Um lote de comandos deve iniciar com o comando para escolha do esquema a ser utilizado (comando *USE*).

Execução condicional de declaração

O PDCRS disponibiliza um comando para execução condicional de outros comandos, o que flexibiliza a execução de lotes de comandos ao permitir testar a existência das entidades manipuladas por este mecanismo. A sintaxe apresentada a seguir mostra o formato desta declaração:

IF [*NOT*] *EXISTS* {*SCHEMA*|*SENSORS*} <componentname> <declaration>;

<componentname> :: nome do componente (esquema ou categoria de sensores) a ter a sua pré-existência testada;

<declaration> :: declaração DML ou DDL a ser executada, caso a condição de pré-existência seja satisfeita.

Os dois exemplos a seguir ilustra o uso do comando condicional:

Exemplo 1: se existe a categoria de sensores *temperatura*, faz a sua remoção do esquema.

```
IF EXISTS SENSORS temperatura DROP SENSORS temperatura;
```

Exemplo 2: se não existe a definição da categoria *temperatura*, faz a sua criação no esquema.

```
IF NOT EXISTS SENSORS temperatura CREATE SENSORS temperatura(
```

```
RegionId   integer, collectedValue float, timestamp  time);
```

4.5 Funções de agregação

A SNQL dá suporte a funções de agregação, que podem fazer parte da cláusula *SELECT*, da mesma forma do SQL padrão. Estendendo estas características, o tratamento das funções de agregação na SNQL pode definir agregação em rede.

A Tabela 4.2 apresenta as funções de agregação correntemente implementadas no interpretador SNQL, que é um componente do PDCRS. Este mecanismo, por definição, permite a eventual evolução no conjunto de funções de agregação disponíveis.

Tabela 4.2 Funções de agregação da SNQL.

Função	**Especificação**	**Tipo de Agregação**
COUNT()	Número de coletas realizadas.	Distributiva
SUM()	Somatório dos valores do atributo especificado para cada coleta realizada.	Distributiva
MAX()	Maior valor entre todos os valores do atributo especificado, nas coletas realizadas.	Distributiva
MIN()	Menor valor entre todos os valores do atributo especificado, nas coletas realizadas.	Distributiva
AVG()	Média aritmética entre os valores do atributo especificado, nas coletas realizadas.	Algébrica

Cada função da tabela supracitada, por definição, estabelece uma estratégia para sumarização dos valores de entrada nos nós distribuídos, e como esses resultados serão novamente consolidados na estação base, o que permite categorizar as funções conforme o seu tipo de agregação. Esta classificação é melhor discutida em [27]. Conforme a função, os tipos de agregação podem ser:

- Distributiva: definida por função que, quando aplicada sobre vários conjuntos (nós sensores) e somada (na estação base), tem o mesmo resultado de quando é aplicado no total. Contagens (*Count*), somas (*Sum*), mínimo (*Min*) e máximo (*Max*) são exemplos deste tipo;
- Algébrica: é a função que pode ser obtida através de M agregações distributivas aplicadas a uma função F(M). Média (*Avg*) é um exemplo

de função de agregação algébrica. No caso da média, a função F calcula a soma e a contagem de cada subconjunto de valores (nós sensores) e esses valores são levados à estação base, que calculará a média geral baseada no montante das somas dividida pela soma das contagens. Outro exemplo de função de agregação algébrica é a função de cálculo de Desvio Padrão.

- Holística: é a agregação mais complexa, que não consegue ser dividida em partes ou em agregações distributivas, exigindo que todos os valores coletados em nós distribuídos sejam enviados para consolidação no nó central, não sendo viável a sua utilização na agregação em rede. Um exemplo deste tipo de agregação é a função de cálculo de Mediana.

4.6 Considerações finais

Este capítulo apresentou a SNQL, proposta de linguagem de consultas baseada no padrão SQL, como alternativa para a especificação de consultas declarativas com cláusulas específicas para aplicações de RSSFs. Maiores detalhes acerca das construções das estrutura sintática desta linguagem são encontrados no Apêndice D. Esta linguagem dará o apoio à proposta núcleo desta aplicação, descrita no Capítulo 5.

Capítulo 5

PDCRS: UM MECANISMO PARA PROCESSAMENTO DISTRIBUÍDO DE CONSULTAS EM RSSF

5.1 Introdução

O processamento de uma consulta pode ser visto como o conjunto de atividades envolvidas na extração de informações de uma ou mais fontes de dados. Conforme a ambientação onde a consulta for executada, a fonte de dados pode ser desde arquivos de documentos textos até sistemas gerenciadores de bancos de dados tradicionais. Neste trabalho, serão focadas como fontes das consultas um conjunto de fluxos de dados proveniente de sensores distribuídos e classificados conforme a categoria das informações presentes nestes dados. Outrossim, cada nó sensor será considerado tanto como uma fonte remota de dados, como também um dispositivo capacitado a processá-los. A estratégia aqui proposta adota o modelo relacional de banco de dados aplicado às necessidades do tipo de rede aqui estudado, aproximando-se do conceito de ambiente de banco de dados distribuídos. Isto permite uma melhor organização lógica das informações, oferece suporte a linguagens de consulta baseadas em SQL (*Structured Query Language*), além de dar maior flexibilidade à aplicação, capacitando-a a responder consultas do tipo *ad hoc* [62][67].

O presente trabalho apresenta um mecanismo de processamento de consultas em redes de sensores, denominado PDCRS (Processamento e Distribuição de Consultas em Redes de Sensores), que tem como objetivo executar consultas em RSSF de forma distribuída nos nós da rede. O PDCRS tem como ponto de partida o processamento de filtros dinâmicos, expressos sob forma de consultas. Essas consultas, para serem injetadas no sistema, deverão passar por etapas de preparação de estruturas que permitirão a sua distribuição e gerenciamento, desde a sua emissão pela aplicação usuária até o recebimento dos resultados gerados pelos nós sensores, de forma contínua. A estratégia aqui apresentada visa oferecer recursos para facilitar a implementação das funções de injeção, distribuição e gerenciamento dos resultados de consultas contínuas, a partir da utilização de

critérios de distribuição baseados em agrupamento por categoria de sensores ou por sua região de localização.

O objetivo deste capítulo é descrever a proposta do PDCRS, e está dividido conforme descrito a seguir. A Seção 5.2 descreve as etapas da operação do PDCRS, a partir da ilustração em um cenário exemplo. Na Seção 5.3 são tecidas as considerações acerca do mecanismo apresentado.

5.2 Processamento distribuído de consultas em RSSF

O PDCRS subdivide as atividades envolvidas com o processamento de uma consulta nas cinco etapas descritas a seguir:

i) Interpretação ou *parsing* (avaliações léxica, sintática e semântica do código das consultas);

ii) Decomposição (otimizações e fragmentações em subconsultas, conforme as categorias de sensores envolvidos);

iii) Distribuição das subconsultas (disseminação dos fragmentos pela RSSF);

iv) Processamento das subconsultas (fragmentos); e;

v) Pós-processamento (consolidação dos resultados).

A estação base da RSSF é responsável por executar as fases (i), (ii), (iii) e (v), citadas acima. Nos nós sensores é executada a fase (iv), que é o processamento dos fragmentos distribuídos da consulta original. O PDCRS possui um conjunto de métodos que podem ser invocados pela aplicação usuária, dentro do esquema pré-definido que visa concretizar as funcionalidades referentes às cinco etapas citadas acima.

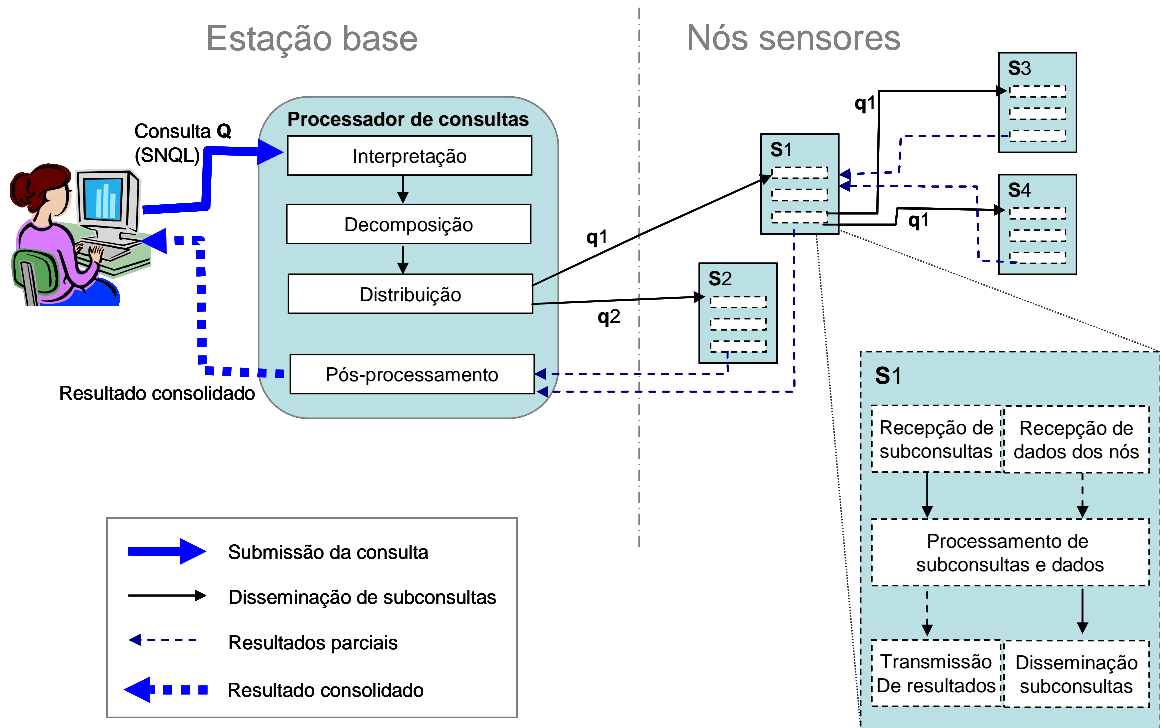


Figura 5.1 Estratégia do PDCRS para processamento de consultas em RSSFs.

A Figura 5.1 dá uma visão geral acerca da estratégia adotada pelo mecanismo PDCRS no processamento de uma consulta no cenário de RSSF: a partir de uma aplicação na estação base, uma consulta (Q), escrita na linguagem SNQL, é submetida ao processador de consultas. Esta consulta Q é então avaliada, interpretada e decomposta nas subconsultas q1 e q2, respectivamente relacionadas com as categorias de sensores identificadas na lista de relações da consulta original (cláusula *FROM*). Em seguida, na fase de distribuição, as subconsultas são disseminadas pela rede de sensores, sendo recebidas pelos nós sensores (S1 e S2) que têm a estação base como nó pai. Caso a subconsulta recebida se refira à categoria de sensor a qual o nó receptor faça parte, esta então começa a ser processada nesse nó. Caso contrário, o nó poderá retransmitir a subconsulta para seus nós descendentes ou ignorá-la, caso o protocolo de roteamento assim defina.

Em cada nó sensor, após a aceitação de uma subconsulta, as seguintes tarefas são executadas: coleta e processamento de dados conforme os critérios da consulta instalada, envio de pacotes de resultados para nós pais (conforme estratégias de roteamento e agregação adotadas) e recebimento de pacotes de

dados de nós sensores filhos, quando for o caso. A execução da subconsulta é finalizada no nó sensor quando o tempo de vida configurado é atingido. Este tempo refere-se ao parâmetro *Time Windows*, da consulta SNQL.

Durante o tempo de vida da consulta, a estação base recebe continuamente os resultados do processamento nos nós sensores, a qual realiza enquanto isso as atividades de pós-processamento desses resultados, a exemplo de agrupamentos, junções e consolidação das agregações, e a entrega dos resultados consolidados à aplicação usuária. Atingido o tempo de vida da consulta, este nó passa a ignorar os pacotes de dados provenientes dos nós sensores, envia pacote de interrupção da consulta corrente para a rede, e entra no estado de espera por submissão de novas consultas pela aplicação observadora.

Nas subseções a seguir serão detalhados os procedimentos operacionais do PDCRS, focando aspectos acerca do gerenciamento de esquemas das categorias de sensores e das consultas sobre RSSF.

5.2.1 Gerenciamento de esquemas

O PDCRS provê recursos para mapeamento dos esquemas de categorias de sensores como relações virtuais definidas dentro do contexto adequado para as consultas. A raiz do gerenciamento se encontra na definição do Esquema Global de relações (mapeamento relativo às categorias de sensores envolvidas), que é conceitualmente o catálogo inerente ao modelo de dados utilizado pela aplicação usuária, onde são mantidas as definições das relações envolvidas.

O gerenciamento das relações de categorias de sensores envolvidas deve se iniciar com a criação (definição e persistência) desses componentes, tal como se segue:

(i) Criação / definição do esquema global;
(ii) Criação / definição das relações (categorias de sensores);
(iii) Definição dos atributos das relações.

O esquema global considerado pode ser dinamicamente construído a partir do próprio PDCRS, e estará funcional desde que seja compatível com o esquema local

das relações advindas dos sensores que compõem as categorias de sensoriamento distribuídas pela rede. Depois de persistido no esquema global, esse modelo de dados deve ser inteiramente conhecido pelo processador de consultas na estação base, visto que apenas este nó central será capaz de inter-relacionar as informações vindas das diferentes categorias de sensores.

A seguir serão detalhados os procedimentos para criação e manutenção do modelo de dados no esquema global.

5.2.1.1 Criação do esquema global

O catálogo do esquema global deverá estar armazenado sob forma de um arquivo em formato XML na estação base da RSSF. Apesar desse arquivo poder ser alterado a partir de ferramentas externas ao PDCRS (a exemplo de editor de textos), a manutenção dessa estrutura pode ser feita pela aplicação gerenciadora que estará utilizando recursos do PDCRS para materializar essas definições, a partir da emissão de comandos de definição de dados apropriados.

Após a criação do esquema local, deverão ser disseminadas aos nós sensores as informações do esquema inerentes à categoria de sensores que cada nó faz parte. Na implementação corrente do PDCRS, uma cópia inteira do esquema global pode existir em cada nó sensor, mas na prática, cada nó sensor irá utilizar as definições vinculadas aos atributos de sua categoria.

Tabela 5.1 Requisitos de elementos para a categoria de sensores no esquema global.

Categoria de sensores	**Nome do atributo**	**Tipo do atributo**	**Descrição**
Temperatura	NodeId	Cadeia	Identificação do nó sensor.
	RegionId	Cadeia	Região do nó sensor.
	CollectedValue	Número real	Valor coletado.
	TimeStamp	Data e hora	Horário da coleta.
Umidade	NodeId	Cadeia	Identificação do nó sensor.
	RegionId	Cadeia	Região do nó sensor.
	CollectedValue	Número real	Valor coletado.
	TimeStamp	Data e hora	Horário da coleta.

Baseado nos requisitos mostrados na Tabela 5.1, serão apresentados, a seguir, os passos para se construir dinamicamente um esquema global de relações denominado “GlobalSchema”, utilizando declarações SNQL submetidas ao componente interpretador presente no PDCRS:

(i) Criar o esquema “GlobalSchema”. Este passo cria o arquivo ‘GlobalSchema.xml’, contendo uma estrutura vazia do catálogo:

CREATE SCHEMA GlobalSchema;

(ii) Abrir o esquema para utilização. A partir deste passo, todos os comandos DDL estarão intrinsecamente se referenciando ao esquema aberto:

USE GlobalSchema;

(iii) Criar esquema de relação mapeando uma categoria de sensores de temperatura (aqui denominado ‘TEMPERATURA’). Nesta relação estarão definidos os atributos ‘NodeId’ do tipo *text*, ‘RegionId’ do tipo *text*, ‘collectedValue’ do tipo *float*; e ‘timeStamp’ do tipo *time*:

CREATE SENSORS TEMPERATURA (NodeId *text*, RegionId *text*, collectedValue *float*, timeStamp *Time*);

(iv) Criar esquema da relação mapeando uma categoria de sensores de umidade (aqui denominado ‘UMIDADE’). Nesta relação estarão definidos os atributos ‘NodeId’ do tipo *text*, ‘RegionId’ do tipo *text*, ‘collectedValue’ do tipo *float*; e ‘timeStamp’ do tipo *time*:

CREATE SENSORS UMIDADE (NodeId *text*, RegionId *text*, collectedValue *float*, timeStamp *Time*);

Ao final da execução dos passos descritos anteriormente, o esquema global estará materializado no arquivo ‘GlobalSchema.xml’, com o conteúdo apresentado na Figura 5.2.

```
<SCHEMA>
 SCHEMATEST
 <RELATIONS>
  <SENSORS>
   TEMPERATURA
   <ATTRIB Type="TEXT">NODEID</ATTRIB>
   <ATTRIB Type="TEXT">REGIONID</ATTRIB>
   <ATTRIB Type="FLOAT">COLLECTEDVALUE</ATTRIB>
   <ATTRIB Type="TIME">TIMESTAMP</ATTRIB>
  </ SENSORS >
  < SENSORS >
   UMIDADE
   <ATTRIB Type="TEXT">NODEID</ATTRIB>
   <ATTRIB Type="TEXT">REGIONID</ATTRIB>
   <ATTRIB Type="FLOAT">COLLECTEDVALUE</ATTRIB>
   <ATTRIB Type="TIME">TIMESTAMP</ATTRIB>
  </ SENSORS >
 </RELATIONS>
</SCHEMA>
```

Figura 5.2 Esquema global materializado sob forma de um arquivo XML.

5.2.2 Processamento de consultas

A descrição a seguir define um exemplo de consulta a ser realizada no cenário de uma RSSF, que servirá como caso ilustrativo quando da descrição das fases do processamento executados pelo PDCRS. Este exemplo representa uma consulta sobre 2 (duas) categorias de sensores distintas (conforme Tabela 5.1). A consulta deve ser definida sobre as duas categorias distintas de sensores citadas anteriormente, onde cada sensor tem capacidade de realizar somente um tipo de sensoriamento. Isto posto, deve-se submeter uma consulta Q para obter o resultado conforme os critérios a seguir:

(i) Selecionar leituras de temperatura > 30° C quando a umidade < 50%;

(ii) Juntar as leituras de temperatura e umidade conforme as regiões semelhantes dos sensores;

(iii) Apresentar valores coletados por categoria de sensor, agregados na origem por quantidade de valores lidos, agrupados por código de região;

(iv) Programar os sensores para enviar os resultados a cada 5 minutos sobre leituras realizadas nos sensores a cada 1 minuto;

(v) Manter essa consulta gerando resultados continuamente durante 10 (dez) dias, reinicializando-a a cada 24 horas.

A consulta Q, expressa na linguagem SNQL, a ser injetada na rede pela aplicação a partir da Estação Base, está representada na Figura 5.3.

```
SELECT  Temp.RegionId, Temp.collectedValue as tempValue,
        COUNT(Temp.*), Hum.collectedValue as humValue, COUNT(Hum.*)
FROM     Temperatura as Temp, Umidade as Hum
WHERE    Temp.RegionId = Hum.RegionId AND
         Temp.collectedValue > 30  AND Hum.collectedValue < 50
GROUP BY Temp.RegionId, tempValue, humValue
TIME WINDOW  1 day
SENSE INTERVAL 1 minute
SEND INTERVAL  5 minute
SCHEDULE 10;
```

Figura 5.3 Consulta Q, escrita em SNQL.

Ao obter a consulta submetida pelo usuário, o processador de consultas do PDCRS será ativado para tratá-la, conforme as fases descritas nas subseções a seguir.

5.2.2.1 Interpretação

Esta fase, executada na estação base, avalia a consulta escrita em linguagem declarativa (SNQL), e a interpreta conforme as regras gramaticais desta linguagem. É também verificado se as relações e seus atributos, referenciados na consulta, estão definidos no esquema global de categorias de sensores da RSSF. Em seguida, a consulta é convertida em uma árvore de análise, e a partir desta são geradas as expressões da álgebra relacional, que dará origem ao plano de execução da consulta (*Query Execution Plan* – QEP).

O QEP aqui gerado é representado por um grafo de operadores, onde os nós representam operadores da álgebra relacional (como junção, seleção, projeção e agregações) ou as categorias de sensores envolvidos. As arestas representam o fluxo de dados percorrendo os nós deste grafo. A Figura 5.4 mostra o plano de execução da consulta exemplificada na Figura 5.3.

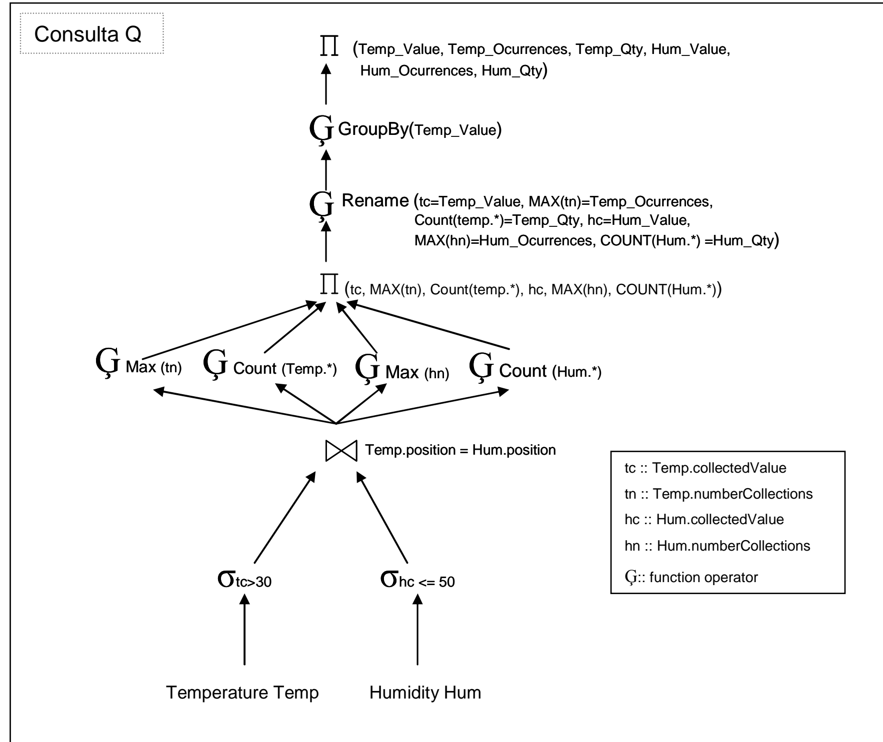


Figura 5.4 Plano de execução da consulta Q.

5.2.2.2 Decomposição

Após o Parsing, o QEP é decomposto em vários subgrafos (fase de decomposição). Cada subgrafo corresponde a um fragmento da consulta Q original. Os critérios para a decomposição são:

(i) categorias de sensores envolvidos na consulta e;

(ii) localização espacial dos sensores.

Assim, podem ser gerados fragmentos destinados a categorias de sensores específicas (temperatura ou umidade, por exemplo) ou fragmentos destinados a sensores localizados em regiões específicas. Assumimos que os sensores conhecem sua localização por possuírem algum sistema de localização próprios (por exemplo, alguns sensores da série MICA2 da Crossbow [19] possuem GPS) ou por existir um atributo que categorize as regiões cobertas por uma RSSF. Para implementar o segundo caso, os sensores devem prover junto aos seus dados de sensoriamento um atributo que o identifique como localizado em dada região.

Dentre os fragmentos gerados a partir da consulta original, um deles é relacionado com a estação base, e é chamado de consulta diretora Q0. Este terá a função de consolidar os resultados emitidos em resposta aos demais fragmentos (as subconsultas Q1 até Qn) após a sua disseminação aos nós distribuídos.

Técnicas de redução são aplicadas aos fragmentos antes de disseminá-los aos sensores. Definimos técnicas de redução para as operações de projeção, agregação, seleção e junção (da álgebra relacional), conforme as regras R1, R2 e R3, descritas a seguir:

(R1) Subconsultas (fragmentos) serão gerados de acordo com as categorias de sensores identificadas na consulta Q original, ou seja, um fragmento para cada categoria. A cláusula *From* de cada subconsulta somente se referenciará à categoria de sensor para o qual a subconsulta tiver sido gerada. A projeção de cada subconsulta terá a lista dos atributos necessários da sua categoria conforme referências encontradas a respeito desses atributos na consulta Q. Na cláusula *From* da consulta diretora Q0 será mantida a lista de projeção integral da consulta original, para ser possível consolidar os resultados correspondentes advindos das consultas distribuídas na fase de pós-processamento.

(R2) Para operações de seleção (Cláusula *Where*): se a condição de seleção se referir a atributos de somente uma determinada categoria de sensores, a operação deve fazer parte apenas da subconsulta correspondente àquela categoria. Assim, essa condição não fará parte do filtro da consulta diretora Q0.

(R3) No caso de existir uma operação de junção envolvendo atributos de duas categorias de sensores definidas na cláusula *From*, a condição relacionada com essa junção somente fará parte da cláusula de seleção da consulta diretora Q0, pois este tipo de operação somente será executado na estação base, quando da consolidação dos resultados no pós-processamento da consulta em questão.

Após a decomposição da consulta inicial em fragmentos e aplicadas as técnicas de redução, os fragmentos são disseminados na rede. A Figura 5.5 ilustra o grafo de operadores da consulta exemplo (Figura 5.3) após a fase de decomposição. A idéia é que com este plano, sejam disseminadas as subconsultas (ilustradas na Figura 5.6) para os nós sensores.

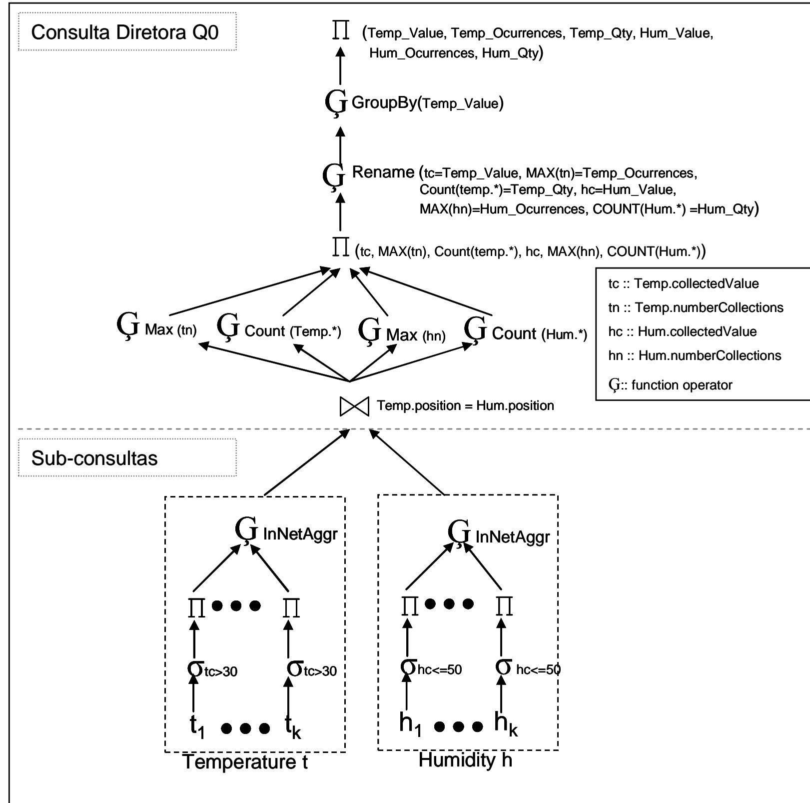


Figura 5.5 Plano de execução gerado na fase de decomposição.

```
/* Consulta Diretora */
Q0::{Manager}
SELECT TEMP.REGIONID, TEMP.COLLECTEDVALUE AS TEMPVALUE,
          SUM(TEMP.COUNT_TEMP), HUM.COLLECTEDVALUE AS HUMVALUE,
          SUM(HUM.COUNT_HUM)
FROM    [Q1] AS TEMP, [Q2] AS HUM
WHERE  TEMP.REGIONID = HUM.REGIONID AND TEMP.COLLECTEDVALUE > 30  AND
          HUM.COLLECTEDVALUE < 50
GROUP BY TEMP.REGIONID
Time Window 86400 SECOND Sense Interval 60 SECOND  Send Interval 300 SECOND
SCHEDULE 10 CYCLE;

/* Subconsultas */
Q1::{TEMPERATURA}
SELECT  REGIONID, COLLECTEDVALUE,  COUNT(*) AS COUNT_TEMP
FROM    TEMPERATURA
WHERE   COLLECTEDVALUE > 30
Time Window 86400 SECOND  Sense Interval 60 SECOND Send Interval 300 SECOND
SCHEDULE 10 CYCLE;

Q2::{UMIDADE}
SELECT   COLLECTEDVALUE, COUNT(*) AS COUNT_HUM
FROM     UMIDADE
WHERE   COLLECTEDVALUE < 50
Time Window 86400 SECOND  Sense Interval 60 SECOND  Send Interval 300 SECOND
SCHEDULE 10 CYCLE;
```

Figura 5.6 Fragmentação gerada pelo PDCRS.

Funções de agregação são naturalmente distribuídas nas subconsultas, para execução nos nós sensores distribuídos. Este comportamento do PDCRS permite o uso de estratégias que implementam algoritmos com técnicas de agregação em rede, a exemplo do demonstrado em [8][11].

A Tabela 5.2 sintetiza as técnicas de redução para projeção após a decomposição de uma consulta.

Tabela 5.2 Relocalização de itens de projeção, após a decomposição.

Tipo do item	Consulta diretora (Estação base)	Subconsultas (Nós sensores)
Atributos	Mantidas todas os itens da projeção original (Consulta Q), e alteradas suas referências para as relações derivadas resultantes das subconsultas.	Mantidos os itens de projeção de Q que fazem referência aos atributos de relações existentes naquela categoria de sensores. Incluídos como item de projeção da subconsulta os atributos de sua categoria que são referenciados nas cláusulas de seleção da consulta diretora Q0.
Constantes[9]	Mantidos todos os itens de constantes.	Removidos todas os itens de constantes.

A Tabela 5.3 sumariza as técnicas de redução para seleção após a decomposição de uma consulta.

Tabela 5.3 Relocalização de itens de seleção, após a decomposição.

Condição	Consulta diretora (Estação base)	Subconsultas (Sensores)
Referencia somente atributos de uma mesma categoria de sensores	Condição não fará parte do filtro de seleção da consulta diretora.	Condição fará parte do filtro das subconsultas do grupo sensor referenciado.
Referencia atributos de mais de uma categoria de sensores	Condição fará parte da consulta diretora. Denota caso de junção entre atributos de relações distintas (inner join).	Condição não fará parte do filtro das subconsultas.

A Figura 5.7 e Figura 5.8 mostram em formato XML, respectivamente, a consulta diretora Q0 que será executada na estação base, e as subconsultas

[9] Se uma constante (por exemplo, uma cadeia literal ou um número) for incluída na projeção da consulta Q, não há necessidade deste item fazer parte da projeção das subconsultas, considerando que não haverá processamento acerca deste. Assim, a constante apenas fará parte da lista de projeção a ser entregue como resultado ao usuário.

(fragmentos) Q1 e Q2 que serão posteriormente disseminados às categorias de sensores correspondentes.

Consulta diretora Q0

```
<QUERY TimeWindow="86400000" SenseInterval="60000" SendInterval="300000" QueryId="931911608">
 <ITEMSLIST Type="VARITEMS">
  <ITEM VId="0" ItemType="Attr" RELATION="TEMPERATURA">REGIONID</ITEM>
  <ITEM VId="1" ItemType="Attr" ColName="TEMPVALUE" RELATION="TEMPERATURA">COLLECTEDVALUE</ITEM
  <ITEM VId="2" ItemType="Attr" RELATION="TEMPERATURA" Function="COUNT">COUNT_TEMP</ITEM>
  <ITEM VId="3" ItemType="Attr" ColName="HUMVALUE" RELATION="UMIDADE">COLLECTEDVALUE</ITEM>
  <ITEM VId="4" ItemType="Attr" RELATION="UMIDADE" Function="COUNT">COUNT_HUM</ITEM>
  <ITEM VId="5" ItemType="Attr" RELATION="UMIDADE">REGIONID</ITEM>
  <ITEM VId="6" ItemType="Lit" DataType="INTEGER">30</ITEM>
  <ITEM VId="7" ItemType="Lit" DataType="INTEGER">50</ITEM>
  <ITEM VId="8" ItemType="Lit" DataType="INTEGER">1</ITEM>
 </ITEMSLIST>
 <RELATIONS>
  <RELATION Alias="TEMP" DerivedIndex="0">TEMPERATURA</RELATION>
  <RELATION Alias="HUM" DerivedIndex="1">UMIDADE</RELATION>
 </RELATIONS>
 <VARIDLIST Type="SELECT">0,1,2,3,4</VARIDLIST>
 <FILTER Type="WHERE">:0 %3D :5 AND :1 %3E :6 AND :3 %3C :7</FILTER>
 <SCHEDULE>10</SCHEDULE>
 <VARIDLIST Type="GROUPBY">0</VARIDLIST>
 <DISTRIBUTED>...</DISTRIBUTED>
</QUERY>
```

Figura 5.7 Consulta diretora Q0, em XML.

Subconsulta Q1

```
 <QUERY TimeWindow="86400000" SenseInterval="60000" SendInterval="300000" QueryId="931911608">
  <ITEMSLIST Type="VARITEMS">
   <ITEM VId="0" ItemType="Attr" RELATION="TEMPERATURA">REGIONID</ITEM>
   <ITEM VId="1" ItemType="Attr" RELATION="TEMPERATURA">COLLECTEDVALUE</ITEM>
   <ITEM VId="2" ItemType="Attr" ColName="COUNT_TEMP" RELATION="TEMPERATURA" Funct="COUNT"> *
</ITEMS>
   <ITEM VId="6" ItemType="Lit" DataType="INTEGER">30</ITEM>
  </ITEMSLIST>
  <RELATIONS><RELATION Alias="TEMP">TEMPERATURA</RELATION> </RELATIONS>
  <VARIDLIST Type="SELECT">0,1,2</VARIDLIST>
  <FILTER Type="WHERE">:1 %3E:6</FILTER>
  <SCHEDULE>10</SCHEDULE>
 </QUERY>
```

Subconsulta Q2

```
<QUERY TimeWindow="86400000" SenseInterval="60000" SendInterval="300000" QueryId="931911608">
 <ITEMSLIST Type="VARITEMS">
  <ITEM VId="3" ItemType="Attr" RELATION="UMIDADE">COLLECTEDVALUE</ITEM>
  <ITEM VId="4" ItemType="Attr" ColName="COUNT_HUM" RELATION="UMIDADE" Function="COUNT">*</ITEM>
  <ITEM VId="7" ItemType="Lit" DataType="INTEGER">50</ITEM>
 </ITEMSLIST>
 <RELATIONS><RELATION Alias="HUM">UMIDADE</RELATION> </RELATIONS>
 <VARIDLIST Type="SELECT">3,4</VARIDLIST>
 <FILTER Type="WHERE">:3 %3C:7</FILTER>
 <SCHEDULE>10</SCHEDULE>
</QUERY>
```

Figura 5.8 Subconsultas Q1 e Q2, em XML.

5.2.2.3 Disseminação das subconsultas

Após a consulta original Q ser decomposta em um número de fragmentos igual ao número de categorias de sensores envolvidas nesta consulta, estas partes deverão

ser distribuídas através da rede até as categorias de sensores de destino. Caso o filtro (cláusula de seleção) das subconsultas envolva atributo de localização (código da região), a distribuição será realizada somente para os sensores de categorias que atendam àquele predicado, ou seja, com localização compatível àquela selecionada. Cada fragmento é encapsulado em um pacote do qual o seu cabeçalho contém as informações necessárias sobre o critério de decomposição adotado (categoria de sensoriamento e localização), para que cada sensor possa avaliar o pacote recebido, e se for o caso, aceitá-lo e processar as subconsultas destinadas a si.

A disseminação das subconsultas será feita a partir da estação base, para os sensores configurados como sendo seus filhos imediatos, sendo os pacotes que carregarão os parâmetros daquele fragmento providos da identificação das categorias de sensores destinatárias, conforme a definição no esquema global. Cada nó filho da estação base, por sua vez, disseminará aquele pacote para os seus filhos, e assim respectivamente, até que todos os nós sensores presentes na rede tenham conhecimento sobre a nova consulta injetada. Esta operação utiliza uma estrutura implementada baseada em árvore de roteamento [63].

Cada nó sensor, após a aceitação do fragmento destinado a si, passa a ter o seu funcionamento determinado pelos parâmetros da subconsulta: o intervalo de sensoriamento conforme o parâmetro *Sense Interval*, o intervalo entre transmissões conforme o *Send Interval*, e a finalização do processamento conforme o tempo da consulta definido em *Time Window*, ou quantidade de linhas processadas, conforme *Data Window*.

Após a disseminação, um temporizador é criado na estação base para controlar o tempo de vida da consulta em processamento, baseado na sua cláusula *Time Window*, e esta estação entra no estado de espera por pacotes de resultados dos nós sensores.

5.2.2.4 Processamento das subconsultas

A execução desta fase é distribuída nos sensores existentes na RSSF. A aceitação ou não, pelos nós sensores, das subconsulta injetadas, dependerá de parâmetros tais como critério de distribuição aplicado e predicados de seleção existentes nas

subconsulta. Por exemplo: se a cláusula WHERE da subconsulta filtrar somente valores presentes em determinados conjunto de sensores, a exemplo do código de um identificador de região geográfica, o nó sensor deverá ter a inteligência necessária para somente “instalar” aquela consulta se for capaz de atender aquele predicado.

A partir da recepção e ativação da subconsulta, cada nó sensor é responsável por filtrar seus dados coletados conforme os parâmetros de projeção e seleção especificados, e enviá-los com destino ao nó central. Os critérios de agregação dependerão das abordagens de agregação utilizadas, que podem ser agregação tradicional ou agregação em rede. Não obstante o nosso processador de consulta utilizar, nas aplicações de prova de conceito, o protocolo ADAGA [8][11], o PDCRS mantém-se extensível, permitindo a adoção de outros mecanismos e abordagens.

A transmissão dos resultados para a estação base, provenientes das aquisições dos valores pelos sensores, será feita através da árvore de roteamento, onde cada sensor enviará as suas leituras para os nós definidos como seus pais, que por sua vez, enviarão esses pacotes para os nós acima na árvore, até alcançar a estação base. Nesses procedimentos poderão ser executadas as operações de agregação em rede definidas a partir dos parâmetros da consulta injetada na rede.

Quando o valor especificado na cláusula *Time Window* for alcançado, o nó sensor finalizará o processamento sobre aquela requisição, e entrará em modo de espera de recepção de nova subconsulta.

5.2.2.5 Pós-processamento para consolidação dos resultados

Esta etapa é executada com a finalidade de preparar os dados para serem apresentados ao usuário e faz uso dos parâmetros presentes na consulta Q0 (consulta diretora), que utiliza como fontes os resultados processados nos sensores das diversas categorias, advindos das subconsultas (Q1..Qn). O processamento final consiste em executar:

(i) Operações de projeção, para reunir os atributos, especificados na consulta submetida pelo usuário, provenientes de diferentes categorias de sensores;

(ii) Operações de junção[10] e/ou união, para relacionar os dados enviados por nós de diferentes categorias de sensores.

(iii) Operações de agregação, para reunir os dados advindos de diferentes nós sensores; e

(iv) Outras operações, não processadas nos nós sensores por apenas terem sentido no contexto da estação base. Um exemplo clássico é a aplicação da cláusula *HAVING*, que especifica predicados aplicáveis ao resultado final obtido após a agregação de todos os resultados parciais gerados para a consulta. Neste caso, apenas a estação base é capaz de processar a operação definida na cláusula *HAVING*.

5.3 Considerações finais

Neste capítulo foi apresentada um processador de distribuição de consultas para redes de sensores sem fio. A solução foi focada nos aspectos relacionados com o desenvolvimento de um mecanismo gerenciador de consultas emitidas em linguagem declarativa de alto nível, no seu planejamento, otimização e procedimentos de fragmentação e distribuição para o conjunto de sensores da rede, e como os dados a serem coletados deverão ser selecionados na origem da sua coleta.

[10] Abordagem detalhada pode ser vista em [8][11].

Capítulo 6
SIMULAÇÃO

6.1 Introdução

A prova de conceito das funcionalidades propostas pelo PDCRS foi realizada num cenário simulado de monitoramento de RSSF virtual, onde foram realizados testes com o mecanismo proposto, conforme detalhado nas subseções a seguir.

Para a concretização das experimentações com o PDCRS, foi construído um conjunto de aplicações, compondo um simulador de rede de sensores sem fio. Nesta implementação (conforme apresentado em [47]) foi possível criar o cenário definido para experimentação deste mecanismo.

O simulador foi desenvolvido em Object Pascal, com versão inicial voltada para ambiente Microsoft Windows, e compõe-se de 4 (quatro) módulos de aplicação, cada um com o seu papel bastante específico dentro desta arquitetura, conforme sintetizado a seguir:

(i) Módulo de interface com usuário, na estação base: gerencia a interface para submissão de consultas SNQL (*Base Station Manager*);

(ii) Módulo de processamento de consultas da estação base: processa consultas e pós-processamento e é a ponte entre o gerenciamento e nós sensores (*Base Station Processor*);

(iii) Nós sensores: módulo que simula os nós sensores distribuídos pela rede de sensores sem fio (*Sensor Processors*);

(iv) Agente de rede: simula o meio físico de comunicação, fornecendo suporte para a interação entre os demais módulos (*Network Agent*).

O processamento de consultas da estação base gerencia o status da rede a partir das informações passadas pelos nós sensores. Estes apenas precisam conhecer que outro(s) nó(s) pode(m) ser seu(s) pai(s), e fornecer informação de controle sobre o seu estado de ativação. Esta estrutura é dinamicamente atualizada quando da ativação atemporal de novos nós sensores, ou a desativação de algum

dos nós sensores existentes. As informações relacionadas à aplicação, tais como requisição das consultas e controle dos resultados gerados, deverão ser injetadas pelo módulo de gerenciamento de interface com usuário e estar sob o controle do módulo de processamento de consultas, ambos na estação base.

Outros detalhes acerca da simulador de rede de sensores sem fio aqui citado encontram-se no Apêndice B desta dissertação e em [47].

6.2 Cenário utilizado

A RSSF considerada nos testes compõe-se de uma arquitetura constituída por um nó central robusto (estação base) quanto à capacidade de memória, disco e processamento, e um conjunto de sensores especializados, espalhados aleatoriamente na área monitorada, dentro de um raio definido com centro na estação base. Cada nó possui um identificador de região (Ri[i=0..n]), sendo i o fator proporcional à distância em que o nó se encontra da estação base. Conforme as especificações nos parâmetros das consultas que podem ser injetadas, pode ser necessária a realização de estratégias de agregação e roteamento nos nós sensores. Nesta prova de conceito foi implementado o algoritmo ADAGA [8][11] juntamente com estratégias de roteamento em árvore [63].

O objetivo foi integrar informações como temperatura e umidade, relacionando-as através de seus atributos de posicionamento. No cenário apresentado aqui, os sensores admitidos são capazes de captar apenas um tipo de informação do ambiente, aqui exemplificado como sensores de temperatura ou de umidade. Todos os sensores que captam o mesmo tipo de informação são ditos como pertencentes a uma mesma categoria de sensores. Embora haja a definição de sensores de categorias de sensoriamento distintas, admite-se que há características idênticas entre eles, como, por exemplo, capacidade de processamento, quantidade de memória e energia disponível e potência de transmissão (que está diretamente relacionada com alcance dos pacotes transmitidos).

Não obstante vários modelos de topologia poderem ser utilizados a partir do PDCRS, na corrente simulação a implementação da topologia tolerou uma estrutura "*scale-free", onde* se admite que um nó sensor pode ter vários nós filhos e mais de

um nó pai [53]. Isso simula situações reais, onde o relacionamento entre os nós pode estar diretamente relacionada à distância entre eles, de forma que, quanto mais próximos estivessem dois sensores, maiores seriam as chances deles trocarem mensagens, independentemente de pertencerem ou não a uma mesma categoria de sensoriamento. Isto também permite implementar técnicas para racionalizar o uso consumo de energia nas operações de recepção e envio de dados.

Este ambiente provê uma estação base logicamente rodeada por um conjunto de nós sensores especializados distribuídos em diversas regiões de localização em relação ao núcleo da rede (quanto maior o número da região, mais distante o nó se encontra do núcleo), dentro da área virtual monitorada (ver Figura 6.1, abaixo). As funcionalidades dos demais nós são função de sua localização dentro da rede, tais como:

- Nós intermediários: são localizados entre a primeira região da rede, próxima ao núcleo (estação base), até antes da última região (borda da rede). Podem possuir função de sensoriamento (coleta de dados) e também têm a função de disseminar as subconsultas para os seus nós filhos, até os nós folhas;
- Nós folhas: são localizados na borda da rede (regiões perimetrais). Tem função de sensoriamento para envio de dados em direção à estação base. Estes filhos utilizam seus nós pais para rotear os pacotes processados em direção à estação base.

Conforme as definições intrínsecas do PDCRS, os nós sensores podem se comunicar entre si e com a estação base, dependendo da relação de roteamento entre eles.

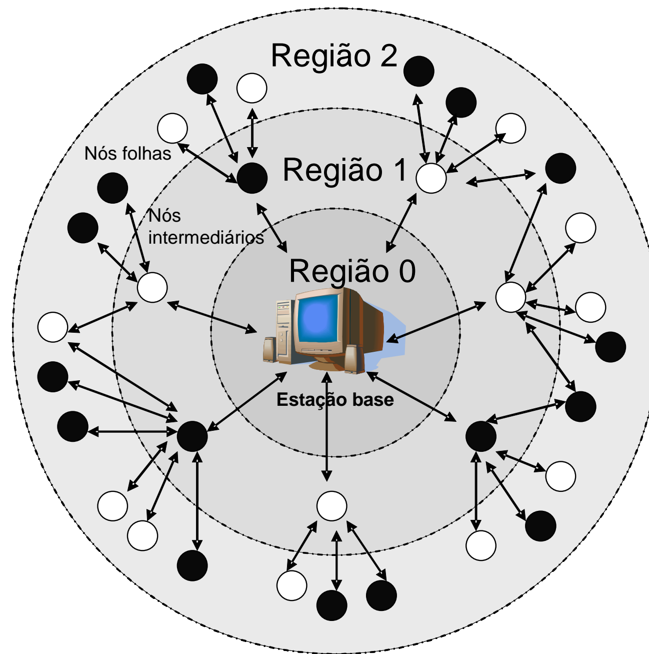


Figura 6.1 Diagrama simplificada da RSSF.

6.3 Testes realizados

Os testes foram realizados utilizando-se seis máquinas (microcomputadores Pentium 4) conectadas em rede física, sendo as aplicações assim distribuídas: (i) uma máquina representou a estação base, hospedando a aplicação de interface com usuário (*Base Station Manager*) e a aplicação de processamento e consolidação de consultas (*Base Station Processor*); (ii) quatro máquinas executaram instâncias da aplicação do módulo de nós sensores (*Sensor Node Processors*), representando as categorias de sensores distribuídos, e; (iii) uma máquina hospedou exclusivamente a aplicação que simula a “nuvem” de comunicação da rede de sensores (Network Agent), através da qual todas as outras aplicações se comunicam.

O papel exercido pelas estações que hospedam a aplicação *Sensor Node Processors* pode ser distribuído em várias máquinas para execução simultânea dentro da rede real, o que possibilita que cada categoria distinta de sensores possa ser simulada em máquinas diferentes, quando houver necessidade de realizar testes de escalonamento com uma grande quantidade de nós sensores.

Foi simulada a execução da consulta Q, mostrada na Figura 5.3. O objetivo foi submeter esta consulta ao mecanismo proposto para avaliar as vantagens de usar o PDCRS, considerando as seguintes métricas: tempo de resposta da rede, tempo de vida da rede, tempo de vida de um nó sensor, nós ativos por tempo e região, nós inativos por tempo e região, taxa de entrega de pacotes e taxa de descarte de pacotes. Essas métricas e os resultados obtidos nos testes são detalhadas na subseção a seguir.

6.4 Resultados

O teste de avaliação do tempo de resposta da rede pretendeu mostrar a média dos tempos decorridos entre as submissões da consultas até o recebimento, pela estação base, dos primeiros pacotes de resposta de cada nó sensor, comparando execuções com diferentes números de nós sensores.

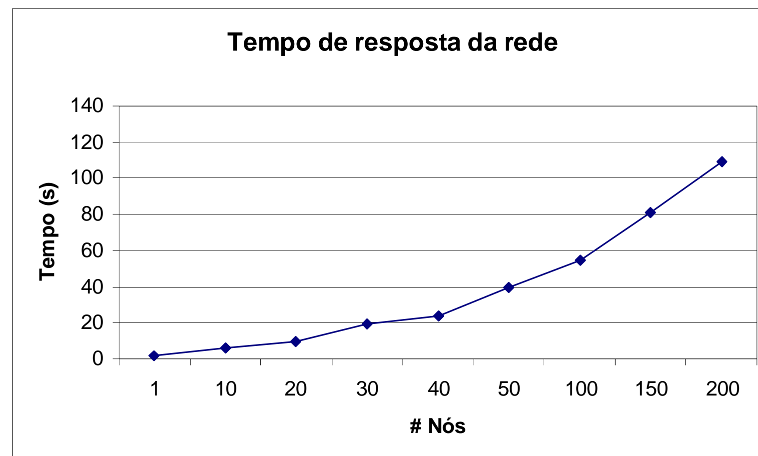


Figura 6.2 Tempo de resposta da rede.

A Figura 6.2 apresenta os tempos de resposta para a consulta Q considerando o número de nós sensors ativos aptos a processar as subconsultas derivadas. O tempo foi medido do momento em que a consulta foi injetada até a recepção pela estação base dos pacotes de resultados gerados pelos nós sensors. Este tempo de execução, que é função do intervalo especificado na cláusula *Send Interval*, pode ser influenciado pelo tempo de execução das fases de interpretação, decomposição, e execução dos fragmentos pelos nós sensores, e pelo tempo de

transporte do pacote através da rede. Além disso, no ambiente simulado, este mesmo tempo também pode ser influenciado pelo desempenho da estação real onde o módulo de nós sensores for executado.

Nos testes realizados com o simulador de RSSF, a influência do desempenho da estação real nos resultados foi mais percebida quando o número de sensores simulados estava acima de 50 (cinqüenta) nós. Este fato se deve à característica de implementação do Simulador de RSSF[11]. Para que os resultados simulados sofressem menos influência do desempenho da máquina, nos testes de tempo de resposta da rede foram utilizadas 4 (quatro) estações reais executando o módulo de nós sensores, com cada uma delas configuradas para simular até 50 nós.

Ao analisarmos o gráfico praticamente linear na Figura 6.2, concluímos que a escalabilidade do PDCRS é proporcional à quantidade de sensores ativos na rede com potencial para responder à consulta injetada.

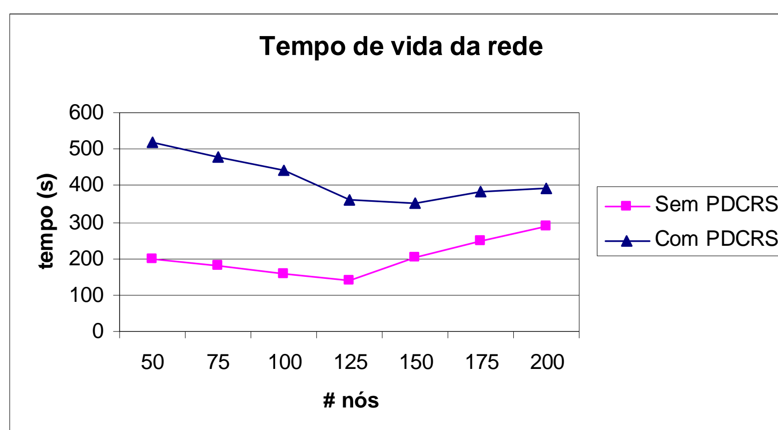


Figura 6.3 Tempo de vida da rede.

A análise do tempo de vida da RSSF relacionou execuções com diferentes números de sensores, e comparou a execução de consulta com e sem processamento das estratégias do PDCRS. A execução com PDCRS significa ativar a filtragem de dados nos sensores e ativação das estratégias de agregação em rede, conforme parâmetros da subconsultas geradas na etapa de decomposição da consulta original. A Figura 6.3 mostra como a rede se comporta a respeito de seu

tempo de vida, que é calculado a partir do tempo de vida de todos os sensores da RSSF. Por tempo de vida da rede traduzimos em quanto tempo a rede está habilitada a fornecer os dados coletados à estação base relativamente a uma dada consulta. Foi simulada a execução da consulta Q (Figura 5.3) utilizando o PDCRS e sem utilizar o PDCRS. No último caso, todos os sensores transmitiram integralmente todos os dados coletados para a estação base. A Figura 6.3 apresenta um importante resultado, pois demonstra que o uso deste processador de consultas estendeu o tempo de vida da RSSF. Conseqüentemente, estes resultados comprovam que o uso da tecnologia de banco de dados em redes de sensores pode melhorar significativamente o tempo de vida útil dessas redes, além de comprovar a eficiência de PDCRS.

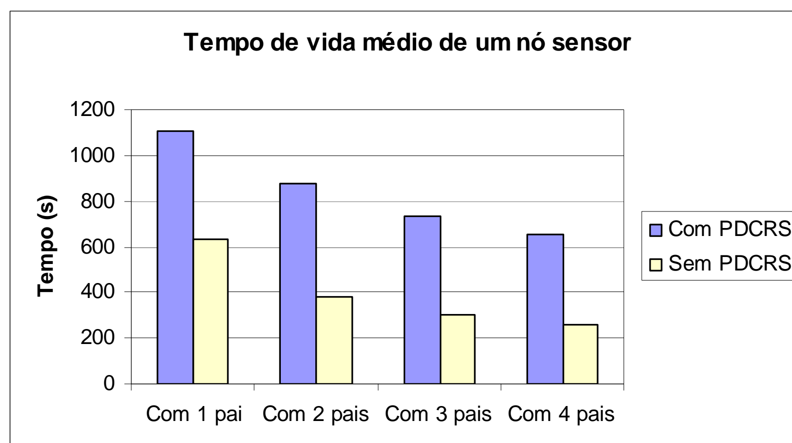


Figura 6.4 Tempo de vida dos nós sensores.

A avaliação do tempo de vida médio de um nó sensor mostrou (Figura 6.4), dentro do tempo de simulação, a relação entre os tempos de vida de nós sensores executando separadamente diferentes tipos de subconsultas, variando o número de nós pais, e comparando a aplicação das estratégias do PDCRS com outra estratégia de distribuição genérica. A figura referenciada acima mostra o tempo médio de vida de um nó sensor considerando a quantidade de nós pais configurada, isto é, o número de sensores aos quais cada nó sensor deve transmitir

---

[11] Detalhes da implementação do módulo de sensores do Simulador de RSSF estão na sessão B.7.4 do Apêndice B desta dissertação.

os seus pacotes de dados com resultados. Esta métrica influencia diretamente no tempo de vida de toda a RSSF: a possibilidade de um nó sensor ter mais de um nó pai influencia positivamente a tolerância a falhas na rede, mas aumenta o consumo de energia em cada nó, pelo maior uso de comunicação, ao replicar os pacotes para os pais definidos.

Considerando que a distribuição dos nós sensores na RSSF simulada se deu nas camadas de regiões, mais próximas ou menos próximas da estação base, buscamos avaliar a diferenciação dos tempos de vida dos nós relacionados com as regiões onde estes se encontravam. Assim, foram avaliadas a taxas percentuais de nós sensores que se mantinham ativos e nós sensores que tinham a sua energia esgotada, no decorrer do tempo de operação da RSSF. Para este teste, foram ativados inicialmente 300 nós sensores, espalhados aleatoriamente pelo cenário, cobrindo uniformemente todas as regiões dentro do raio definido da RSSF.

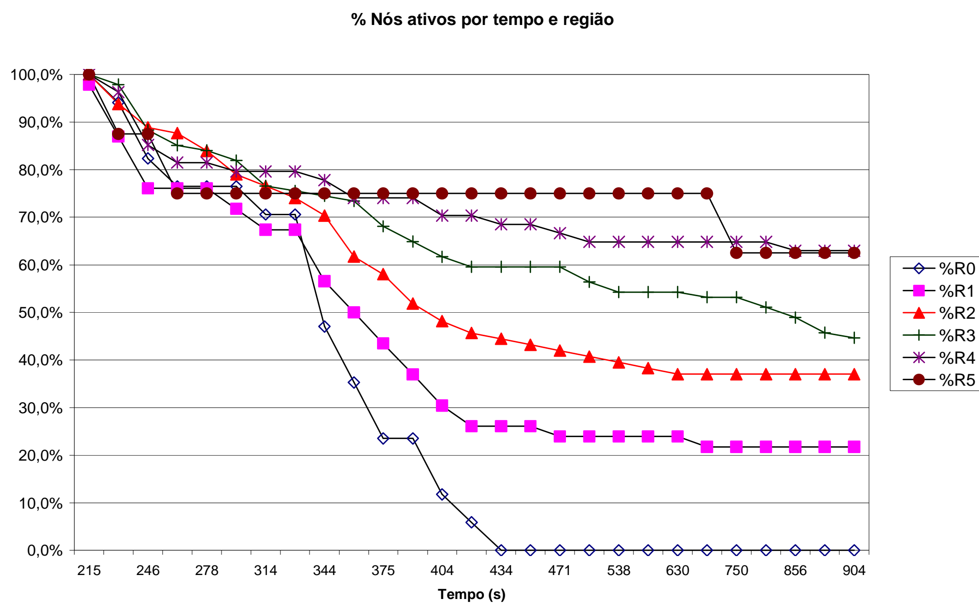


Figura 6.5 Nós ativos por tempo e região.

A Figura 6.5 mostra, a partir do número total de nós de cada região, o percentual dos nós que continuavam ativos, e a Figura 6.6, apresenta a contrapartida, relacionando os percentuais dos nós que iam ficando com energia

esgotada (nós “mortos”). Foi constatado pela simulação que os nós em regiões mais próximas da estação base sempre serão mais utilizados pela estratégia de distribuição da consulta e recepção dos resultados, considerando que, além de processarem a coleta de seus próprios dados, também realizam o redirecionamento dos dados de/para os seus nós filhos. Naturalmente Isto denota a necessidade de projetos de topologias das redes preverem nós mais robustos quando próximos da estação base, principalmente em relação ao aspecto de energia.

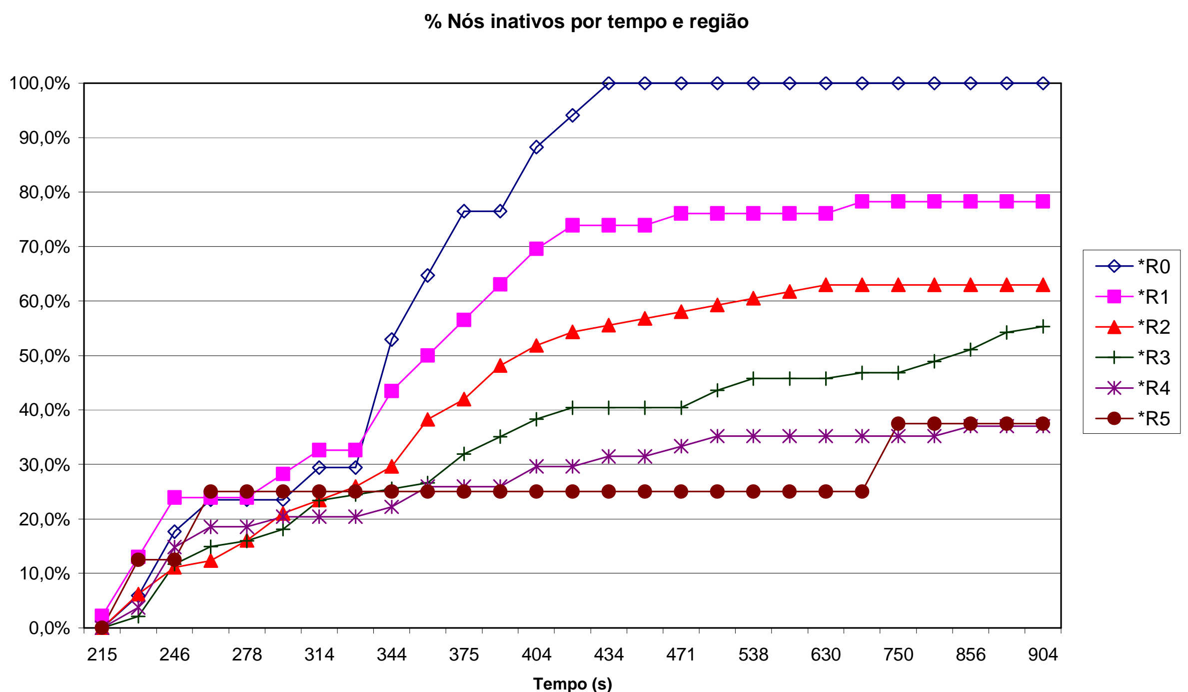


Figura 6.6 Nós inativos por tempo e região.

Finalmente, foi avaliada a relação entre o número de pacotes transmitidos pelos nós de origem e a sua recepção com sucesso pelos nós destinatários, considerando o uso ou não do PDCRS. Isto pretendeu simular situações dentro de uma rede com características de comunicação *broadcast*, quando um determinado pacote pode ou não atingir o seu destino. A taxa de descarte é a diferença entre a quantidade de pacotes transmitidos pelos nós de origem e a quantidade de pacotes recebidos pelos nós destinatários. Um pacote transmitido será descartado caso o agente de rede não dê vazão a ele no tempo necessário para a sua transmissão e

recepção, conforme a banda de comunicação definida e a distâncias entre os nós envolvidos. A Figura 6.7 demonstra que a estratégia do PDCRS diminui a quantidade de pacotes descartados, haja vista a diminuição do tráfego na rede, resultado da racionalização das transmissões ocorrida com a utilização do mecanismo aqui proposto.

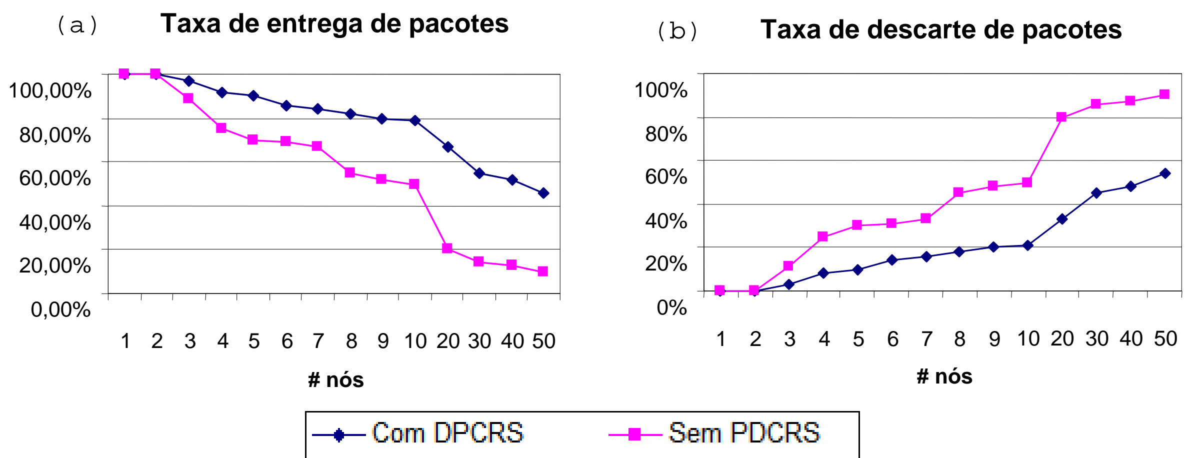


Figura 6.7 Taxa de entrega de pacotes e Taxa de descarte de pacotes

.

6.5 Considerações finais

Neste capítulo foram apresentados os testes realizados em ambiente simulado acerca do processador de consultas distribuído para redes de sensores sem fio, objeto desta dissertação. Os resultados das tarefas de experimentação realizadas, utilizando um ambiente de simulação, demonstraram a eficiência do mecanismo proposto.

Capítulo 7

CONCLUSÕES E TRABALHOS FUTUROS

Nesta dissertação foram apresentados aspectos relacionados a redes de sensores sem fio (RSSF) e processamento de consultas contínuas, sendo estes conceitos referenciados através de exemplos de projetos e produtos desenvolvidos tendo como foco estas tecnologias. Estes assuntos, abordados de forma introdutória, serviram como fulcro para a proposta do mecanismo objeto deste trabalho.

Como núcleo deste trabalho, foi proposto um mecanismo para processamento e distribuição de consultas em RSSFs, denominado PDCRS, caracterizado por gerenciar a distribuição e a execução de consultas a partir de uma estação base para conjuntos de nós sensores de categorias diversas disseminados numa RSSF. A arquitetura do mecanismo proposto foi construída com a premissa de racionalizar a forma como este tipo de rede trata a disseminação de consultas e o tratamento de seus resultados, visando reduzir consumo de energia e de memória dos componentes de uma RSSF durante o processamento de consultas.

O mecanismo aqui proposto objetiva facilitar a implementação de processadores de consultas distribuídas num cenário de RSSF, além de definir como os dados objetos das tarefas de sensoriamento deverão ser selecionados na sua origem (nós sensores), e o seu gerenciamento a partir de uma estação base. A sua estrutura permite que sejam implementadas extensões relacionadas com agregações em rede, agrupamentos e junções. Implementação desta natureza foi realizada no simulador de RSSF, ferramenta de apoio que foi desenvolvida para as provas deste trabalho. Testes preliminares utilizando simulação comprovaram a usabilidade e a eficiência do mecanismo PDCRS.

Também foi apresentada neste trabalho a proposta de uma linguagem para consultas declarativas, denominada SNQL, voltada ao desenvolvimento de aplicações em redes de sensores sem fio, à qual foram incluídas, entre outras, cláusulas que determinam o tempo de execução da consulta, intervalos entre sensoriamentos e intervalos entre transmissões de pacotes de resultados. Essa linguagem foi implementada no sistema simulador de RSSF, tendo sido a

ferramenta utilizada como prova de conceito do mecanismo núcleo desta dissertação.

O simulador de redes de sensores sem fio para distribuição de consultas contínuas, denominado SNETSIM, que foi implementado no contexto deste trabalho, considerou otimizações relacionadas a agregações em rede, decisões sobre o descarte de pacotes, técnicas de roteamento e estratégias de conservação de energia e memória a partir da manipulação dinâmica dos parâmetros de consultas em SNQL.

Como trabalhos futuros planeja-se continuar com as seguintes atividades:

(i) Definir e Implementar medidas de melhor gerenciamento acerca da escalabilidade da rede, tal como controles dinâmicos para definição dos relacionamento entre os nós, baseados na distribuição geográfica em relação ao núcleo da rede;

(ii) Definir e Implementar controles sobre potência do sinal de transmissão em função da energia presente nos nós, permitindo assim o desenvolvimento de cenários de comunicação entre os nós mais próximos do mundo real, para testar o comportamento do PDCRS sobre esses cenários;

(iii) Definir mecanismos padronizados para pós-processamento da consulta na estação base, facilitando a customização de estratégias de junção e agrupamento no núcleo da rede;

(iv) Definir e implementar mecanismo de junção distribuída nos nós sensores;

(v) Adaptar e implementar os componentes do PDCRS em nós sensores reais, a partir da chegada de sensores das séries MICA2 e MICADOTS (da Crossbow [19]) em nosso grupo de pesquisa (projeto financiado pela Fundação Cearense de Apoio a Pesquisa). Desta forma, será possível avaliar o comportamento do PDCRS ao ser executado em uma RSSF real. Mais ainda, será possível observar o comportamento do PDCRS ao ter parte de sua execução realizada em ambientes com severas restrições reais de memória e bateria, como os sensores.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Akyildiz, I., Su, W., Sankarasubramaniam, Y., Cayirci, E. *A survey on sensor networks*. IEEE Communications, pg. 102-114, Agosto de 2002.

[2] Al-Karaki, J., Kamal, A. *A taxonomy of routing techniques in wireless sensor networks*. Handbook of sensor networks: compact wireless and wired sensing systems. Edited by Mohammad Ilyas and Imad Mahgoub, CRC Press, pg. 117-140, Julho de 2004.

[3] Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., Widom, J. *STREAM: The Stanford Data Stream Management System.* Department of Computer Science, Stanford University, Março de 2004.

[4] Arasu, A., Babu S., Widom, J. *The CQL Continuous Query Language: semantic foundations and query execution*. Technical report, Stanford University, Outubro de 2003.

[5] Babu, S., Motwani, R., Babcock, B., Datar, M. *Models and issues in data stream systems*. ACM SIGMOD/PODS, Junho de 2002.

[6] Babu, S., Widom, J. *Continuous Queries over Data Streams*. ACM SIGMOD Record, Vol. 30, Issue 3, Setembro de 2001.

[7] Bonnet, P., Gehrke, J., Seshadri, P. *Querying the physical world*. IEEE Personal Communications, Vol. 7, N° 5, pg. 10-15, Outubro de 2000.

[8] Brayner, A., Lopes, A., Meira, D., Vasconcelos, R., Menezes, R. *ADAGA: ADaptive AGgregation Algorithm for Sensor Networks*. XXI Simpósio Brasileiro de Banco de Dados – SBBD, pg. 191-205, Outubro de 2006.

[9] Brayner, A., Lopes, A., Meira, D.,Vasconcelos, R., Menezes, R. *An Adaptive In-network Aggregation Operator for Query Processing in Wireless Sensor Networks*. Journal of Systems and Software, Holanda, v. 21, p. 1-20, 2007.

[10] Brayner, A., Lopes, A., Menezes, R., Vasconcelos, R. *Balancing energy consumption and memory usage in sensor data processing*. ACM Symposium on Applied Computing – SAC, Março de 2007.

[11] Brayner, A., Lopes, A., Meira, D., Menezes, R., Vasconcelos, R., Menezes, R. *Toward adaptive query processing in wireless sensor networks*. Signal Processing, Volume 87, Issue 12, pg. 2911-2933, Maio de 2007.

[12] Center for Embedded Networked Sensing, *Annual Progress Report*. Disponível em http://research.cens.ucla.edu/pls/portal/docs/PAGE/CENS/CENS_ABOUT_US/ANNUALREPORTPUBLIC2003.PDF, Acessado em Julho de 2007.

[13] Chandrasekaran, S., Franklin, M. *Streaming Queries over Streaming Data*. 28th VLDB Conference, Janeiro de 2002.

[14] Chaudhuri, S. *An Overview of Query Optimization in Relational Systems*. Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, p.34-43, Junho de 1998.

[15] Chen, J., DeWitt, D. J., Tian, F., Wang, Y. *NiagaraCQ: a scalable continuous query system for internet databases*. ACM SIGMOD International Conference on Management of Data - COMAD, pg. 379-390, Maio de 2000.

[16] Chen, W., Hou, J. *Data Gathering and Fusion in Sensor Network*. Handbook of Sensor Networks – Algorithms and Architectures. Published by John Wiley & Sons, Inc., Hoboken, New Jersey, Maio de 2005.

[17] Code Gear. *Code Gear Home Page.* Disponível em http://www.codegear.com/main.aspx, acessado em Agosto de 2007.

[18] Cranor, C. D., Johnson, T., Spatscheck, O., Shkapenyuk, V. *Gigascope: a stream database for network applications*. ACM SIGMOD Conference, pg. 647-651, Junho de 2003.

[19] Crossbow Internet page, *Wireless Sensor Networks*. Disponível em http://www.xbow.com/Home/HomePage.aspx, Acessado em Julho de 2007.

[20] Free Pascal. *Open Source Compiler For Pascal And Object Pascal*. Disponível em http://www.freepascal.org/ , acessado em Agosto de 2007.

[21] Garcia-Molina, H., Ullman, J., Widom, J. *Database System Implementation*. Prentice-Hall, Inc., Janeiro de 2000.

[22] Garcia-Molina, H., Ullman, J., Widom, J. *Database Systems: The Complete Book*. Prentice Hall, Outubro de 2001.

[23] Golab, L. *Querying sliding windows over on-line data streams*. International Conference on Data Engineering ICDE/EDBT Ph.D. Workshop, pg. 1-10, Março de 2004.

[24] Golab, L., Özsu, M. *Issues in Data Stream Management*. ACM SIGMOD Record, Vol. 32, No. 2, Junho de 2003.

[25] Govindan, R., Hellerstein, J., Hong, W., Madden, S., Franklin, M., Shenker, S. *The sensor network as a database*. Technical Report, University of Southern California, Setembro de 2002.

[26] Graefe, G. *Query Evaluation Techniques for Large Databases*. ACM Computing Surveys, Vol. 25, pg. 73-170, Junho de 1993.

[27] Gray, J., Bosworth, A., Layman, A., Pirahesh, H. *Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-total*. International Conference on Data Engineering - ICDE, pg. 152-159, Fevereiro de 1996.

[28] Heinzelman, W., Chandrakasan, R., Balakrishnan, A. *Energy-efficient communication in wireless sensor networks.* Hawaii International Conference on Systems Sciences - HICSS, pg. 1-10, Janeiro de 2000.

[29] Heinzelman, W., Kulik , J., Balakrishnan, H. *Adaptive protocols for information dissemination in wireless sensor networks*. ACM/IEEE International Conference in Mobile Computing and Network – MobiCom, pg. 174–185, Agosto de 1999.

[30] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K. *System architecture directions for networked sensors.* International Conference on Architectural Support Programming Languages Operating Systems - ASPLOS, pg. 93–104, Novembro de 2000.

[31] Intanagonwiwat, C., Govindan, R., Estrin D. *Directed difusion: a scalable and robust communication paradigm for sensor networks*. ACM International Conference in Mobile Computing and Network - MobiCom, pg. 56-67, Agosto de 2000.

[32] Jalote, P. *Fault tolerance in distributed systems*. Prentice Hall PTR; US Edition, Abril de 1994.

[33] Koudas, N., Srivastava, D. *Data stream query processing: a tutorial*. Proceedings of the 29th VLDB Conference, Outubro de 2003.

[34] Krishnamachari, B., Estrin, D., Wicker , S. *Modelling Data-Centric Routing in Wireless Sensor Networks*. Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (InfoCom'02), Junho de 2002.

[35] Lahiri, K., Raghunathan, A., Dey, S., Panigrahi, D. *Battery-driven system design: A new frontier in low power design*. Proceedings of International Conference on VLSI Design, Janeiro de 2002.

[36] Lazarus Project. *About the Lazarus Project*. Disponível em http://www.lazarus.freepascal.org/, acessado em Agosto de 2007.

[37] Li, L., Halpern, J. *Minimum energy mobile wireless networks revisited*. IEEE International Conference on Communications, Junho de 2001.

[38] Li, X., Kim, Y., Govindan, R., Hong, W. *Multi-dimensional range queries in sensor networks*. In Proceedings of the 1st ACM Conference on Embedded Networked Sensor Systems (Sensys '03), Novembro de 2003.

[39] Madden, S. *Query Processing for Streaming Sensor Data*. PhD Qualifying Exam Proposal, UC Berkeley, Maio de 2002.

[40] Madden, S., Franklin, M., Hellerstein, J., Hong, W. *The design of an acquisitional query processor for sensor networks*. In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pg. 491–502, Junho de 2003.

[41] Madden, S., Franklin, M. J., Hellerstein, J., Hong, W. *TAG: a Tiny AGgregation service for ad-hoc sensor networks*. Symposium on Operating System Design and Implementation – OSDI, pg. 171–182, Dezembro de 2002.

[42] Madden, S., Franklin, M., Hellerstein, J. *TinyDB: an acquisitional query processing system for sensor networks*. ACM Transactions on Database Systems - TODS, Vol. 30, N$^{o.}$ 1, pg. 122-173, Março de 2005.

[43] Madden, S., Franklin, M., Hellerstein, J., Hong, W. *The design of an acquisitional query processor for sensor networks*. ACM SIGMOD International Conference on Management of Data - COMAD, pg. 491-502, Junho de 2003.

[44] Mainwaring, A., Polastre, J., Szewczyk, R., Culler, D., Anderson, J. *Wireless Sensor Networks for Habitat Monitoring*. WSNA'02, Setembro de 2002.

[45] Manjeshwar, A., Agrawal, D. *APTEEN: A Hybrid Protocol for Efficient Routing in Wireless Sensor Networks*. Proceedings 2nd. Workshop on Parallel and Distributed Computing, Issues in Wireless Networks and Mobili Computing, Abril de 2002.

[46] Manjeshwar, A., Agrawal, D. *TEEN: A Routing Protocol for Enhanced Efficiency in Wireless Sensor Networks*. Parallel and Distributed Processing Symposium. Proceedings 15th International Volume, Abril de 2001.

[47] Meira, D., Brayner, A., Lopes, A., Menezes, R. *Simulador de Rede de Sensores para Distribuição de Consultas Contínuas*. XXII Simpósio Brasileiro de Banco de Dados – SBBD, IV Sessão de Demos, pg. 3-8, Outubro de 2007.

[48] Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., Varma, R. *Query Processing, Resource Management, and Approximation in a Data Stream Management System*. 2003 CIDR Conference, Janeiro de 2003.

[49] Perrig, A., Tygar, J. *Secure Broadcast Communication in Wired and Wireless Networks*. Kluwer Academic Publishers, Outubro de 2002.

[50] Raghunathan, V., Schurgers, C., Park, S., and Srivasta, M. *Energy aware wireless microsensor networks*. IEEE Signal Processing Magazine, Vol.19, nº2 , pg. 40-50, Março de 2002.

[51] Rodoplu, V., Meng, T. *Minimum Energy Mobile Wireless Networks*. IEEE Journal Selected Areas in Communications, Vol. 17, nr. 8, Agosto de 1999.

[52] Ruiz, L. B., Nogueira, J. M., Loureiro, A. *Sensor network management*. Handbook of sensor networks: compact wireless and wired sensing systems. Edited by Mohammad Ilyas and Imad Mahgoub, CRC Press, pg. 57–84, Julho de 2004.

[53] Shen, C., Jaikaeo, C. Srisathapornphat, C. *Sensor Network Architecture and Applications*. Handbook of sensor networks: compact wireless and wired sensing systems. Edited by Mohammad Ilyas and Imad Mahgoub, CRC Press, pg. 154-166, Julho de 2004.

[54] Sohraby, K, Minoli, D., Znati, T. *Introduction and Overview of Wireless Sensor Networks.* Wireless Sensor Networks – Technology, Protocols, and Applications. Published by John Wiley & Sons, Inc., pg. 1-31, Janeiro de 2007.

[55] Sohraby, K, Minoli, D., Znati, T. *Medium Access Control Protocols for Wireless Sensor Networks.* Wireless Sensor Networks – Technology, Protocols, and Applications. Published by John Wiley & Sons, Inc., pg. 142-193, Janeiro de 2007.

[56] SQL Database Reference Material. *Sql Database Resources*. Disponível em http://www.sql.org, acessado em Agosto de 2007.

[57] Srisathapornphat, C., Jaikaeo, C., Shen, C. *Sensor information networking architecture and applications*. IEEE Personal Communication, pg. 52–59, Agosto de 2001.

[58] Su, W., Cayirci, E., Akan, O. B. *Overview of communication protocols for sensor networks*. Handbook of sensor networks: compact wireless and

wired sensing systems. Edited by Mohammad Ilyas and Imad Mahgoub, CRC Press, pg. 314-329, Julho de 2004.

[59] Vieira, M., *Embedded system for wireless sensor network*. Dissertação de Mestrado da Universidade Federal de Minas Gerais, Belo Horizonte-MG, Abril de 2004.

[60] Waneke, B., Lebowitz, B., Pister, K. S. J. *Smart dust: communicating with a cubic-millimeter computer*. IEEE Computer, pg. 2-9, Janeiro de 2001.

[61] Wang, H., Estrin, D., Girod, L. *Preprocessing in a tiered sensor network for habitat monitoring*, EURASIP J. Appl. Signal Process., Março de 2003.

[62] Wilschut, Annita N., Apers, Peer M.G. *Dataflow query execution in a parallel main-memory environment.* International Conference on Parallel and Distributed Information Systems - PDIS, pg. 68-77, Dezembro de 1991.

[63] Woo, A., Tong, T., Culler, D. *Taming the Underlying Challenges for Reliable Multihop Routing in Sensor Networks*. ACM SenSys, ACM Press, Novembro de 2003.

[64] Word Wide Web Consortium, W3C. *Extensible Markup Language (XML)*. Disponível em http://www.w3.org/XML/, acessado em Agosto de 2007.

[65] Yao, Y., Gehrke, J. *Query processing for sensor networks*. Conference on Innovative Data Systems Research – CIDR, pg. 233-244, Janeiro de 2003.

[66] Yao, Y., Gehrke, J. *The Cougar approach to in-network query processing in sensor networks*. SIGMOD Record, Vol. 31, N° 3, pg. 9-18, Setembro de 2002.

[67] Zhao, F., Guibas, L. *Wireless sensor networks – An information processing approach*. Morgan Kaufmann press, Julho de 2004.

APÊNDICES

Apêndice A. Algoritmos do PDCRS

A.1 Introdução

Nas subseções a seguir, serão apresentados os macro-algoritmos dos procedimentos relacionados com o mecanismo de processamento de consultas em RSSF objeto deste trabalho (PDCRS), conforme a plataforma executora (estação base ou nós sensores) e a fase em execução.

A.2 Procedimentos da estação base

Os procedimentos a seguir descritos, ativados na estação base, por definição são executados por módulos executores separados, e em paralelo. Isto permite que o controle do tempo de vida da consulta pela estação base seja independente da interface de gerenciamento feito pela aplicação usuária.

5.2.1 Obtenção da consulta

As iterações aqui descritas (Figura A.1) demonstram o fluxo do processamento das etapas de obtenção do código da consulta (escrita em linguagem declarativa SNQL), interpretação da consulta, injeção da consulta na rede (através do módulo de processamento da estação base) e pós-processamento dos resultados provenientes dos nós sensores.

```
Obtenção da consulta

1: [Enquanto aplicação.ativada faça
2:   [ObtemNovaConsulta(SNQL)
3:   [RealizaParsingConsulta(SNQL => Q)
4:   [DecompõeConsulta(Q => Qdistribuída[Q0, Q1,...Qn])
5:   [InjetaConsulta(Q) //Estação Base: Processamento da Consulta
6:   [Enquanto não mudar consulta e aplicação.ativada faça
7:     [ObtemResultadosProvenientesDaConsulta(Q.R)
8:     [ApresentaResultadosDaConsulta(Q.R)
9:   [Fim enquanto
10: [Fim enquanto
```

Figura A.1 Algoritmo gerenciamento para obtenção da consulta.

Enquanto a aplicação estiver ativa (linha 1), a interface para obtenção de consultas permitirá a entrada de uma nova consulta ou alteração da consulta

corrente em processamento (linha 2). Para cada nova consulta informada em linguagem declarativa, será realizada a sua interpretação (linha 3), e análise semântica baseada no esquema global ativo, e geração do plano de execução 'bruto', ou seja, sem considerar a decomposição em subconsultas. Em seguida, será realizada a decomposição da consulta (Q) em subconsultas, conforme as categorias de sensores envolvidas ou a sua região de localização (linha 4). Após a decomposição, será realizada a injeção da consulta a partir de chamada ao módulo de processamento. Após a injeção da nova consulta (linha 5) através módulo executor do processamento (procedimento descrito na subseção 5.2.2), esta interface entrará no estado de processamento final para obter continuamente os resultados advindos dos nós sensores (também através do módulo executor de processamento), para a consulta corrente (linha 6 e 7), e apresentá-los à aplicação usuária (linha 8).

5.2.2 Controle e processamento da consulta

Este módulo realiza a injeção das subconsultas na rede, disseminado-as aos conjuntos de sensores destinatários de acordo com a sua categoria de sensoriamento, e mantém o controle sobre o tempo de vida da consulta em execução, recebendo os pacotes de respostas continuamente, provenientes dos nós sensores. Também neste módulo devem ser realizadas as funções de pós-processamento, a exemplo de uniões, junções, agrupamentos, entre outras tarefas necessárias à consolidação dos resultados, para em seguida repassar ao módulo de gerenciamento da interface (ver subseção 5.2.1) com usuário uma única fonte de dados resultante.

```
Processamento na estação base

1: [Enquanto aplicação.ativada faça
2:   [ObtemNovaConsultaInjetada(Q)
3:   [DisseminaSubConsultas(Q1,...,Qn)
4:   [Inicialize tempoTimeWindow
5:   [Enquanto tempoTimeWindow < Q.TimeWindow e não mudar consulta faça
6:     [ObtemResultadosConsulta(Q.R)  //Nós sensores
7:     [FazPósProcessamentoResultados(Q.R => Q.Rp)
8:     [RetornaResultados(Q.Rp)  //Estação Base: Gerenciamento
9:   [Fim enquanto
10: [Fim enquanto
```

Figura A.2 Algoritmo processamento da consulta na estação base.

Conforme a Figura A.2, enquanto a aplicação estiver ativa (linha 1), o módulo obterá eventual nova consulta advinda do módulo de interface com o usuário (linha 2). Esta consulta já terá sido interpretada e decomposta, e será recebida aqui no formato estruturado. Será realizada a disseminação dos fragmentos na rede (linha 3), ou seja, transmissão das subconsultas para os nós sensores filhos da estação base, que serão responsáveis por enviar para os seus filhos, e assim sucessivamente, até alcançar todos os sensores das categorias envolvidas.

Após a distribuição dos fragmentos de consultas, serão inicializados controles (linha 4) para manter a consulta ativa, durante a vigência do tempo de vida desta (parâmetro *Time Window*). A iteração iniciada na linha 5 manterá ativa a consulta enquanto não for sinalizada nenhuma alteração de parâmetros, e controlará o pós-processamento dos resultados: na linha 6, recebe os dados gerados pelos sensores; na linha 7, realiza o pós-processamento dos resultados (junções, agrupamentos, etc.); e na linha 8, retorna os resultados continuamente para o módulo de gerenciamento da interface com o usuário.

A.3 Nós sensores: processamento de subconsultas

Assumiu-se neste trabalho uma arquitetura simplificada para os dispositivos nós sensores, que não têm capacidade de executar mais de um conjunto de instruções em paralelo, o que definiu o modelo do seu algoritmo de processamento, que necessita ser executado de forma seqüencial, não obstante poderem existir outras arquiteturas de nós sensores capazes de realizar mais que uma atividade simultaneamente.

Conforme Figura A.3, enquanto o nó sensor estiver operacional (linha 1), ele realizará seqüencialmente as operações descritas: receberá uma nova consulta Qi (subconsulta) a ser instalada (linha 2), que corresponde a um fragmento de Q, gerado na estação base, referente a categoria de sensoriamento da qual este nó faz parte. Serão inicializados controles (linha 3) para manter a consulta ativa, durante a vigência do seu tempo de vida (parâmetro *Time Window*).

Na seqüência, a iteração mostrada (linha 4) define que o sensor tratará esta consulta durante a vigência definida, ou enquanto não houver alteração da consulta, e enquanto houver recursos de energia neste nó[12].

```
Processamento nos nós sensores: Principal

1:[Enquanto nó.funcional faça
2:  [ObtemNovaConsulta(Qi) //Qi::subconsulta de Q original
3:  [Inicialize tempoTimeWindow
4:  [Enquanto tempoTimeWindow < Qi.TimeWindow e Sensor.Energia>0
     e não mudou consulta faça
5:    [MonitorarRecursos;
6:    [Inicialize tempoSendInterval;
7:    [Inicialize tempoSenseInterval;
8:    [Enquanto tempoSendInterval < Qi.TimeSend e Sensor.Energia>0
       e não mudou consulta faça
9:      [Se tempoSenseInterval >= Qi.SenseInterval então
10:       [Inicialize tempoSenseInterval
11:       [ColeteDadosDoAmbiente(NovaTupla)
12:       [EmpacoteDados(NovaTupla)
13:     [Senão
14:       [RecebePacotes
15:     [Fim Se
16:   [Fim Enquanto
17:   [EnviaPacotes
18: [Fim Enquanto
19:[Fim Enquanto
```

Figura A.3 Algoritmo geral de processamento nos nós - Sensores.

Na linha 5 há a primeira chamada na função de monitoração de recursos dentro do ciclo, para ajustar as variáveis de controle de tempo e gerenciamento de energia de memória, conforme os parâmetros da consulta. Na linha 6 e 7 ocorrem as inicializações das variáveis que controlam o tempo de espera para transmissões de pacotes (com base no parâmetro *Send Interval*) e o tempo de acúmulo de dados coletados no sensoriamento local (com base no parâmetro *Sense Interval*).

O segundo *loop* definido na linha 8 especifica o ciclo de aquisições de dados de sensoriamento antes que os resultados possam ser transmitidos para a rede (conforme parâmetro *Send Interval* da consulta). A transmissão dos resultados se fará conforme a estratégia de roteamento escolhida. Dentro dessa iteração será controlado o tempo de amostragem (linhas 8 e 9) de aquisição de dados locais (sensoriamento) conforme definido no parâmetro *Sense Interval*, e a aquisição do valor do sensor propriamente dito (linha 11). Na linha 12 a nova tupla adquirida é

[12] Obviamente, se acabar a energia, o nó deixará de funcionar. Este teste na iteração apenas destaca este fato, para efeito de clarificar o algoritmo.

submetida ao empacotamento, onde serão executadas os procedimentos de filtragens (projeção e seleção da consulta).

São realizadas as atividades de monitoração de recursos de memória e energia, além da recepção de pacotes de dados provenientes de outros nós, também de forma seqüencial, durante o tempo em que não se faz amostragem para aquisição (linha 14), além do controles das áreas de transmissão e recepção, utilizados pelas estratégias de agregação e roteamento. Finalmente, a cada conclusão de ciclos de aquisição, são transmitidos os resultados dos dados coletados para a estação base (linha 17).

```
Processamento nos nós sensores: Obtemnovaconsulta

1:[Enquanto aplicacao.ativada e não ProcessandoConsulta faça
2:  [RecebePacotes
3:  [EnviarPacotes
4:[Fim Enquanto
```

Figura A.4 Algoritmo do bloco ObtemNovaConsulta - Sensores.

A algoritmo descrito na Figura A.3 é compatível com a estratégia adotada em [8], pela sua estrutura linear de processamento, mas destacam-se aqui as seguintes diferenças:

i) Inclusão de extensão de código para tratar os procedimentos de obtenção da consulta injetada na rede de sensores (iteração inicial linha 1, e atividade da linha 2). Ver também a Figura A.4;

ii) Monitoração dos recursos de energia e memória realizada dentro da iteração de recepção de pacotes (linha 7, Figura A.5) e no início do ciclo de sensoriamento (linha 5, Figura A.3). Isto garante a atualização das variáveis de controle conforme os parâmetros correntes da consulta;

iii) Durante o ciclo de espera para transmitir pacote (*send interval*), será realizada verificação para receber dados de outros sensores, evitando assim situações de perda de pacotes enviados por outros nós durante o intervalo dedicado ao acumulo da coleta das amostragens de sensoriamento (Linha 13 e 14, Figura A.3). Esta

ação também serve para monitorar comandos e alterações na consulta, advindos da estação base.

iv) O bloco de recepção de pacotes provenientes de outros nós não monopoliza as funções do nó sensor, em casos em que há grande fluxo de mensagens sendo transmitidas dentro da RSSF (quando há um grande número de sensores envolvidos), pois somente é recebida a quantidade de mensagens disponíveis no momento em que o bloco é chamado. Caso haja novas mensagens recebidas após estar dentro da iteração do bloco, as demais serão processadas somente no próximo ciclo. Ver algoritmo na Figura A.6.

```
Processamento nos nós sensores: RecebePacotes

1:[Se AreaDeRecepcao.Count = 0 então
2:[  Retorne;                        // nada a fazer aqui
3:[Fim se
4:[QtdMsgRx ← AreaDeRecepcao.Count;  // quantas mensagens esperando
5:[QtdMsgTrat ← 0;                   // quantas mensagens já tratadas
6:[Enquanto (QtdMsgTrat < QtdMsgRx) faça
7:  [MonitorarRecursos;
8:  [NovaMensagem ← AreaDeRecepcao.PegaMensagem
9:  [QtdMsgTrat ← QtdMsgTrat+1
10  [Se NovaMensagem.Tipo = DadosDeSensor então
11:   [Se NovaMensagem.Destino = EsteGrupoSensor então
12:     [Se ProcessandoConsulta entao
13:       [PacoteDeTrabalho ← ProcessaAgregaçãoEmRede(NovaMensagem)
14:     [Fim Se
15:   [Senão
16:     [PoePacoteNaAreaDeEnvio(NovaMensagem)
17:   [Fim se
18: [Senão
19:   [ProcessaMensagemRecebida(NovaMensagem);
20: [Fim Se
21:[Fim Enquanto
```

Figura A.5 Algoritmo do bloco RecebePacotes - Sensores.

```
Processamento nos nós sensores: ProcessaMensagemRecebida(NovaMensagem)

1: [Caso NovaMensagem.Tipo seja:
2:   [SubConsulta:
3:     [TrataParâmetrosDaSubConsulta (Nova ou alteração de parâmetros);
4:     [ProcessandoConsulta ← Sim;
5:   [PararConsulta:
6:     [ProcessandoConsulta ← Não;
7: [Fim Caso
```

Figura A.6 Algoritmo do bloco ProcessaMensagemRecebida - Sensores.

A lógica de empacotamento de tuplas (contendo os dados coletados localmente pelo nó sensor) está descrita no macro-algoritmo apresentado na Figura A.7.

```
Filtragem e empacotamento de novas tuplas coletadas no nó sensor

1: [Se há Memória Disponivel para NovaTupla então
2:   [ConsultaAtiva.AssinalaTupla(SensorsCateg,NovaTupla,HeadProjeção);
3:   [Se ConsultaAtiva.FiltroWhere.SelecionaTupla então
4:     [Se ainda não existia no pacote tupla com esta mesma chave então
5:       [Insere esta nova tupla no PacoteDeTrabalho
6:     [Senão // se já existia tupla no pacote, com essa chave
7:       [Agrega os valores da nova tupla no pacote de trabalho
8:     [FimSe
9:  [FimSe
10:[FimSe
```

Figura A.7 Algoritmo de empacotamento de nova tupla coletada – Sensores.

Conforme apresentado na Figura A.7, o aproveitamento da nova tupla coletada é limitado pela memória disponível no nó sensor para expandir o pacote de trabalho (Linha 1). A linha 2 indica o procedimento de seleção horizontal da tupla (itens de projeção da subconsulta que indicarão quais colunas da nova tupla farão parte deste processamento). A Linha 3 faz a seleção vertical desta tupla (cláusula *WHERE*), onde somente as tuplas que passarem por este filtro é que farão parte do próximo procedimento. O teste que vai da linha 4 a linha 8 verifica se já existia tupla empacotada possuidora de chave igual à da nova tupla. Dependendo disto, os novos valores poderão ser agregados em acumuladores anteriores, ou servirão como nova entrada no pacote de trabalho corrente.

Apêndice B. Simulador de rede de sensores sem fio

B.1 Introdução

Este apêndice apresenta a implementação de um simulador de uma rede de sensores sem fio, o qual denominamos "SNETSIM" [47], formado por um conjunto de aplicações desenvolvidas com o objetivo de auxiliar na prova de conceito do mecanismo de distribuição e processamento de consultas contínuas proposto nesta dissertação (PDCRS).

As definições correntes do simulador aqui descrito não pretendem englobar todos os aspectos necessários existentes num modelo de uma RSSF real, mas dar o suporte mínimo para o desenvolvimento de estratégias de processamento de consultas contínuas, e facilitar a implementação das aplicações usuárias do nosso mecanismo proposto, pela ocultação de detalhes inerentes a protocolos de comunicação e controle do fluxo dos pacotes trafegados entre os módulos conceituais.

B.2 Motivação

Para dar suporte ao desenvolvimento do mecanismo de processamento e distribuição de consultas em redes de sensores sem fio, detalhado no Capítulo 5 desta dissertação, foi necessário preparar um cenário mínimo para as provas de conceito do mecanismo PDCRS. Assim, a criação de aplicações usuárias das classes que implementam as fases do processamento e distribuição das consultas, foi o ponto de partida para a criação de um ambiente objetivo que simulasse uma rede de sensores, usando como plataforma operacional microcomputadores padrão PC, que podem estar interligados em rede baseada nos protocolos TCP/IP. Denominamos este simulador pelo acrônimo SNETSIM (*Sensor NETwork SIMulator*).

O cenário demandado para as provas de conceito do PDCRS provê uma estação base como nó central, logicamente circundada por nós sensores distribuídos aleatoriamente em diversos níveis de localização em relação a núcleo da rede. Conforme as definições próprias do PDCRS e implementadas no

SNETSIM, os nós sensores podem se comunicar entre si e com a estação base. Neste cenário, a iniciação da comunicação se dá conforme o roteamento adotado, ou seja, sobre quais sensores podem enviar pacotes para outro conjunto de sensores.

A RSSF simulada é definida como sendo um conjunto de nós distribuídos num espaço bidimensional de raio r, em cujo centro (base.x=0, base.y=0) encontra-se a estação base. Cada nó sensor encontra-se dentro deste espaço, em sua posição (sensor.x, sensor.y), cuja capacidade de comunicação com os demais nós é definida como sendo a sua potência de alcance. Estes parâmetros (raio da RSSF, posições dos nós sensores e potência de alcance dos sensores) podem ser customizados na configuração de cada nó, ou determinados aleatoriamente na carga da aplicação.

A simulação consiste na submissão de consultas a partir da estação base, que as interpretará e distribuirá os seus fragmentos aos nós sensores, conforme a categoria de sensoriamento da qual fazem parte, até que os pacotes da consulta alcancem todos os nós da rede. Estas distribuições consideram as questões relacionadas com o tipo de sensoriamento que cada nós estiver apto a realizar.

B.3 Categorização das estações de trabalho

O SNETSIM é composto por 3 (três) tipos de estação de trabalho, cada um com o seu papel bastante específico dentro desta arquitetura:

(i) Estação base: concentrará os módulos relativos ao nó central da rede:

   a. Módulo interface com usuário (*Base Station Manager*), onde o usuário submete consultas escritas em SNQL;

   b. Módulo processamento de consultas (*Base Station Processor*): realiza a comunicação com os nós sensores, controla o tempo de vida da consulta e faz o pós-processamento desta.

(ii) Nós sensores: estação que executa o módulo que implementa os nós sensores distribuídos pela rede (*Sensor Node Processor*). Simula sensores de diversas categorias. Pode haver mais que uma estação de sensores na rede real, onde cada uma pode simular um ou mais sensor. Cada sensor

simulado, ao ser criado, terá uma identificação única, gerada automaticamente pelo simulador;

(iii) Agente de rede: estação que executa o módulo de comunicação dentro da rede de sensores sem fio (*Network Agent*). Simula o meio físico por onde as comunicações entre os módulos trafegam (nuvem da rede). Mantém os canais de comunicação entre os nós virtuais da RSSF, simulando *broadcast* como uma estação de rádio, a partir dos pacotes enviados pelas demais estações.

Cada estação do simulador pode ser materializada dentro de um ou mais computadores executando os módulos de serviços (aplicativos executáveis) relativos a cada tipo de estação a ser operacionalizada. Para que uma simulação ganhe em termos de escalabilidade, as estações podem ser instaladas em mais de um computador que participem da mesma rede hospedeira (rede nativa dos computadores). Assim, é transparente para a aplicação SNETSIM se os nós virtuais estão dispostos localmente (na mesma máquina), ou disseminados através de uma rede real. A Figura B.1 mostra o diagrama com a interação entre os módulos de serviços do simulador.

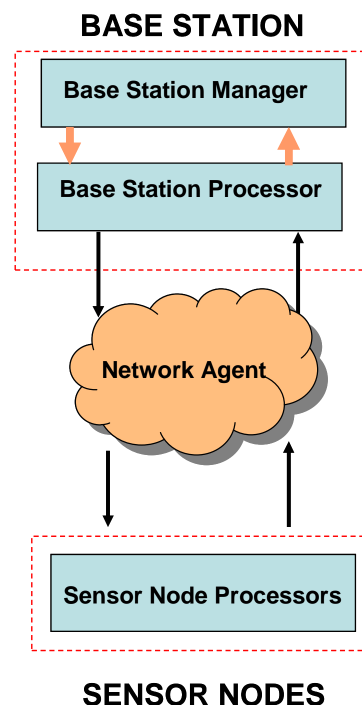


Figura B.1 Diagrama com interação das estações da SNETSIM.

B.4 Componentes para aplicações SNETSIM

Cada um dos módulos que compõem a SNETSIM representa uma aplicação usuária do mecanismo PDCRS. Este mecanismo, representado pelo conjunto de componentes e classes reutilizáveis, implementam rotinas que são responsáveis pelas funções em alto nível de comunicação entre os módulos. De forma transparente, apenas utilizando-se de métodos dos componentes PDCRS, esses módulos trocam mensagens entre si, que podem carregar tanto as consultas a serem injetadas na rede, como os resultados obtidos após o seu processamento ou consolidação. O componente principal utilizado na comunicação entre os módulos é o PDCRS *Comm*, que disponibiliza uma camada com as rotinas de comunicação que deixam transparente para o aplicativo do usuário os detalhes de como os pacotes trafegam pela rede.

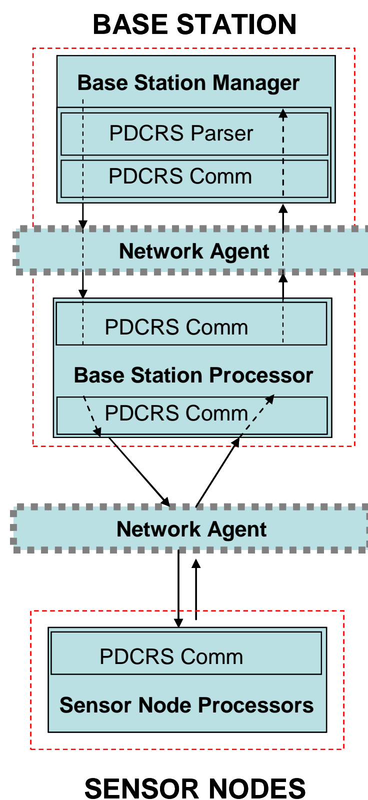


Figura B.2 Diagrama com a comunicação entre os módulos do PDCRS.

Conforme ilustrado na Figura B.2, o módulo *Base Station Manager* utiliza diretamente o componente PDCRS *Parser*, responsável pelas rotinas de interpretação da consulta emitida em SNQL. Este componente, implicitamente, utiliza o componente PDCRS Comm para trocar mensagens com o módulo *Base Station Processor*. O módulo *Base Station Processor*, por sua vez, utiliza diretamente o componente PDCRS *Comm* para trocar mensagens para os módulos *Base Station Manager* e o módulo *Sensor Node Processor*.

Observação: toda a comunicação entre os módulos, que podem estar em estações distintas dentro da rede real, é feita através do módulo *Network Agent*, responsável pelo roteamento das mensagens trafegadas dentro da rede real TCP/IP. Assim, os detalhes relativos a este protocolo de rede ficam encapsulados dentro das rotinas do componente PDCRS *Comm*.

B.5 Operação da rede SNETSIM

Os módulos da SNETSIM podem ser instalados independentemente um do outro, mas a funcionalidade de cada módulo somente torna-se ativa quando da descoberta do módulo agente de rede (*Network Agent*) ativo. No caso dos módulos vinculados à estação base, além da dependência do agente de rede, os módulos *Base Station Manager* e *Base Station Processor* também são interdependentes entre si, apesar de poderem estar em estações físicas (computadores) distintos dentro da rede TCP/IP real.

Ao ser ativado, qualquer nó primeiramente busca descobrir onde se encontra o agente de rede, enviando anúncio (*discover request*) a partir de um pacote de *broadcast*. Se existir um agente de rede ativo, este responde, enviando um pacote de resposta (*discover response*) destinado ao nó anunciante. Cada anúncio de *discover request* recebido pelo *Network Agent* serve também para que este agente atualize uma tabela de nós ativos, para controlar quais deles receberão os pacotes trafegados pela rede, simulando assim uma grande rede de *broadcast*, tal como uma estação de rádio. Assim, por definição, todos os pacotes enviados pela rede serão fisicamente escutados por todos os nós (não obstante existir no formato da mensagem indicações acerca do nó originador da mensagem e o seu destinatário).

Em tempo de operação, a cada consulta injetada na rede pela Estação Base, as subconsultas derivadas geradas são identificadas conforme a categoria de sensoriamento dos nós destinatários, e disseminadas via *broadcast* pelo agente de rede. Cada nó sensor, apesar de escutar todos as mensagens trafegadas, aceitará somente aquelas originadas pelos nós configurados como seus pais, e processará somente aquelas mensagens destinadas à sua categoria. Os resultados do processamento local serão enviados pelo nó aos seus nós pais, que, após aplicar as estratégias adotadas (a exemplo de agregação em rede), irão retornando os pacotes até atingirem a estação base, que fará o pós-processamento e repassará os resultados finais ao usuário.

B.6 Comunicação entre os módulos

Considerando que toda a comunicação entre os módulos é feita através do agente de rede (*Network Agent*), este componente necessita ser 'descoberto' pelos demais módulos antes que eles possam iniciar a comunicação entre si dentro da rede real. O agente de rede é implementando como um servidor de comunicação, e aceita conexões de vários clientes (módulos) simultaneamente.

Para um módulo se conectar ao agente de rede, ele precisa inicialmente descobrir em qual estação física da rede este agente está sendo executado. O procedimento de descoberta baseia-se em enviar uma mensagem multicast via UDP (*User Datagram Protocol)*, denominada *Discover Request*. Este tipo de mensagem será respondido, exclusivamente, pelo módulo *Network Agent*, com outra mensagem UDP, denominada *Discover Response*, direcionada ao nó requisitante, o qual, ao receber esta resposta, terá as informações acerca da localização do agente na rede. As portas de comunicação utilizadas para os procedimentos de descoberta do *Network Agent* são denominadas Canal de Descoberta (*DiscoveryChannel*).

Após o nó requisitante descobrir o agente de rede, esse realiza uma conexão TCP com o agente, e envia uma mensagem inicial denominada *Open Request*, requisitando informações do agente para efeito de configuração de alguns aspectos da RSSF simulada. Em resposta, o agente responde com uma mensagem *Open Response*. A partir desta etapa, toda a comunicação é feita por esse canal, que é

denominado Canal de Dados (*DataChannel*). Os procedimentos de descoberta e conexão com o agente de rede são sintetizados na Figura B.3.

Nós Sensores e Estações: Descoberta e conexão com Network Agent

```
1:[Enquanto Sensor não conectado com o Network Agent Faça
2:  [Envia mensagem Discover Request (UDP)
3:  [Aguarda mensagem Discover Response (UDP)
4:  [Se Recebeu Discover Response então
5:    [Obtem Endereço Network Agent
6:    [Realiza conexão TCP com o Network Agent pelo DataChannel
7:    [Envia mensagem Open Request pelo DataChannel
8:    [Aguarda mensagem Open Response do DataChannel
9:    [Recebe Open Response (sinalização de conexão ok)
10:   [Ajusta configurações do nó, conforme Open Response
11: [Fim Se
12:[Fim Enquanto
```

Figura B.3 Algoritmo para descoberta e conexão com o Agente de rede.

Não obstante cada nó ficar com a sua própria conexão TCP com o *Network Agent*, toda mensagem enviada por qualquer componente da rede pela canal de dados será sempre replicado para os demais canais, simulando um *broadcast* real, a exemplo do que acontece numa rede em que o meio físico é provido por rádio. O protocolo de roteamento utilizado pelo PDCRS será responsável pela filtragem das mensagens em cada nó. A Figura B.4 mostra os algoritmos de comunicação processados pelo agente de rede.

Network Agent: DiscoveryChannel

```
1:Enquanto Aplicação.Ativa Faça
2:  Aguarda mensagens Discover Request (UDP)
3:  Recebe mensagem Discover Request de nó requisitante
4:  Envia mensagem Discovery Response para nó requisitante
5:Fim Enquanto
```

Network Agent: Aceitando novas conexões no DataChannel

```
1:Enquanto Aplicação.Ativa Faça
2:  Aguarda conexões estações clientes no DataChannel (TCP)
3:  Aceita e mantém aberta nova conexão no DataChannel
4:Fim Enquanto
```

Network Agent: Tratando requisições de nós clientes via DataChannel

```
1:Enquanto Aplicação.Ativa Faça
2:  Recebe nova mensagem no DataChannel
3:  Se tipo da mensagem recebida igual a OpenRequest então
4:    Envia mensagem OpenResponse para o nó requisitante
5:  Senão
6:    Reenvia para todos os nós da rede (broadcast DataChannels)
7:  Fim se
8:Fim Enquanto
```

Figura B.4 Algoritmo para tratamento de comunicação pelo agente de rede.

B.7 Implementação do simulador SNETSIM

O simulador SNETSIM é composto por 04 (quatro) aplicações, representadas pelos módulos executáveis (programas) listados a abaixo.

- NetAgent.exe: aplicação "*Network Agent*". É o agente de rede, que simula a rede da RSSF. Todos os demais módulos se comunicam através deste módulo, que tem o papel de redirecionador de mensagens, e mantenedor dos canais de descoberta e de dados.
- BaseMan.exe: aplicação "*Base Station Manager*". Interface do gerenciador da interface com usuário, responsável pela submissão, interpretação e decomposição de consultas.
- BaseProc.exe: aplicação "*Base Station Processo*r". Controle da execução da consulta injetada na RSSF. Responsável pela disseminação das subconsultas e pós-processamento dos resultados.
- SensorProc.exe: aplicação "*Sensor Node Processor*". Faz a simulação dos nós sensores dentro da RSSF. Responsável pelo processamento das subconsultas, coleta de valores de sensoriamento em cada nó sensor e estratégias de agregação em rede.

As aplicações citadas a seguir podem ser instaladas no mesmo computador ou em computadores distintos dentro da rede real, quando da ativação do simulador.

A corrente implementação foi realizada sob o sistema operacional *Windows* XP, codificada em *Object Pascal.* Uma das premissas para sua programação foi utilizar o mínimo de componentes e classes nativas de ferramentas de programação que implementam esta linguagem, para que seja possível migrar de ambiente de desenvolvimento com o menor esforço possível, quando necessário. Isto permite que o código fonte seja compilado por ferramentas de desenvolvimento que utilizam o padrão *Object Pascal*, sem grandes esforços para compatibilização,

a exemplo das ferramentas da empresa *Borland (Delphi Win32 e Turbo Delphi* [17]) ou das ferramentas de código livre *FreePascal e Lazarus Project* [36][20].

Dentro do escopo dos nossos trabalhos futuros, está a conclusão da migração do código das classes aqui descritas para outras linguagens de programação, tais como linguagem *C* e linguagem *Java*.

Nas seções a seguir são mostrados informações particularizadas acerca da implementação de cada um dos módulos que fazem o simulador SNETSIM.

B.7.1 Módulo Network Agent (NetAgent)

Esta aplicação deve ser executada antes dos demais módulos do simulador. Pode ser instalada em qualquer estação da rede real (TCP/IP), a qual fará parte da RSSF simulada. Os seus arquivos de trabalho são:

1) NetAgent.exe: programa executável, no formato binário Win32;
2) SiteConfig.xml: configuração da rede, no formato texto XML. Este arquivo é carregado pela aplicação NetAgent no momento da carga, e deverá estar na mesma pasta (diretório) da aplicação. A subseção B.7.5, abaixo, descreve o formato e os parâmetros presentes nesse arquivo;

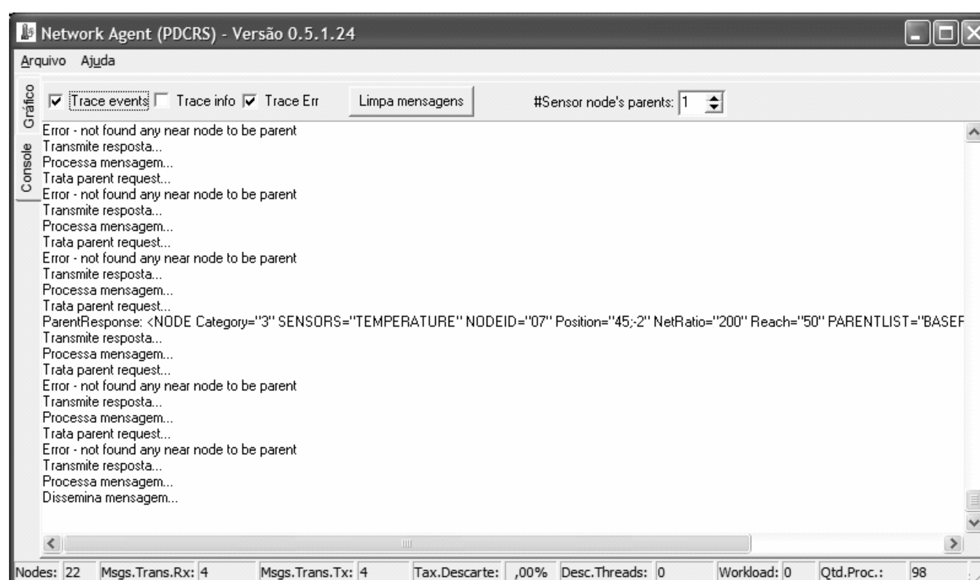


Figura B.5 Tela da console do Agente de Rede.

Ao ser carregado (Figura B.5), o NetAgent entra em modo de espera de conexões TCP (relativas aos nós sensores da rede). Estas conexões são denominadas canal de dados (*DataChannel*), por onde as aplicações se comunicarão, a partir do redirecionamento de mensagens feito por este agente.

Também ao ser carregado, o agente de rede fica no aguardo de pacotes *multicast* UDP, que representam as atividades de descoberta da sua estação da rede real pelas demais aplicações (*DiscoveryChannel*).

04 (quatro) opções de rastreamento (trace) desejados das atividades do agente de rede podem ser ativados: (i) rastreamento de eventos de conexões (*TraceEvents*); (ii) rastreamento de quaisquer informações de comunicação (*TraceInfo*); (ii) rastreamento de erros (*TraceErr*); e (iv) rastreamento gráfico. As três primeiras opções de rastreamento, se ligadas, geram mensagens na console do agente (ver Figura B.5). A gravação das mensagens de rastreamento em arquivos de *logs*[13] é possível, desde que na configuração de *logs* do SiteConfig.xml isto seja definido.

O rastreamento gráfico (Figura B.6) permite acompanhar graficamente o tráfego de mensagens entre os nós conectados à rede, inclusive configurar um nó específico para rastreamento individual. Permite também verificar graficamente a paternidade dos nós da rede.

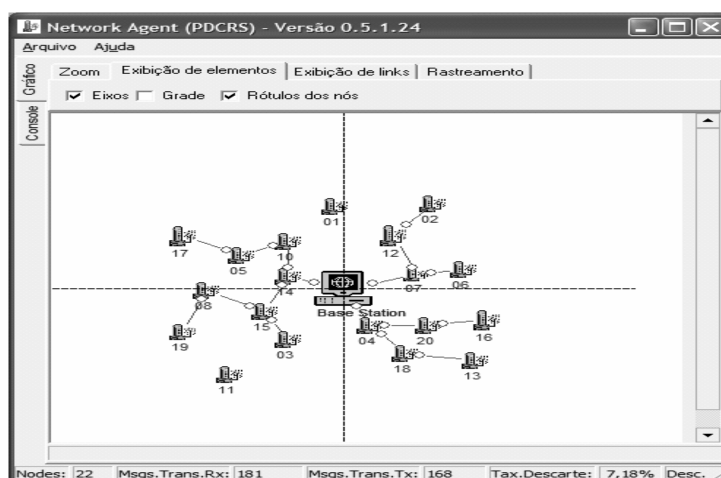


Figura B.6 Tela de rastreamento gráfico de comunicação na RSSF – *NetAgent*.

B.7.2 Módulo Base Station Manager (BaseMan)

Esta aplicação representa a porta de entrada das consultas a serem tratadas pelo PDCRS. É a interface com o usuário demandante, oferecendo controles amigáveis para a edição e manutenção de consultas contínuas, antes de as mesmas serem injetadas na RSSF. Os seus arquivos de trabalho:

1) BaseMan.exe: programa executável, no formato binário Win32;
2) SiteConfig.xml: configuração da rede, no formato texto XML. Este arquivo é carregado pela aplicação BaseMan no momento da carga, e deverá estar na mesma pasta (diretório) da aplicação. A subseção B.7.5, abaixo, descreve o formato e os parâmetros presentes nesse arquivo;
3) GlobalSchema.xml: catálogo com o esquema global das relações envolvidas com as consultas contínuas a serem emitidas. A subseção B.7.6 descreve a estrutura deste catálogo.

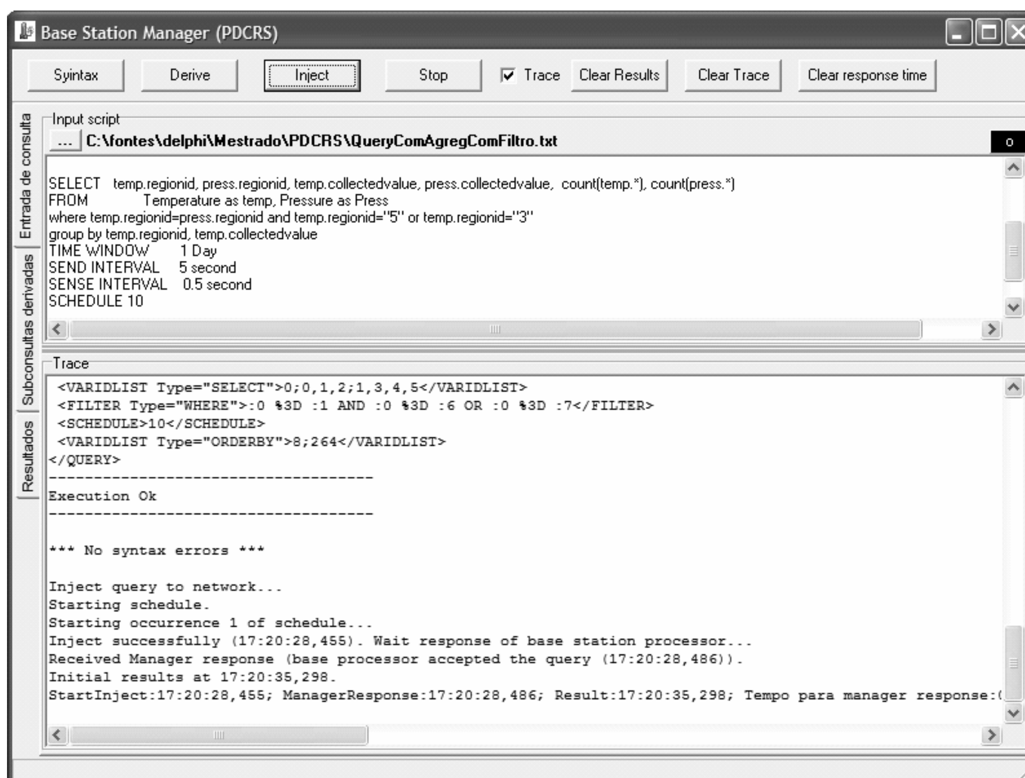


Figura B.7 Tela principal da aplicação *Base Station Manager*.

[13] *Logs*: registros históricos das ocorrências relevantes, que pode ser utilizado posteriormente em diagnósticos de eventuais problemas.

Ao ser carregado, o BaseMan poderá entrar em modo de procura automática da aplicação agente de rede (NetAgent), que poderá estar em qualquer estação dentro da rede real (utilização do *DiscoveryChannel*, via protocolo UDP). Este procedimento será executado se o parâmetro <DiscoveryChannel>Auto=”yes”, do SiteConfig.xml, estiver ativo. Caso contrário, esta aplicação assumirá a configuração do *DataChannel* presente no arquivo SiteConfig.xml.

Depois de configurado ou descoberto o *DataChannel*, o módulo BaseMan se conecta, via TCP ao agente de rede. A conexão ao *DataChannel* por esta aplicação servirá exclusivamente para a sua comunicação com a aplicação *Base Station Processor* (BaseProc). O módulo NetAgent somente aceitará a conexão do *Base Station Manager* no canal de dados se a aplicação *Base Station Processor* (NetProc) já estiver conectada na rede. Isto é obrigatório pelo fato de que toda a comunicação do módulo BaseMan ser logicamente direcionada somente para o BaseProc.

Em paralelo às atividades de conexão com a rede, após ser carregada, a aplicação BaseMan já estará pronta para receber entrada de consultas pelo usuário, a partir de sua interface (Figura B.7). Os controles ou campos disponíveis nessa tela principal estão descritos a seguir:

Caixas de texto:

- [*Input Script*]: campo de texto para digitação de nova consulta contínua em linguagem SNQL, que será injetada na RSSF. A consulta também pode ser restaurada (lida) a partir de um arquivo de texto previamente armazenado (botão [...] do controle [*Input Script*]). O texto aqui entrado também pode ser um *script* SNQL, contendo mais de um comando DDL, mas somente um comando DML (consulta). Os comandos devem ser separados por “;” (ponto-e-vírgula);
- [SNQL *subqueries*]: mostra resultado da decomposição da consulta entrada no campo [*Input Script*], após ser clicado no botão [*Derive*] ou [*Inject*];

- [XML *subqueries*]: mostra o resultado da decomposição da consulta entrada no campo [*Input Script*], após ser clicado no botão [*Derive*] ou [*Inject*]. Este formato é o que será utilizado para o transporte e processamento da consulta nos demais módulos;
- [*Trace*]: caixa de texto dos resultados do rastreamento das operações desta aplicação, contendo mensagens de erro e o passo a passo da interpretação das consultas;
- [Resultados]: caixa onde são apresentados os resultados (dados) advindos do processamento nos sensores;
- [Tempos de resposta]: lista com os tempos de resposta de cada sensor identificado na rede.

Botões:

- [*Sintax*]: apenas verifica a sintaxe do script SNQL entrado no campo [*Input Script*];
- [*Derive*]: verifica a sintaxe do script SNQL entrado no campo [*Input Script*], faz a análise semântica conforme o esquema global utilizado e também mostra a decomposição da consulta, mas não injeta a consulta na rede;
- [*Inject*]: realiza todas as fases de processamento do *script* entrado no campo [*Input Script*], e também injeta a consulta na rede: envia todo o script (já em XML) para o módulo *Base Station Processor*. Esta opção só estará disponível se a conexão com o *DataChannel* do agente de rede tiver sido realizado com sucesso. Este comando também serve para injetar novamente uma mesma consulta na rede, apenas alterando algum de seus parâmetros, de forma transparente;
- [*Stop*]: cancela a execução de uma consulta contínua que havia sido injetada na rede. Isto fará com que este módulo envie um comando *Stop* Query para o módulo *Base Station Processor*, que por sua vez disseminará um pacote de *Stop Query* pela RSSF, fazendo os nós sensores interromperem a sua atividade de processamento;

- [*Clear Results*]: limpa a tela de resultados de consultas;
- [*Clear Trace*]: limpa a caixa de texto [*Trace*];

Outros controles:

- Caixa de checagem [*Trace*]: liga/desliga rastreamento dos eventos;
- Botão [...]: carregar um script a partir de um arquivo de texto;
- Botão [o]: salvar o texto da caixa [*Input Script*] num arquivo de texto.

Após a entrada de uma nova consulta e a sua injeção na rede, este módulo entra em espera para receber os resultados pós-processados advindos da aplicação *Base Station Processor*.

Os resultados recebidos serão mostrados na caixa tela “Xml *Results*”. Como se tratam de aplicações pilotos para a prova de conceito do PDCRS, não são efetuados maiores tratamentos sobre os resultados apresentados pelos módulos da estação base, se limitando o *Base Station Manager* a mostrar textualmente as linhas dos resultados. Para cada nó sensor identificado a partir dos resultados obtidos, será calculado o seu tempo de resposta, que é o tempo decorrido a partir da injeção da consulta até o recebimento do primeiro pacote do nó em questão. A lista com os tempos de resposta são mostrados na aba de resultados.

Estando uma consulta contínua sendo processada pela rede, a aplicação BaseMan pode até ser encerrada, sem que o processamento da consulta seja interrompido. Isto é possível porque todo o controle, a partir dessa fase, é realizado pelo módulo BaseProc (*Base Station Processor*). Nesse caso, assim que a aplicação *Base Station Manager* for restaurada, esta voltará a receber continuamente os resultados da consulta ativa.

B.7.3 Módulo Base Station Processor (BaseProc)

Esta aplicação representa o núcleo do controle sobre uma consulta contínua em processamento pela RSSF, pois é responsável pela disseminação das subconsultas geradas pela fase de decomposição realizada pelo módulo BaseMan, e também é responsável por manter a consulta “viva” perante a estação base, ao

receber os resultados advindos dos nós sensores, e realizar a fase de pós-processamento, antes de direcionar os resultados para módulo *Base Station Manager*, que os entregará ao usuário. Os seus arquivos de trabalho são:

1) BaseProc.exe: programa executável, no formato binário Win32;
2) SiteConfig.xml: configuração da rede, no formato texto XML. Este arquivo é carregado pela aplicação BaseProc no momento da carga, e deverá estar na mesma pasta (diretório) da aplicação. A subseção B.7.5, abaixo, descreve o formato e os parâmetros presentes nesse arquivo;
3) GlobalSchema.xml: catálogo com o esquema global das relações envolvidas com a consulta que está sendo processada. A subseção B.7.6 descreve a estrutura deste catálogo.

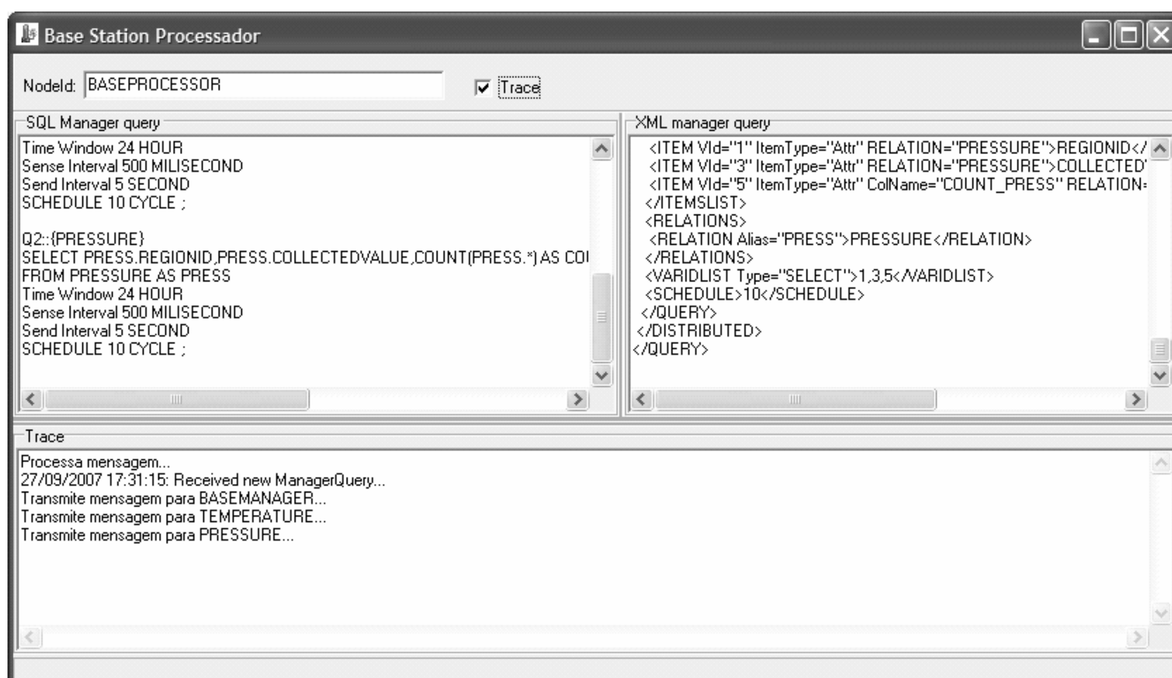


Figura B.8 Tela principal da aplicação *Base Station Processor*.

Ao ser carregado, o BaseProc poderá entrar em modo de procura automática da aplicação agente de rede (NetAgent), que poderá estar em qualquer estação dentro da rede real (utilização do *DiscoveryChannel*, via protocolo UDP). Este procedimento será executado se o parâmetro *<DiscoveryChannel>Auto*=”*yes*”, do

SiteConfig.xml, estiver ativo. Caso contrário, esta aplicação assumirá a configuração do *DataChannel* presente no arquivo SiteConfig.xml.

Depois de configurado ou descoberto o *DataChannel*, a BaseProc se conecta, via TCP, ao agente de rede. A conexão ao *DataChannel* por esta aplicação servirá para a sua comunicação com a aplicação *Base Station Manager* (BaseMan), e também com a aplicação *Sensor Node Processor*, que pode estar representada por uma ou mais instâncias do módulo SensorProc, inclusive sendo executado em mais de uma computador na rede real.

Embora toda a lógica desta aplicação seja executada sem a necessidade de uma interface com usuário, existem os seguintes controles ou campos disponíveis na sua tela principal (Figura B.8), que são:

Caixas de texto:

- [SNQL *subqueries*]: remonta, em SNQL, o resultado da decomposição da consulta passada pela aplicação BaseMan. Isto serve apenas para propósitos de rastreamento;
- [XML *subqueries*]: mostra o *script* das consultas decompostas, passadas pela aplicação BaseMan. Esta apresentação serve somente para propósitos de rastreamento. Este formato é o que será utilizado na disseminação das subconsultas pela RSSF;
- [*Trace*]: caixa de texto com o rastreamento das operações desta aplicação, contendo mensagens de erro e o passo a passo da interpretação das consultas. O rastreamento pode ser ativado ou desativado, através da caixa de checagem [*Trace*];

Estando em execução, esta aplicação fica em compasso de espera de consulta a ser injetada na rede, ou alteração de parâmetros de consulta existente (do módulo BaseMan), e também aguardando resultados contínuos das subconsultas, caso exista consulta ativa, originários dos nós sensores. Logicamente, os nós sensores para quem o BaseProc dissemina consultas são os nós definidos como sendo seus filhos, o que pode ser definido na configuração de rede do arquivo SiteConfig.Xml, conforme apregoa o protocolo de roteamento adotado neste mecanismo.

Após receber os resultados dos nós sensores filhos, é responsabilidade do BaseProc realizar a fase de pós-processamento estabelecida nos fluxos do PDCRS, antes de entregar os resultados para a aplicação BaseMan. Este pós-processamento dependerá das estratégias de junções e agrupamentos adotadas na estação base. Um exemplo deste tipo de processamento é o algoritmo ADAPT, que foi proposto em [8].

B.7.4 Módulo Sensor Nodes Processor (SensorProc)

A aplicação SensorProc é uma aplicação que utiliza uma classe que encapsula as rotinas para processamento de subconsultas nos nós sensores (Ver Apêndice D. Principais classes do PDCRS, Classe TSensorProc), simulando o comportamento desses dispositivos não apenas nos aspectos de coleta de dados, como também implementando estratégias de roteamento para comunicação entre os nós, além de seguir um modelo de custo de utilização de sua energia (Ver Apêndice C. Modelo de consumo de energia pelo sensor). Os seus arquivos de trabalho são:

1) SensorProc.exe: programa executável, no formato binário Win32;
2) SiteConfig.xml: configuração da rede, no formato texto XML. Este arquivo é carregado pela aplicação SensorProc no momento da carga, e deverá estar na mesma pasta (diretório) da aplicação. A subseção B.7.5, abaixo, descreve o formato e os parâmetros presentes nesse arquivo;
3) GlobalSchema.xml: catálogo com o esquema global das relações envolvidas com a consulta que está sendo processada. A subseção B.7.6 descreve a estrutura deste catálogo;
4) SensorList.xml: lista de definições estáticas para criação de novos nós sensores. A Figura B.13, mais adiante, expõe o formato deste arquivo.

Sensors

File Edit Windows

Node Id	Sensors Group	% Avail.Mem	% Avail.Ener...	Act.Query	Status	Send Ajusta...	Sense Ajust...	Se
01	TEMPERATURE	100.0	92.4	Sim	17:32:30: Waiting Query...	5000	500	50
02	TEMPERATURE	100.0	31.7	Sim	17:32:31: Waiting Query...	5000	500	50
03	TEMPERATURE	86.5	55.8	Sim	17:32:31: Received area was proces...	5000	500	50
04	TEMPERATURE	100.0	0.0	Não	17:26:41: *Finished*	86400000	500	50
05	TEMPERATURE	100.0	35.0	Sim	17:32:31: Waiting Query...	5000	500	50
06	TEMPERATURE	100.0	35.7	Sim	17:32:30: Waiting Query...	5000	500	50
07	TEMPERATURE	100.0	0.0	Não	17:30:11: *Finished*	86400000	500	50
08	TEMPERATURE	100.0	37.4	Sim	17:32:30: Waiting Query...	5000	500	50
09	TEMPERATURE	100.0	92.4	Sim	17:32:31: Received area was proces...	5000	500	50
10	TEMPERATURE	100.0	34.0	Sim	17:32:31: Received area was proces...	5000	500	50
11	PRESSURE	100.0	93.6	Sim	17:32:31: Waiting Query...	5000	500	50
12	PRESSURE	100.0	6.2	Sim	17:32:31: Waiting Query...	5000	500	50
13	PRESSURE	46.4	19.6	Sim	17:32:30: Received area was proces...	5000	500	50
14	PRESSURE	100.0	0.0	Não	17:25:22: *Finished*	86400000	500	50
15	PRESSURE	100.0	18.0	Sim	17:32:31: Received area was proces...	5000	500	50
16	PRESSURE	100.0	35.8	Sim	17:32:31: Waiting Query...	5000	500	50
17	PRESSURE	100.0	48.6	Sim	17:32:31: Received area was proces...	5000	500	50
18	PRESSURE	100.0	23.9	Sim	17:32:31: Received area was proces...	5000	500	50
19	PRESSURE	100.0	49.5	Sim	17:32:30: Waiting Query...	5000	500	50
20	PRESSURE	100.0	27.4	Sim	17:32:31: Waiting Query...	5000	500	50
21	TEMPERATURE	100.0	99.9	Sim	17:32:31: Waiting Query...	0	0	0
22	TEMPERATURE	100.0	99.9	Sim	17:32:31: Waiting Query...	0	0	0
23	TEMPERATURE	100.0	99.9	Sim	17:32:31: Waiting Query...	0	0	0
24	TEMPERATURE	100.0	99.9	Sim	17:32:31: Waiting Query...	0	0	0
25	TEMPERATURE	100.0	99.9	Sim	17:32:30: Waiting Query...	0	0	0
26	TEMPERATURE	100.0	99.9	Sim	17:32:30: Waiting Query...	0	0	0

35 sensores

Figura B.9 Tela principal da aplicação Sensor Node Processor.

A Figura B.9 mostra a tela principal da SensorProc, onde estão sendo mostrados nós sensores em execução por esta aplicação. Para se criar novos nós, esta aplicação oferece uma interface com o usuário onde é possível a criação de novas instâncias de categorias de nós sensores, conforme as definições de relações presentes no catálogo do esquema global utilizado. A Figura B.10 mostra o menu da tela inicial, com as opções de criação de novas categorias, conforme as seguintes opções:

(i) [*Create new sensors group*]: criação interativa de nós sensores baseada nas estruturas de relações do esquema global utilizado;

(ii) [*Create sensors group from list*]: criação de nós sensors a partir de lista estática previamente construída, e baseada nas estruturas de relações do esquema global utilizado.

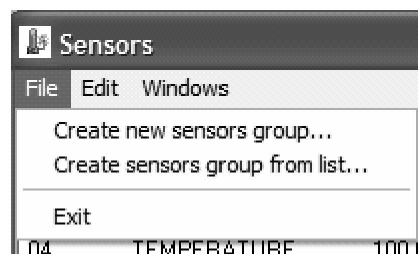


Figura B.10 Opções para criação de nós sensores – SensorProc.

Criação interativa de novos nós sensores

A Figura B.11 apresenta a interface por onde novos nós sensores podem ser criados. Os controles desta tela são descritos a seguir:

Figura B.11 Criação de nós sensors interativamente – SensorProc.

- [Nome do grupo de sensores]: lista contendo os nomes das categorias de sensores identificados no esquema global, de onde pode ser escolhido de qual categoria o novo nó sensor pertencerá;
- [Quantidade de sensores]: quantidade de nós sensores que serão criados;
- [Node id (Numerar a partir de)]: identificação do nó sensor que será criado, ou, no caso de a quantidade de sensores ser maior que 1 (um), neste campo deve ser informado o número de identificação a partir do qual os novos sensores deverão ser numerados. Obs.: a identificação de cada nó sensor na RSSF deve ser única. Por isso, esta aplicação sempre sugere um número imediatamente superior ao último nó sensor criado.
- [ParentList]: lista contendo os nós sensores pais do(s) nó(s) sensor(es) a ser criado. Se este campo for deixado com “*”, os pais dos sensores a

serem criados serão definidos automaticamente pelo agente de rede (aplicação NetAgent);

- [Ligar Trace]: se marcado, o(s) novo(s) nó(s) sensor(es) iniciarão com a opção de rastreamento ligada. Esta opção também poderá ser configurada posteriormente, com o nó sensor em funcionamento.
- [OK]: ao ser clicado neste botão, o(s) novo(s) nó(s) será(ao) criado(s), e inicializados.

Obs.: as coordenadas de cada nó sensor criado por esta opções serão definidas aleatoriamente pelo agente de rede.

Criação de novos nós sensores a partir de lista pré-definida

A Figura B.12 apresenta a interface por onde novos nós sensores podem ser criados a partir de definições estática que foram previamente armazenados em um arquivo (lista de sensores). Os controles desta tela são descritos a seguir:

Figura B.12 Criação de nós sensors a partir de lista pré-definida – SensorProc.

- [Nome da lista de sensores]: lista contendo as definições para a criação de novos sensores. Cada arquivo pode conter definições de sensores para uma única categoria de sensoriamento. Esta lista é um arquivo no formato XML, conforme o exemplo a seguir, representado pela Figura B.13, e descrições a seguir:

- o [*SensorGroup*]: Nome da categoria de sensores para o qual os novos nós serão criados. Esta categoria deverá ter sua estrutura definida no esquema global;
- o [*NodeId*]: identificação do nó sensor que será criado. Esta identificação deverá ser única em toda a rede. Caso já existe na RSSF um nó com esta identificação, o novo sensor não será criado (a aplicação apresentará mensagem de erro);
- o [*NodePosition*]: coordenadas do no sensor dentro do espaço bidimensional da RSSF simulada. Se esta opção for preenchida com "*", esta definição será feita pelo agente de rede, aleatoriamente;
- o [*ParentList*]: lista contendo os nós sensores pais do nó sensor. Se este campo for deixado com "*", os pais do sensor serão definidos automaticamente pelo agente de rede (aplicação NetAgent);

- [Ligar *Trace*]: se marcado, o(s) novo(s) nó(s) sensor(es) iniciarão com a opção de rastreamento ligada. Esta opção também poderá ser configurada posteriormente, com o nó sensor em funcionamento.
- [OK]: ao ser clicado neste botão, o(s) novo(s) nó(s) será(ao) criado(s), e inicializados.

```
<SENSORLIST SensorsGroup="TEMPERATURA">
  <SENSOR NODEID="01" NODEPOSITION="*" PARENTLIST="BASE" />
  <SENSOR NODEID="02" NODEPOSITION="*" PARENTLIST="BASE" />
  <SENSOR NODEID="03" NODEPOSITION="*" PARENTLIST="BASE" />
  <SENSOR NODEID="04" NODEPOSITION="*" PARENTLIST="01;05" />
  <SENSOR NODEID="05" NODEPOSITION="*" PARENTLIST="01;02" />
  <SENSOR NODEID="06" NODEPOSITION="*" PARENTLIST="03" />
  <SENSOR NODEID="07" NODEPOSITION="*" PARENTLIST="03" />
  <SENSOR NODEID="08" NODEPOSITION="*" PARENTLIST="04" />
  <SENSOR NODEID="09" NODEPOSITION="*" PARENTLIST="05" />
  <SENSOR NODEID="10" NODEPOSITION="*" PARENTLIST="06" />
  <SENSOR NODEID="11" NODEPOSITION="*" PARENTLIST="06;07" />
  <SENSOR NODEID="12" NODEPOSITION="*" PARENTLIST="07" />
</SENSORLIST>
```

Figura B.13 Exemplo de arquivo de lista de sensors – SensorProc

O exemplo da Figura B.13 define a criação de 10 (dez) novos sensores para a categoria de sensoriamento "Temperatura". O diagrama apresentado na Figura

B.14 demonstra a direção dos fluxos de disseminação de subconsultas conforme as definições de paternidade dos nós sensores envolvidos no exemplo corrente.

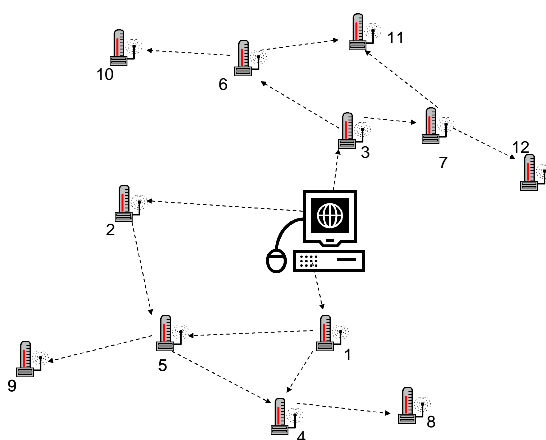


Figura B.14 Diagrama da rede de sensores conforme lista de sensores exemplo.

Operação dos nós sensores

Cada nó sensor criado na aplicação SensorProc corresponde a uma instância da classe TSensorProc, sendo a sua execução realizada como uma *Thread*[14] independente dentro desta aplicação. Cada objeto dessa classe, ao ser instanciado, entra no modo de procura automática da aplicação agente de rede (NetAgent), que poderá estar em qualquer estação dentro da rede real (utilização do *DiscoveryChannel*, via protocolo UDP). Este procedimento será executado se o parâmetro <DiscoveryChannel>Auto=”yes”, do SiteConfig.xml, estiver ativo. Caso contrário, será assumida a configuração do *DataChannel* presente no arquivo SiteConfig.xml.

Depois de configurado ou descoberto o *DataChannel*, cada instância da TSensorProc se conecta, independentemente, via TCP, ao agente de rede. A conexão ao *DataChannel* por esta aplicação servirá para a sua comunicação com os demais nós da rede, inclusive a aplicação *Base Station Processor*.

[14] *Thread*: divisão de um processo em duas ou mais tarefas que podem ser executadas simultaneamente por um sistema operacional.

As informações sobre a operação de cada nó sensor que são mostradas continuamente nas colunas da lista de sensores ativos, na tela principal da aplicação SensorProc (Figura B.9), são descritas a seguir:

- [*NodeId*]: Identificação do nó sensor;
- [*Sensors Group*]: nome da categoria de sensores;
- [% Avail.Mem]: memória (simulada) disponível para este nó sensor;
- [% Avail.Energy]: energia (simulada) disponível para este nó sensor;
- [Act.Query]: indica se a consulta está ativa (sendo executada);
- [Status]: mostra qual atividade o sensor está executando;
- [Send Ajustado]: parâmetro *Send Interval* da consulta, após o ajuste realizado pela fase de monitoração de recursos;
- [Sense Ajustado]: parâmetro *Sense Interval* da consulta, após o ajuste realizado pela fase de monitoração de recursos;
- [Send Original]: parâmetro *Send Interval* que foi originalmente definido pelo usuário na consulta SNQL;
- [Sense Original]: parâmetro *Sense Interval* que foi originalmente definido pelo usuário na consulta SNQL;
- [Parents]: identificação dos nós pais deste nó sensor.

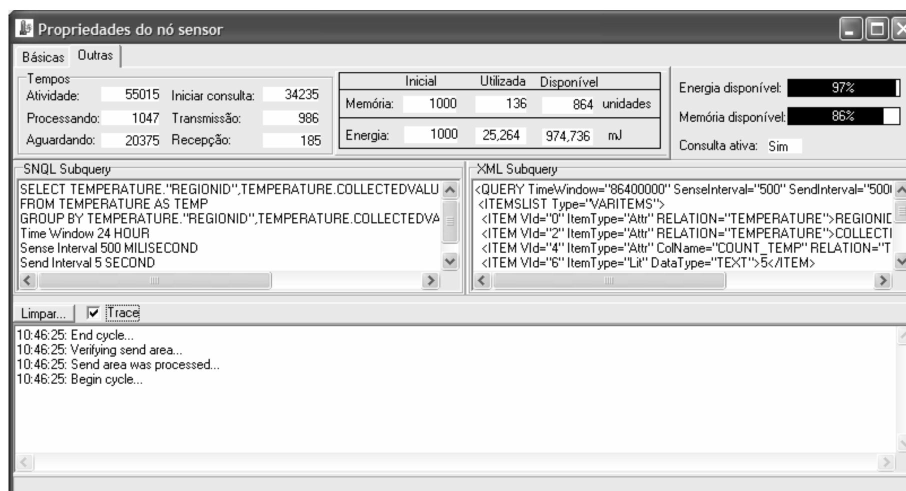


Figura B.15 Tela de propriedades de um nó sensor – SensorProc.

Outras propriedades de um nó sensor, constante desta lista, também podem ser observadas, pela opção “propriedades” (mouse botão direita em item da lista), que apresenta a tela mostrada na Figura B.15.

B.7.5 Parâmetros de configuração

A Figura B.16 mostra um exemplo de conteúdo para o arquivo de configuração SiteConfig.xml. Abaixo é detalhado o significado de cada parâmetro, e em qual aplicação estes são utilizados.

```
<SITECONFIG SENSORSGROUP="*" NODEID="*" GLOBALSCHEMA="*"
            NODEPOSITION="*" PARENTLIST="1;2;3;4">
  <NETWORK Ratio="200" NodesReach="10">
    <DISCOVERYCHANNEL NETADDRESS="127.0.0.1" AGENTPORT="2222"
                      SENSORPORT="3333" AUTO="YES" />
    <DATACHANNEL ADDRESS="127.0.0.1" PORT="1111" />
  </NETWORK>
  <LOGS>
    <EVENTS PATH="Events.log">NO</EVENTS>
    <ERRORS PATH="Errors.log">NO</ERRORS>
  </LOGS>
</SITECONFIG>
```

Figura B.16 Configuração exemplo para SiteConfig.xml.

Parâmetros <SiteConfig> : os parâmetros deste grupo serão utilizados pelas aplicações BaseMan, BaseProc e SensorProc. No caso da aplicação SensorProc, os valores dos parâmetros *SensorsGroup*, *NodeId* e *ParentList* (ver abaixo) poderão ser sobrepostos por valores dados pela aplicação (no caso de geração de mais de um nó sensor pela mesma instância da aplicação), ou mesmo após a conexão com o agente de rede, o qual poderá denominar automaticamente os nós que forem se conectando à rede.

- SensorsGroup: nome da categoria de sensores da qual o nó faz parte. No caso da estação base (aplicações BaseMan e BaseProc), este valor será sempre assumido como sendo igual a “Base”. Para os nós sensores, caso este parâmetro seja igual a “*”, o nome da categoria de sensoriamento deverá ser gerado pela aplicação (através de alguma interface interativa ou arbitrada pela aplicação);
- NodeId: identificação do nó sensor. No caso da estação base (aplicações BaseMan e BaseProc), este valor será sempre assumido

como sendo igual a “BaseMan” e “BaseProc”. Para os nós sensores, caso este parâmetro seja igual a “*”, a identificação do nó sensor deverá ser gerado pela aplicação (através de alguma interface interativa ou arbitrada pela aplicação);

- GlobalSchema: nome do arquivo contendo o esquema global da RSSF (catálogo do mapeamento entre grupos sensores x relações). Caso este valor seja igual a “*”, será assumido o default “GlobalSchema.xml”. Caso o nome aqui configurado não contenha o caminho completo, será assumido o caminho do módulo executável da aplicação;
- NodePosition: coordenadas do nó sensor dentro do espaço bidimensional representando a RSSF. Caso este valor seja igual a “*”, as coordenadas serão geradas aleatoriamente pela aplicação;
- ParentList: lista de nós “pais” do nó corrente. No caso das aplicações BaseProc e BaseMan, este parâmetro não será utilizado, pois não há nós pais para a estação base. Para a aplicação SensorProc, este parâmetro define para quais nós determinados pacotes de mensagens deverão ser enviados, objetivando fazê-los chegar na estação base, conforme a estratégia de roteamento adotada;

Parâmetros <*Network*>: parâmetros utilizados pela aplicação NetAgent para controlar o redirecionamento de pacotes para os nós da rede.

- *Ratio*: número inteiro que define o raio da rede de sensores sem fio (RSSF). Lembrando que, por definição, a rede tem como centro a estação base;
- *NodesReach*: potência de alcance dos nós da rede. Número inteiro que define a distância máxima que cada nó conseguirá “enxergar” os nós vizinhos. Este parâmetro permitirá que o agente de rede simule a capacidade de cada nó sensor para enviar pacotes para os destinos.

Parâmetros <*DiscoveryChannel*>: parâmetros utilizados pelas aplicações para localizar automaticamente (se parâmetro *Auto*=”*yes*”) a estação da rede real onde o agente de rede estiver sendo executado. As aplicações (BaseMan, BaseProc e

SensorProc) emitirão pacote *multicast* UDP, denominado *DiscoveryRequest*, para a rede indicada pelo parâmetro *NetAddress*.

- *NetAddress*: endereço IP da rede por onde o agente de rede (NetAgent) será procurado pelos demais nós;
- *AgentPort*: porta pela qual o agente de rede receberá pacotes UDP de *DiscoveryRequest*, das demais estações que querem se conectar à rede;
- *SensorPort*: porta pela qual os sensores receberão a resposta (*DiscoveryResponse*) do agente de rede, quando este é encontrado;
- *Auto*: este parâmetro, se o valor for "*yes*", define que o nó da rede procurará automaticamente o agente de rede, através da emissão de pacote DiscoveryRequest. Caso contrário (se o valor for "*no*"), o nó tentará se conectar (*DataChannel*) diretamente ao endereço e porta definidos nos parâmetros <*DataChannel*> (conexão manual).

Parâmetros <*DataChannel*>: canal de dados (TCP) por onde as demais estações se comunicarão com o agente de rede (*NetAgent*). Inicialmente estes parâmetros são utilizados pela aplicação NetAgent, para configurar o *socket* que se manterá sempre aguardando novas conexões das aplicações BaseMan, BaseProc e SensorProc. Por outro lado, as aplicações se conectarão ao agente de rede abrindo conexões a partir destes parâmetros (que podem ser descobertos ou arbitrados pela configuração).

- *Address*: endereço IP de rede real onde se encontra a aplicação NetAgent (agente de rede) carregada. Utilizado pelas demais aplicações, caso a descoberta do agente não seja automática, via *DiscoveryChannel*;
- *Port*: porta pela qual o agente de rede se manterá escutando novas conexões provenientes das demais aplicações. As demais aplicações poderão descobrir esta porta automaticamente, pelo *DiscoveryChannel*, se a configuração for automática.

Parâmetros <*Logs*>: define gravação de arquivos de logs de erros e eventos diversos. Utilizados por todas as aplicações.

- *<Events>*: “*Yes*” ou “*No*”: liga ou desliga a gravação em arquivos dos *logs* de eventos;
- *<Events> Path*: caminho e nome do arquivo de *log* de eventos;
- *<Errors>*: “*Yes*” ou “*No*”: liga ou desliga a gravação em arquivos dos *logs* de erros;
- <Errors> Path: caminho e nome do arquivo de log de erros.

B.7.6 Catálogo do esquema global

A Figura B.17, abaixo, mostra a estrutura de um arquivo de catálogo das relações, também denominado arquivo de esquema global, onde são definidos os mapeamentos dos resultados gerados pelos nós sensores para relações, conforme suas categorias de sensoriamento, e a definição das estruturas de tabelas reais utilizadas na estação base. Ver também a Figura 5.2, no Capítulo 5 desta dissertação, onde é mostrado um exemplo completo de um esquema global gerado. As aplicações representando a estação base (BaseMan e BaseProc) e também a aplicação que simula os nós sensores utilizam este arquivo. Abaixo são descritos os elementos deste arquivo.

```
<SCHEMA>
  Nome_do_esquema
  <RELATIONS>
    <SENSORS>
      NomeCategoriaSensor_1
      <ATTRIB Type="TipoDoAtributo_1">NomeDoAtributo_1</ATTRIB>
        . . .
      <ATTRIB Type="TipoDoAtributo_n">NomeDoAtributo_n</ATTRIB>
    </ SENSORS >
  </RELATIONS>
</SCHEMA>
```

Figura B.17 Estrutura do catálogo do esquema global

<Schema> Nome_do_esquema: nome do esquema gerado. Identificação gerada na criação deste arquivo.

<Relations>: lista de elementos contendo estruturas das relações existentes, que podem representar o mapeamento relacional das categorias de sensores, ou outras estruturas de relações que futuramente podem ser suportadas pelo PDCRS.

<*Sensors*> NomeCategoriaSensor: nome da categoria de sensoriamento do dispositivo simulado, que será utilizado como identificação da relação quando do processamento das consultas.

<*Attrib*> *Type*: tipo de dado relativo ao atributo da relação;

<*Attrib*> NomeDoAtributo: nome da coluna da relação, que será utilizado no processamento das consultas;

<*Table*> NomeTabelaBase: nome da tabela de dados existente na estação base, que é referenciada em consultas.

Apêndice C. Modelo de consumo de energia pelo sensor

O modelo simplificado de custo de energia para nós sensores apresentado a seguir foi utilizado pelo simulador apresentado no Apêndice B deste trabalho.

Tabela C.1 Parâmetros de consultas.

# Item	Parâmetros de consultas	Exemplos
1	TimeWindow	24 horas
2	SendInterval	5 minutos
3	SenseInterval	60 segundos
4	nChildren	2 (valor médio. É variável durante a operação)
5	nParents	1

A Tabela C.1 mostra exemplos de parâmetros de consultas, conforme descrição abaixo:

- 1: Tempo de vida da consulta;
- 2: Intervalo entre transmissões;
- 3: Intervalo entre coletas;
- 4: Quantidade de sensores filhos de um sensor que não é folha;
- 5: Número de sensores pais de qualquer sensor.

Tabela C.2 Parâmetros dependentes da plataforma de sensor.

# Item	Parâmetros da plataforma	Exemplos, conforme [42]
6	tAcq	0,35 milissegundos
7	sOverPacket	20 bytes
8	sTuple	20 bytes em média
9	mWAcq	13 miliWatt
10	mWSleep	0,0006 miliWatt
11	nBytesRef	100 bytes
12	mWSendBytes	200 miliWatt
13	tSendBytes	30,0 milissegundos
14	mWRecBytes	40 miliWatt
15	mWproc	12 miliWatt
16	fAggr	0,5

A Tabela C.2 mostra os parâmetros cujos valores dependem da arquitetura da plataforma de nó sensor utilizado e de características do modelo de dados aplicado, conforme descrição abaixo:

- 6: Tempo gasto para adquirir uma tupla;
- 7: Tamanho do cabeçalho e controles do pacote;

- 8: Tamanho médio de uma tupla;
- 9: Potência necessária por aquisição, incluído energia para processamento;
- 10: Potência necessária para manter processador "dormindo" (aguardando próximo intervalo de atividade de coleta de dados ou comunicação);
- 11: Número de bytes para referência nos cálculos;
- 12: Potência necessária para enviar a quantidade de bytes de referência;
- 13: Tempo necessário para enviar a quantidade de bytes de referência (em média para 49 Kb/s);
- 14: Potência necessária para receber a quantidade de bytes de referência;
- 15: Potência de consumo do processador;
- 16: Fator de agregação para os pacotes gerados. 0 > fAggr >= 1. Este valor dependerá da amostra coletada pelos sensores, conforme os filtros da consulta. Para menor agregação mínima (pior), este fator tenderá a 1. Para maior agregação (melhor), este fator tenderá a 0.

Tabela C.3 Parâmetros derivados calculados.

# Item	Itens derivados	Fórmulas
17	mJSendBytes	mWSendBytes x tSendBytes
18	mJRecBytes	mWRecBytes x tSendBytes
19	nPacksGenTW	TimeWindow / SendInterval
20	nAcqSI	SendInterval / SenseInterval
21	nTuplePerPack	nAcqSI x fAggr
22	tSendTuple	sTuple x tSendBytes / nBytesRef
23	mJAcq	mWAcq x tacq
24	mJSendOver	sOverPacket x mJSendBytes / nBytesRef
25	mJSendTuple	sTuple x mJSendBytes / nBytesRef
26	mJSendPack	mJSendOver + (nTuplePerPack x mJSendTuple)
27	mJRecOver	sOverPacket x mJRecBytes / nBytesRef
28	mJRecTuple	sTuple x mJRecBytes / nBytesRef
29	mJRecPacket	mjRecOver + (nTuplePerPack x mjRecTuple)
30	mJSleepSec	mWSleep x 1 segundo
31	mJProcTW	mWSleep x TimeWindow
32	mJGenTuplesSI	mJAcq x nAcqSI

33	tGenPack	nAcqSI x tAcq
34	tSendPack	nTuplePerPack x tSendTuple
35	tProcPack	tGenPack + tSendPack
36	sMaxPack	nAcqSI x sTuple + sOverPacket
37	sSizePackAggr	nTuplePerPack x sTuple + sOverPacket
38	tGenPacksTW	nPacksGenTW x tGenPack
39	tSendPacksTW	nPacksGenTW x tSendPack x nParents
40	tProcPacks	tSendPacksTW + tGenPacksTW
41	tSleepTW	TimeWindow – tProcPacks
42	nAcqTW	nPacksGenTW x nAcqSI
43	mJGenPacksTW	mJGenTuplesSI x nPacksGenTW
44	mJSleepTW	tSleepTW x mJSleepSec
45	mJRecPacksTW	nPacksGenTW x mJRecPacket x nChildren
46	mJSendPacksTW	(nPacksGenTW x mJSendPack x nParents) + (nPacksGenTW x nChildren x fAggr)
47	**mJProcSendRecTW**	**mJSleepTW + mJGenPacksTW + mJSendPacksTW + mJRecPacksTW**

A Tabela C.3 mostra os parâmetros derivados que são calculados com base nos parâmetros da consulta (Tabela C.1) e nos parâmetros dependentes da plataforma e aplicação (Tabela C.2), conforme descrição abaixo:

- 17: Energia necessária para enviar a quantidade de bytes de referência;
- 18: Energia necessária para receber a quantidade de bytes de referência;
- 19: Número máximo de pacotes gerados durante o tempo de vida da consulta;
- 20: Número máximo de aquisições durante o intervalo para transmissão: número máximo de tuplas que um pacote terá;
- 21: Número de tuplas acumulada por pacote, considerando fator de agregação;
- 22: Tempo para enviar uma tupla;
- 23: Energia gasta por aquisição;
- 24: Energia gasta para enviar o cabeçalho de cada pacote;
- 25: Energia gasta para enviar uma tupla;
- 26: Energia gasta para enviar um pacote;
- 27: Energia gasta para receber o cabeçalho de um pacote;

- 28: Energia gasta para receber uma tupla;
- 29: Energia gasta para receber um pacote;
- 30: Energia gasta para o processador dormindo por 1 segundo;
- 31: Energia gasta para o processador funcionar pelo período da consulta (*TimeWindow*), enquanto estiver "dormindo" (aguardando próximo intervalo);
- 32: Energia gasta para gerar tuplas a cada *send interval*;
- 33: Tempo usado na geração de um pacote máximo;
- 34: Tempo para enviar um pacote;
- 35: Tempo para gerar e enviar um pacote;
- 36: Tamanho máximo de um pacote, sem considerar fator de agregação;
- 37: Tamanho de um pacote com agregação;
- 38: Tempo gasto para gerar todos os pacotes durante a vida da consulta;
- 39: Tempo gasto para (somente) enviar todos os pacotes durante a vida da consulta;
- 40: Tempo para compor e enviar todos os pacotes durante a vida da consulta;
- 41: Tempo diário que o processador passa "dormindo";
- 42: Número de aquisições (tuplas) no *TimeWindow*;
- 43: Energia gasta para gerar todos os pacotes durante o tempo de vida da consulta;
- 44: Energia gasta durante o tempo da consulta, relativo aos intervalos em que o processador ficar "dormindo";
- 45: Energia (máxima) gasta para receber os pacotes dos filhos (nem todos os pacotes a serem enviados pelos filhos irão para todos os pais);

- 46: Energia gasta para enviar pacotes durante o tempo de vida da consulta;
- 47: Energia gasta para o nó sensor funcionar e enviar pacotes durante o tempo da consulta.

Tabela C.4 Unidades de medida.

Unidades de medidas	**Descrição das fórmulas**
Potência = W (Watt)	Potência = Corrente x Voltagem
Corrente = A (Ampére)	Corrente = Potência / Voltagem
Energia = J (Joule)	Energia = Potência x Tempo
Voltagem = V (Volt)	Voltagem = Corrente / Potência
Tempo = s (Segundo)	Potência = Energia / Tempo

A Tabela C.4 mostra as unidades de medida relacionadas com o modelo de consumo de energia aqui apresentado, além das fórmulas que relacionam uma unidade a outra.

Apêndice D. Principais classes do PDCRS

A seguir serão descritas as interfaces das principais classes do PDCRS, com os seus métodos e atributos relevantes, conforme o diagrama na Figura D.1.

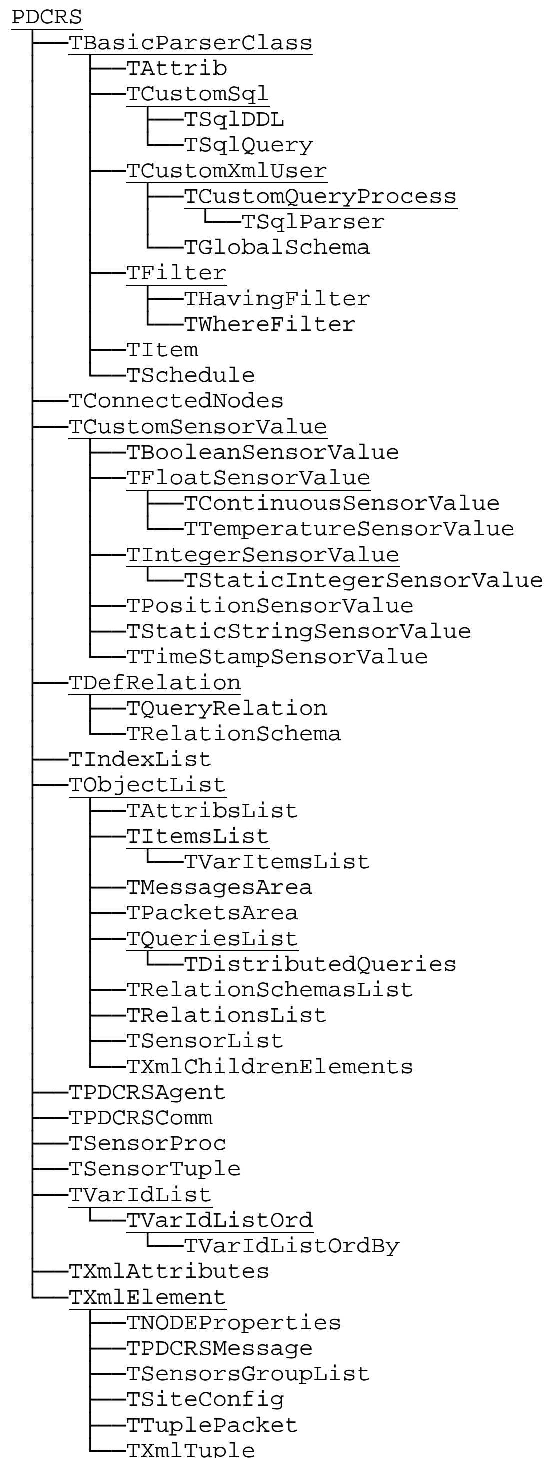


Figura D.1 Diagrama de classes do PDCRS.

```
// Classe TBasicParserClass: funções ancestrais de interpretação de cadeias de comandos e expressões
// Aplicações usuárias: BaseStationManager.
TBasicParserClass = class(TPDCRSControl)
  public
    constructor Create(AOwner:TComponent); override;
    destructor Destroy; override;
    procedure ClearTrace;
    function ErrorMessage: string; overload;
    function ErrorMessage(v_CodigoErro:Integer): string; overload;
    procedure InitializeParser;
  end;

// Classe TSqlParser: rotinas principais do interpretador de comandos SNQL e execução de consultas
// Aplicações usuárias: BaseStationManager.
TSqlParser = class(TCustomQueryProcess)
  public
    constructor Create(AOwner:TComponent); override;
    destructor Destroy; override;
    function ExecuteBatchScriptFromFile(v_ScriptFileName:String): Boolean;
    function ExecuteBatchScriptFromString(v_ScriptString:String): Boolean;
    function ExecuteParser: Boolean;
    function ExecuteParserFromFile(v_FileName:String): Boolean;
    function ExecuteParserFromString(v_SqlString:String): Boolean;
    procedure Initialize;
    function LoadFromXmlElement(v_XmlElement:TXmlElement): Boolean; override;
    property CheckSchemaSemantic: Boolean read v_CheckSchemaSemantic write
              v_CheckSchemaSemantic;
    property CustomSql: TCustomSql read f_CustomSql;
    property ErrorColumnNumber: Integer read v_Pt_Lin;
    property ErrorLineNumber: Integer read v_Num_Lin;
    property GlobalSchemaName: string read v_GlobalSchemaName write  p_PutGlobalSchemaName;
    property InputSqlString: TStringList read v_SqlStringEntrada write v_SqlStringEntrada;
    property OwnSqlQuery: Boolean read v_OwnSqlQuery write v_OwnSqlQuery;
    property SqlDDL: TSqlDDL read v_SqlDDL write v_SqlDDL;
    property SqlQuery: TSqlQuery read v_SqlQuery write v_SqlQuery;
    property UpgradeSchema: Boolean read v_UpgradeSchema write v_UpgradeSchema;
  end;

// Classe TGlobalSchema: guarda e controla o catálogo geral das relações dos grupos sensores
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.
TGlobalSchema = class(TCustomXmlUser)
  public
    constructor Create; reintroduce;
    constructor CreateCopy(v_GlobalSchema:TGlobalSchema);
    destructor Destroy; override;
    function DeleteSchemaFile: Boolean;
    procedure Initialize;
    function LoadFromXmlElement(v_XmlElement:TXmlElement): Boolean; override;
    function SaveSchemaFile: Boolean;
    property CreateXmlElement: TXmlElement read f_CreateXmlElement;
    property RelationSchemas: TRelationSchemasList read v_RelationSchemas;
    property SchemaId: string read v_SchemaId write v_SchemaId;
  end;

// Classe TAttrib: controla estrutura associada a um atributo de uma relação
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.
TAttrib = class(TBasicParserClass)
```

```
  public
    function CreateCopy: TAttrib;
    property CreateXmlElement: TXmlElement read f_CreateXmlElement;
    property DataType: T_Domain read v_DataType;
    property Name: string read v_Name write v_Name;
  end;
```

// Classe TAttribsList: lista de atributos de uma relação
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.

```
TAttribsList = class(TObjectList)
  public
    function Add(v_Attrib:TAttrib): Boolean; reintroduce; virtual;
    function AttribByName(Name:String): TAttrib;
    function CreateCopy: TAttribsList;
    property CreateXmlAttribs: TXmlChildrenElements read f_CreateXmlAttribs;
    property Items[Index: Integer]: TAttrib read f_GetItems write p_SetItems;   default;
  end;
```

// Classe abstrata TCustomSql: funções ancestrais para intepretação e gerenciamento de comandos SNQL
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.

```
TCustomSql = class(TBasicParserClass)
  protected
    procedure CopyGlobalSchemaFrom(var v_GlobalSchema:TGlobalSchema);
    procedure SaveToFile(v_String:String;v_FileName:String);
    procedure TransferGlobalSchemaFrom(var v_GlobalSchema:TGlobalSchema);
    procedure TransferGlobalSchemaTo(var v_GlobalSchema:TGlobalSchema);
    function VerifyRelationOnGlobalSchema(v_Relation:String): Integer; overload;
    function VerifyRelationOnGlobalSchema(v_RelationName:String;var
                v_RelationSchema:TRelationSchema): Integer; overload;
    function VerifyRelationOnGlobalSchema(v_Relation:String; v_RelationType:T_Token): Integer; overload;
    function VerifyRelationOnGlobalSchema(v_Relation:TDefRelation): Integer;  overload;
    function VerifyRelationOnGlobalSchema(v_Relation:TQueryRelation): Integer;   overload;
   public
    constructor Create; reintroduce;
    constructor CreateCopy(v_CustomSql:TCustomSql);
    destructor Destroy; override;
    function Execute: Boolean; virtual; abstract;
    procedure Initialize;
    procedure SqlSaveToFile(v_FileName:String);
    procedure XmlSaveToFile(v_FileName:String);
    property CreateXmlElement: TXmlElement read f_CreateXmlElement;
    property GlobalSchema: TglobalSchema read v_GlobalSchema write v_GlobalSchema;
    property Ignored: Boolean read v_Ignorado;
    property IndentXml: Boolean read v_IndentXml write v_IndentXml;
    property LogMsg: string read v_LogMsg write v_LogMsg;
    property OwnGlobalSchema: Boolean read v_OwnGlobalSchema write v_OwnGlobalSchema;
    property SqlText: string read f_GetSqlText;
    property XmlElement: TXmlElement write p_PutXmlElement;
    property XmlText: string read f_GetXmlText write p_PutXmlText;
  end;
```

// Classe TSqlDDL: funções para intepretação e gerenciamento de comandos DDL (SNQL)
// Aplicações usuárias: BaseStationManager.

```
TSqlDDL = class(TCustomSql)
  public
    constructor Create;
    destructor Destroy; override;
    function Execute: Boolean; override;
```

```
    property RelationSchema: TRelationSchema read v_RelationSchema;
  end;
```

// Classe TSqlQuery: funções para intepretação e gerenciamento de comandos DML (consultas SNQL)
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.

```
TSqlQuery = class(TCustomSql)
  public
    constructor Create;
    constructor CreateCopy(v_SqlQuery:TSqlQuery;v_Completa:Boolean);
    destructor Destroy; override;
    function AssignTuple(RelationId:String;CurrentTuple,
      RelationHeader:T_ArrayVariant): Boolean;
    function EvaluateFilter(FilterContent:String): Boolean;
    function Execute: Boolean; override;
    function PrepareDistribution: Boolean; virtual;
    function VerifyQueryRelationsOnGlobalSchema(var v_Info:String): Integer;
    property DistributedQueries: TDistributedQueries read v_DistributedQueries;
    property FilterHaving: THavingFilter read v_FilterHaving;
    property FilterWhere: TWhereFilter read v_FilterWhere;
    property GroupByItemIndexes: TVarIdList read v_GroupByItemIndexes;
    property OrderByItemIndexes: TVarIdListOrdBy read v_OrderByItemIndexes;
    property ProjectItemIndexes: TVarIdListOrd read v_ProjectItemIndexes;
    property QueryId: Integer read v_QueryId;
    property Relations: TRelationsList read v_Relations;
    property Schedule: TSchedule read v_Schedule;
    property SendInterval: Int64 read v_SendInterval;
    property SenseInterval: Int64 read v_SenseInterval;
    property SubqueryRelation: TQueryRelation read f_GetSubqueryRelation;
    property TimeWindow: Int64 read v_TimeWindow;
    property VarItems: TVarItemsList read v_VarItems write p_SetVarItems;
  end;
```

// Classe abstrata TCustomXmlUser: base para classes do interpretador
// Aplicações usuárias: BaseStationManager.

```
TCustomXmlUser = class(TBasicParserClass)
  protected
    constructor CreateCopy(v_CustomXmlUser:TCustomXmlUser);
    function LoadFromXmlFile(v_FileName:String): Boolean; virtual;
    function LoadFromXmlString(v_XmlString:String): Boolean; virtual;
    procedure p_PutRelation(v_XmlRelation:TXmlElement; v_RelationSchema:TRelationSchema);
    property LogMsg: string read v_LogMsg write v_LogMsg;
  end;
```

// Classe base TCustomQueryProcess: gerencia processanento de consultas pelo interpretador
// Aplicações usuárias: BaseStationManager.

```
TCustomQueryProcess = class(TCustomXmlUser)
  protected
    FOnAvailableDataResult: TQueryDataResultEvent;
    FOnBeginSchedule: TScheduleEvent;
    FOnBeginScheduleOccurrence: TScheduleOccurrenceEvent;
    FOnCommDisconnect: TNotifyEvent;
    FOnEndSchedule: TScheduleEvent;
    FOnEndScheduleOccurrence: TScheduleOccurrenceEvent;
  public
    constructor Create(AOwner:TComponent); override;
    destructor Destroy; override;
    function ActivateComm: Boolean;
    function InjectQuery(v_SqlQuery:TSqlQuery): Boolean;
```

```
    function StopQuery: Boolean;
    property AdditionalInfo: string read v_AdditionalInfo;
    property CommActive: Boolean read f_GetCommActive write p_SetCommActive;
    property PDCRSComm: TPDCRSComm read v_PDCRSComm;
    property EndOfData: Boolean read f_FimDeDados;
    property ErrorCode: Integer read v_Erro;
    property Trace: Boolean read v_Trace write v_Trace;
    property TraceMessages: string read v_TraceMessages;
  published
    property OnAvailableDataResult: TQueryDataResultEvent read  FOnAvailableDataResult write
              p_SetOnAvailableDataResult;
    property OnBeginSchedule: TScheduleEvent read FOnBeginSchedule write  FOnBeginSchedule;
    property OnBeginScheduleOccurrence: TScheduleOccurrenceEvent read FOnBeginScheduleOccurrence
              write FOnBeginScheduleOccurrence;
    property OnCommDisconnect: TNotifyEvent read FOnCommDisconnect write FOnCommDisconnect;
    property OnEndSchedule: TScheduleEvent read FOnEndSchedule write  FOnEndSchedule;
    property OnEndScheduleOccurrence: TScheduleOccurrenceEvent read  FOnEndScheduleOccurrence write
              FOnEndScheduleOccurrence;
  end;
```

// Classe base TFilter: guarda de filtros de consultas (Having e Where)
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.

```
TFilter = class(TBasicParserClass)
  protected
    v_FilterContent: string;
    v_FilterName: string;
    function f_CreateXmlElement: TXmlElement;
  public
    constructor Create(v_FilterName:String); reintroduce;
    function CreateCopy: TFilter;
    procedure p_PutXmlElement(v_XmlElement:TXmlElement);
    property CreateXmlElement: TXmlElement read f_CreateXmlElement;
    property FilterContent: string read v_FilterContent write v_FilterContent;
    property FilterName: string read v_FilterName;
  end;
```

// Classe THavingFilter: controle de filtros da cláusula de agrupamento nas consultas (Having)
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.

```
THavingFilter = class(TFilter)
  public
    constructor Create;
    function CreateCopy: THavingFilter;
  end;
```

// Classe TWhereFilter: controle de filtros de cláusula de seleção nas consultas (Where)
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.

```
TWhereFilter = class(TFilter)
  public
    constructor Create;
    function CreateCopy: TWhereFilter;
  end;
```

// Classe TItem: estrutura de um identificador (variável ou atributo)
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.

```
TItem = class(TBasicParserClass)
  public
    function CreateCopy: TItem;
    procedure p_PutXmlElement(v_XmlElement:TXmlElement);
```

```
    property ColumnName: string read v_ColumnName;
    property CreateXmlElement: TXmlElement read f_CreateXmlElement;
    property DataType: t_Domain read v_DataType write v_DataType;
    property FunctAggreg: T_Token read v_FunctAggreg;
    property ItemType: t_ItemType read v_ItemType;
    property Ref: Boolean read v_Ref write v_Ref;
    property Relation: TQueryRelation read v_Relation write v_Relation;
    property RelationId: string read v_RelationId;
    property ValueAttrib: Variant read v_ValueAttrib write v_ValueAttrib;
    property ValueName: string read v_ValueName;
    property VarId: string read v_VarId write v_VarId;
  end;

// Classe base TItemsList: lista de identificadores
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.
TItemsList = class(TObjectList)
  public
    function Add(v_Item:TItem): Integer; reintroduce;
    function AddIfNew(var v_Item:TItem): Integer;
    function CreateCopy: TItemsList;
    function Exists(v_Item:TItem): Integer;
    function IndexOf(v_Item:TItem): Integer;
    procedure p_PutXmlElement(v_XmlElement:TXmlElement);
    property CreateXmlElement: TXmlElement read f_CreateXmlElement;
    property Items[Index: Integer]: TItem read f_GetItems write p_SetItems;    default;
    property ItemsByName[RelatId,Name:String]: TItem read f_GetItemsByName     write p_SetItemsByName;
    property VarByName[VarId:String]: TItem read f_GetVarByName write    p_SetVarByName;
  end;

// Classe TVarItemsList: lista de identificadores de uma consulta (parâmetros de uma consulta)
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.
TVarItemsList = class(TItemsList)
  public
    function CreateCopy: TVarItemsList;
    procedure p_PutXmlElement(v_XmlElement:TXmlElement);
    property CreateXmlElement: TXmlElement read f_CreateXmlElement;
  end;

// Classe TSchedule: controle dos parâmetros de agendamento de consultas
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.
TSchedule = class(TBasicParserClass)
  public
    function CreateCopy: TSchedule;
    procedure Initialize;
    procedure p_PutXmlElement(v_XmlElement:TXmlElement);
    property After: Boolean read v_After write v_After;
    property CreateXmlElement: TXmlElement read f_CreateXmlElement;
    property DayPeriod: Word read v_DayPeriod write v_DayPeriod;
    property Hour: TDateTime read v_Hour write v_Hour;
    property Occurrences: Cardinal read v_Occurrences write v_Occurrences;
    property Periodicity: t_Token read v_Periodicity write v_Periodicity;
    property Start: Int64 read v_Start write v_Start;
    property Wait: Int64 read v_Wait write v_Wait;
  end;

// Classe base TDefRelation: definição básica de uma relação
// Aplicações usuárias: BaseStationManager.
TDefRelation = class
```

```
    v_Name: string;
  public
    constructor CreateCopy(v_DefRelation:TDefRelation);
    property Name: string read v_Name write v_Name;
  end;
```

// Classe TQueryRelation: gerencia parâmetros de uma relação em uma consulta
// Aplicações usuárias: BaseStationManager.

```
TQueryRelation = class(TDefRelation)
  public
    constructor Create;
    constructor CreateCopy(v_Relation:TQueryRelation);
    destructor Destroy; override;
    procedure p_PutXmlElement(v_XmlElement:TXmlElement);
    property Alias: string read v_Alias write v_Alias;
    property CreateXmlElement: TXmlElement read f_CreateXmlElement;
    property RelationSchema: TRelationSchema read v_RelationSchema write
        v_RelationSchema;
    property SubqueryIndex: Integer read v_SubqueryIndex;
  end;
```

// Classe TRelationsList: lista de relações envolvidas numa consulta
// Aplicações usuárias: BaseStationManager.

```
 TRelationsList = class(TObjectList)
  public
    function Add(v_Relation:TQueryRelation): Integer; reintroduce;
    function CreateCopy: TRelationsList;
    procedure p_PutXmlElement(v_XmlElement:TXmlElement);
    property CreateXmlElement: TXmlElement read f_CreateXmlElement;
    property Items[Index: Integer]: TQueryRelation read f_GetItems write  p_SetItems; default;
  end;
```

// Classe TRelationSchema: gerencia estrutura de uma relação dentro do esquema global (catálogo)
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.

```
TRelationSchema = class(TDefRelation)
  public
    constructor Create;
    constructor CreateCopy(v_RelationSchema:TRelationSchema);
    destructor Destroy; override;
    function GetAttribDataType(v_AttribName:String): T_Domain;
    procedure Initialize;
    property AttribList: TAttribsList read v_AttribList;
    property CreateXmlElement: TXmlElement read f_CreateXmlElement;
    property Name: string read v_Name write v_Name;
    property RelationType: T_Token read v_RelationType write v_RelationType;
  end;
```

// Classe TRelationsSchemasList: gerencia lista de esquemas de relações de um catálogo
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.

```
 TRelationSchemasList = class(TObjectList)
  public
    function Add(RelationSchema:TRelationSchema): Integer; virtual;
    function CreateCopy: TRelationSchemasList;
    function Exists(v_RelationName:String): Boolean;
    function IndexOf(v_RelationName:String): Integer; overload;
    function RelationSchemaByName(v_RelationName:String): TRelationSchema;
    property CreateXmlElement: TXmlElement read f_CreateXmlElement;
    property Items[Index:Integer]: TRelationSchema read f_Items; default;
```

```
  end;

// Classe base TIndexList: estruturas para listas de identificadores
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.
 TIndexList = class(TObject)
  public
    constructor Create(v_ListName:String); reintroduce;
    function Add(v_Index:Integer): Integer; virtual;
    procedure Clear;
    function Count: Integer;
    function CreateCopy: TIndexList; virtual;
    procedure Delete(Index:Integer); virtual;
    procedure p_PutXmlElement(v_XmlElement:TXmlElement);
    property CreateXmlElement: TXmlElement read f_CreateXmlElement;
    property Items[Index: Integer]: Integer read f_GetItems write p_SetItems; default;
  end;

// Classe base TQueriesList: tratamentos para listas de consultas
// Aplicações usuárias: BaseStationManager, BaseStationProc.
TQueriesList = class(TObjectList)
  protected
    function f_CreateXmlElement: TXmlElement;
    function f_GetItems(Index: Integer): TSqlQuery;
    function f_GetSql: string;
    procedure p_PutXmlElement(v_XmlElement:TXmlElement);
    procedure p_SetItems(Index: Integer; v_Query: TSqlQuery);
  public
    function Add(v_Query:TSqlQuery): Integer; reintroduce; virtual;
    property Items[Index: Integer]: TSqlQuery read f_GetItems write p_SetItems;  default;
  end;

// Classe TDistributedQueries: tratamentos para listas de subconsultas (derivadas)
// Aplicações usuárias: BaseStationManager, BaseStationProc.
 TDistributedQueries = class(TQueriesList)
  protected
    function f_CreateXmlElement: TXmlElement;
    function f_GetSql: string;
    procedure p_PutXmlElement(v_XmlElement:TXmlElement);
  end;

// Classe TVarIdList: lista de identificadores de variáveis
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.
TVarIdList = class(TObject)
  public
    constructor Create(v_ListName:String;v_XmlShortList:Boolean); reintroduce;
    function Add(v_VarId:String): Integer; virtual;
    procedure Clear;
    function Count: Integer;
    function CreateCopy: TVarIdList; virtual;
    procedure Delete(Index:Integer); virtual;
    procedure p_PutXmlElement(v_XmlElement:TXmlElement);
    property CreateXmlElement: TXmlElement read f_CreateXmlElement;
    property Items[Index: Integer]: string read f_GetItems write p_SetItems;      default;
    property XmlShortList: Boolean read v_XmlShortList write v_XmlShortList;
  end;

// Classe TVarIdListOrd: lista de classificação de identificadores
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.
```

```
TVarIdListOrd = class(TVarIdList)
public
  constructor Create(v_ListName:String;v_XmlShortList:Boolean); reintroduce;
  function Add(v_VarId:String;v_OrdGrp:Integer): Integer; reintroduce;     virtual;
  procedure Clear;
  function CreateCopy: TVarIdListOrd; reintroduce; virtual;
  procedure Delete(Index:Integer); reintroduce; virtual;
  procedure p_PutXmlElement(v_XmlElement:TXmlElement);
  property CreateXmlElement: TXmlElement read f_CreateXmlElement;
  property OrderGroup[Index: Integer]: Integer read f_GetOrdGrp write  p_SetOrdGrp;
end;
```

// Classe TPDCRSComm: Gerencia rotinas de comunicação entre os módulos do PDCRS
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.

```
TPDCRSComm = class(TPDCRSControl)
public
  constructor Create(Owner:TComponent); override;
  destructor Destroy; override;
  procedure AbortCurrentOperation;
  function CloseSession: Boolean;
  function CodeErrorToString(v_CodeError:Integer): string;
  function DoTransaction(v_Request:TPDCRSMessage;v_Response:TPDCRSMessage):
    Boolean;
  function HasAvailableData: Boolean;
  function OpenRequest(v_ConnectionApp:T_ConnectionApp): Boolean; overload;
  function OpenRequest(v_ConnectionApp:T_ConnectionApp;
    v_SensorsGroup:String): Boolean; overload;
  procedure ProcessMessages;
  function ReceiveMessage(v_Message:TPDCRSMessage): Boolean;
  function SendMessage(v_Message:TPDCRSMessage): Boolean;
  property Active: Boolean read v_Active;
  property BaseAddress: string read f_Servidor write p_Servidor;
  property BaseTcpPort: string read f_PortaTcp write p_PortaTcp;
  property CreateErrorLog: Boolean read f_CreateErrorLog write
    p_CreateErrorLog;
  property CreateEventLog: Boolean read f_CreateEventLog write
    p_CreateEventLog;
  property ErrorLogFile: string read v_ErrorLogFile write v_ErrorLogFile;
  property EventLogFile: string read v_EventLogFile write v_EventLogFile;
  property LastAuxiliarErrorCode: Integer read v_CodErroAux;
  property LastMessageCriticalError: string read v_UltMsgErroCrit;
  property LastMessageProtocolError: string read f_UltimaMsgErroTcp;
  property LastMessageTransactionError: string read f_UltimaMsgErroTcp;
  property LastTransactionErrorCode: Integer read f_UltimoCodErroTransacao;
  property SiteConfig: TSiteConfig read f_GetSiteConfig write p_SetSiteConfig;
published
  property GlobalSchema: TObject read v_GlobalSchema;
  property LoadSiteConfig: Boolean read v_LoadSiteConfig write
    p_SetLoadSiteConfig;
  property Visible;
  property OnAvailableData: TNotifyEvent read FOnReceivedData write
    p_SetOnAvailableData;
  property OnDisconnect: TNotifyEvent read FOnDisconnect write
    p_SetOnDisconnect;
  property OnLog: TProcLog read FOnLog write FOnLog;
end;
```

// Classe TSiteConfig: controla a configuração de um nó na rede de sensores sem fio

```
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.
TSiteConfig = class(TXmlElement)
  public
    constructor Create;
    constructor CreateName(v_FileName:String);
    destructor Destroy; override;
    procedure Initialize;
    procedure LoadConfig;
    procedure SaveConfig;
    property ArqErros: string read f_GetArqErros write p_SetArqErros;
    property ArqEventos: string read f_GetArqEventos write p_SetArqEventos;
    property AutoDiscoveryAgent: Boolean read f_GetAutoDiscovery write p_SetAutoDiscovery;
    property ConfigFileName: string read v_ConfigFileName write v_ConfigFileName;
    property DataChannelAddr: string read f_GetDataChannelAddr write p_SetDataChannelAddr;
    property DataChannelPort: string read f_GetDataChannelPort write p_SetDataChannelPort;
    property DiscoverChannelAgentPort: string read  f_GetDiscoverChannelAgentPort write
            p_SetDiscoverChannelAgentPort;
    property DiscoverChannelNetAddr: string read f_GetDiscoverChannelAddr write
            p_SetDiscoverChannelAddr;
    property DiscoverChannelSensorPort: string read  f_GetDiscoverChannelSensorPort write
            p_SetDiscoverChannelSensorPort;
    property GraphChannelAgentPort: string read f_GetGraphChannelAgentPort  write
            p_SetGraphChannelAgentPort;
    property GraphChannelInterfacePort: string read f_GetGraphChannelInterfacePort write
            p_SetGraphChannelInterfacePort;
    property GraphChannelNetAddr: string read f_GetGraphChannelAddr write  p_SetGraphChannelAddr;
    property LogErros: Boolean read f_GetLogErros write p_SetLogErros;
    property LogEventos: Boolean read f_GetLogEventos write p_SetLogEventos;
    property Name: string read f_GetName;
    property NetRatio: Integer read f_GetNetRatio write p_SetNetRatio;
    property NodeId: string read f_GetNodeId write p_SetNodeId;
    property NodePosition: string read f_GetNodePosition write p_SetNodePosition;
    property NodesReach: Integer read f_GetNodesReach write p_SetNodesReach;
    property ParentList: T_ArrayString read f_GetParentList write p_SetParentList;
    property RegionId: string read f_GetRegionId write p_SetRegionId;
    property SchemaId: string read f_GetSchemaId;
    property SensorsGroupId: string read f_GetSensorsGroupId write p_SetSensorsGroupId;
    property XmlText: string read f_GeraXml write p_CarregaXml;
  end;

// Classe TPDCRSMessage: rotinas para gerar e manter uma mensagem do PDCRS
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.
TPDCRSMessage = class(TXmlElement)
  public
    constructor Create;
    constructor CreateMessage(v_MessageType:Integer;v_Origin, v_Destination:String; v_TargetType,
            v_Action:Integer; v_Content:String; v_SenderPosition:String);
    constructor CreateName(v_Name:String);
    procedure Initialize;
    property Action: Integer read f_GetAction write p_PutAction;
    property Destination: string read f_GetDestination write p_PutDestination;
    property ErrorCode: Integer read f_GetErrorCode write p_PutErrorCode;
    property MessageType: Integer read f_GetMessageType write p_PutMessagetype;
    property Name: string read f_GetName;
    property Origin: string read f_GetOrigin write p_PutOrigin;
    property SenderPosition: string read f_GetSenderPosition write p_PutSenderPosition;
    property TargetType: Integer read f_GetTargetType write p_PutTargetType;
    property TimeStamp: TDateTime read f_GetTimeStamp write p_PutTimeStamp;
```

```
    property XmlText: string read f_GeraXml write p_CarregaXml;
  end;
```

// Classe TNODEProperties: propriedades de um nó para as rotinas de rede do PDCRS
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.

```
TNODEProperties = class(TXmlElement)
  public
    constructor Create; overload;
    constructor Create(v_Category:T_ConnectionApp); overload;
    constructor Create(v_Category:T_ConnectionApp;v_SensorsGroup, v_Position:String;
                v_NetRatio,v_Reach:Integer;v_ParentList:String);  overload;
    property Category: T_ConnectionApp read f_GetCategory write p_PutCategory;
    property NetRatio: Integer read f_GetNetRatio write p_PutNetRatio;
    property ParentList: string read f_GetParentList write p_PutParentList;
    property Position: string read f_GetPosition write p_PutPosition;
    property Reach: Integer read f_GetReach write p_PutReach;
    property SensorsGroup: string read f_GetSensorsGroup write p_PutSensorsGroup;
  end;
```

// Classe TMessagesArea: controla pacotes ainda não identificados recebidos e a transmitir pelos sensores
// Aplicações usuárias: SensorNodeProcessor.

```
 TMessagesArea = class(TObjectList)
  public
    function Add(v_PDCRSMessage:TPDCRSMessage): Integer;
    procedure Delete(Index:Cardinal);
    procedure p_PegaMensagem(var v_PDCRSMessage:TPDCRSMessage);
    procedure p_PoeMensagem(v_PDCRSMessage:TPDCRSMessage);
    property UsedMemory: Cardinal read v_MemoriaUsada;
  end;
```

// Classe TPacketsArea: controla pacotes já identificados a transmitir pelos sensores
// Aplicações usuárias: SensorNodeProcessor.

```
TPacketsArea = class(TObjectList)
  public
    function Add(v_TuplePacket:TTuplePacket): Integer;
    procedure Delete(Index:Cardinal);
    procedure p_PegaPacote(var v_TuplePacket:TTuplePacket);
    procedure p_PoePacote(v_TuplePacket:TTuplePacket);
    property UsedMemory: Cardinal read v_MemoriaUsada;
  end;
```

// Classe TSensorList: lista de valores de sensores que preenchem uma tupla
// Aplicações usuárias: SensorNodeProcessor.

```
TSensorList = class(TObjectList)
  public
    function Add(v_Sensor:TCustomSensor): Integer; reintroduce; virtual;
    property Items[Index: Integer]: TCustomSensor read f_GetItems write  p_SetItems; default;
  end;
```

// Classe TSensorProc: rotinas de processamento nos nós sensores
// Aplicações usuárias: SensorNodeProcessor.

```
TSensorProc = class(TObject)
  private
    ...
    PDCRSComm1: TPDCRSComm;
    FOnTrace: TOnTrace;
    SensorTuple1: TSensorTuple;
    ValueSensors: Array of TCustomSensor;
```

```
    v_AreaDeEnvio: TPacketsArea;
    v_AreaDeRecepcao: TMessagesArea;
    v_ConsultaAtivada: Boolean;
    v_ProcessandoConsulta: Boolean;
    v_Reiniciar: Boolean;
    v_SensorRelationSchema: TRelationSchema;
    v_SqlQuery: TSqlQuery;
     ...
  protected
    function f_MemoriaDisponivel: Cardinal;
    procedure p_Estagio1;
    procedure p_Estagio3_MonitorarRecursos;
    procedure p_Estagio4_ReceberPacotes;
    procedure p_Estagio5_EnviarPacotes;
    procedure p_PegaPacoteDaAreaDeEnvio(var v_Pacote:TTuplePacket;var v_NrCopiasLocais:integer);
    procedure p_PoePacoteNaAreaDeEnvio(v_Pacote:TTuplePacket);
    ...
  public
    constructor Create(SensorsGroup:String);
    destructor Destroy; override;
    function f_SendIntervalAjustado: Int64;
    function f_SenseIntervalAjustado: Int64;
    procedure p_AtivarSensor;
    procedure p_CancelaSensor;
    procedure p_InterrompeSensor;
    procedure p_TransmiteMensagem(v_MessageType:Integer;v_TargetType:Integer;
                v_Destination:String;v_Message:String;v_Action:Integer); overload;
    procedure p_TransmiteMensagem(v_MessageType:Integer;v_Message:String); overload;
    procedure p_TransmiteMensagem(v_PDCRSMessage:TPDCRSMessage); overload;
    property ActivedQuery: Boolean read v_ConsultaAtivada;
    property EnergiaDisponivel: Cardinal read v_EnergiaDisponivel write p_SetEnergiaDisponivel;
    property EnergiaMaxima: Cardinal read v_EnergiaMaxima write p_SetEnergiaMaxima;
    property GlobalSchema: TGlobalSchema read f_GetGlobalSchema;
    property LogType: t_Logtype read v_LogType;
    property MemoriaDisponivel: Cardinal read f_MemoriaDisponivel;
    property MemoriaMaxima: Cardinal read v_MemoriaMaxima write p_SetMemoriaMaxima;
    property PercAvailEnergy: Single read f_PercEnergiaDisp;
    property PercAvailMem: Single read f_PercMemDisp;
    property SensorsGroup: string read f_GetSensorsGroup write p_SetSensorsGroup;
    property SiteConfig: TSiteConfig read f_GetSiteConfig;
    property SqlQuery: TSqlQuery read v_SqlQuery;
    property Tag: Integer read v_Tag write v_Tag;
    property TraceMsgs: string read v_MsgsTrace write v_MsgsTrace;
  published
    property Trace: Boolean read v_Trace write v_Trace;
    property OnTrace: TOnTrace read FOnTrace write FOnTrace;
  end;
```

// Classe TTuplePacket: repositório para uma tupla de valores de sensores, para transmissão
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.

```
TTuplePacket = class(TXmlElement)
  private
    v_ColTypes: T_ArrayInteger;
    function f_GetIdPacket: string;
  public
    constructor Create(v_ColTypes:T_ArrayInteger); reintroduce;
    property ColTypes: T_ArrayInteger read v_ColTypes write v_ColTypes;
    property IdPacket: string read f_GetIdPacket;
```

```
  end;
```

// Classe TXmlTuple: tupla de valores de sensores
// Aplicações usuárias: BaseStationManager, BaseStationProc, SensorNodeProcessor.

```
  TXmlTuple = class(TXmlElement)
  public
    constructor Create(v_RepositoryPacket:TTuplePacket); reintroduce;
    constructor CreateFromArrays(v_RepositoryPacket:TTuplePacket;var
                v_HeaderRelation,v_AttribValues:T_ArrayVariant);
    function HeaderRelation: T_ArrayVariant;
    procedure TupleToArrays(var v_HeaderRelation,v_AttribValues:T_ArrayVariant);
    function TupleValues: T_ArrayVariant;
    property RepositoryPacket: TTuplePacket read v_RepositoryPacket write v_RepositoryPacket;
    property StrFromAttribs: string read f_GetStrFromAttribs;
    property XmlText: string read f_GeraXml write p_CarregaXml;
  end;
```

// Classe abstrata TCustomSensorValue: simulação de valor para um sensor especializado virtual
// Aplicações usuárias: SensorNodeProcessor.

```
TCustomSensorValue = class
  protected
    …
    function f_ConvValue(Value:String): Variant; virtual; abstract;
    …
  public
    constructor Create; override;
    destructor Destroy; override;
    procedure DoSense;
    property Value: Variant read v_Value;
  published
    property Active: Boolean read f_GetActive write p_SetActive;
    property AllowedValues: TStringList read v_AllowedValues write  p_SetAllowedValues;
    property ColumnName: string read v_ColumnName write p_SetColumnName;
    property ColumnOrder: Integer read v_ColumnOrder write p_SetColumnOrder;
    property InitialValue: Variant read v_InitialValue write p_SetInitialValue;
    property RandomizeAllowedValues: Boolean read v_RandomizeAllowedValues  write
                v_RandomizeAllowedValues;
    property SenseInterval: Integer read f_GetSenseInterval write p_SetSenseInterval;
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
  end;
```

// Classe TBooleanSensorValue: simulação de valor lógico para um sensor virtual
// Aplicações usuárias: SensorNodeProcessor.

```
TBooleanSensorValue = class(TCustomSensorValue)
  protected
    function f_ConvValue(Value:String): Variant; override;
     …
  public
    constructor Create; override;
  published
    property Value: Boolean read f_Value;
  end;
```

// Classe TFloatSensorValue: simulação de número real para um sensor virtual
// Aplicações usuárias: SensorNodeProcessor.

```
TFloatSensorValue = class(TCustomSensorValue)
  protected
    …
```

```
    function f_ConvValue(Value:String): Variant; override;
  public
    constructor Create; override;
  published
    property MaxValue: Single read v_MaxValue write v_MaxValue;
    property MinValue: Single read v_MinValue write v_MinValue;
    property Value: Single read f_Value;
  end;
```

// Classe TIntegerSensorValue: simulação de número inteiro para um sensor virtual
// Aplicações usuárias: SensorNodeProcessor.

```
TIntegerSensorValue = class(TCustomSensorValue)
   protected
    …
    function f_ConvValue(Value:String): Variant; override;
  public
    constructor Create; override;
  published
    property MaxValue: Integer read v_MaxValue write v_MaxValue;
    property MinValue: Integer read v_MinValue write v_MinValue;
    property Value: Integer read f_Value;
  end;
```

// Classe TTimeStampSensorValue: simulação de valor de data e hora para sensor virtual
// Aplicações usuárias: SensorNodeProcessor.

```
TTimeStampSensorValue = class(TCustomSensorValue)
   protected
    …
    function f_ConvValue(Value:String): Variant; override;
  public
    constructor Create; override;
    property Value: TDateTime read f_Value;
  end;
```

// Classe TStaticStringSensorValue: simulação de valor estático texto para sensor virtual
// Aplicações usuárias: SensorNodeProcessor.

```
TStaticStringSensorValue = class(TCustomSensorValue)
  protected
   …
   function f_ConvValue(Value:String): Variant; override;
  public
    constructor Create(AOwner:TComponent); override;
  published
    property Value: string read f_Value write p_SetValue;
  end;
```

// Classe TStaticIntegerSensorValue: simulação de valor estático inteiro para sensor virtual
// Aplicações usuárias: SensorNodeProcessor.

```
TStaticIntegerSensorValue = class(TIntegerSensorValue)
  protected
    …
    function f_Value: Integer;
    procedure p_SetValue(Value: Integer);
  public
    constructor Create(AOwner:TComponent); override;
  published
    property Value: Integer read f_Value write p_SetValue;
  end;
```

// Classe TContinuousSensorValue: simulação de valor inteiro contínuo para sensor virtual
// Aplicações usuárias: SensorNodeProcessor.

```
TContinuousSensorValue = class(TFloatSensorValue)
  protected
    function f_RandomValue: Variant; override;
  public
    constructor Create(AOwner:TComponent); override;
  end;
```

// Classe TTemperaturaSensorValue: simulação de valor de temperatura para sensor virtual
// Aplicações usuárias: SensorNodeProcessor.

```
TTemperaturaSensorValue = class(TFloatSensorValue)
  protected
    function f_RandomValue: Variant; override;
  public
    constructor Create(AOwner:TComponent); override;
  end;
```

// Classe TSensorTuple: simulação de tupla de valores gerada por sensor virtual
// Aplicações usuárias: SensorNodeProcessor.

```
TSensorTuple = class
  public
    constructor Create; override;
    destructor Destroy; override;
    function ArrayVarToStr(v_ArrayVariant:T_ArrayVariant): string;
    function CurrentTuple: T_ArrayVariant;
    procedure DoSense;
    function HeaderRelation: T_ArrayVariant;
    property SensorList: TSensorList read v_SensorList write v_SensorList;
  published
    property Active: Boolean read f_GetActive write p_SetActive;
    property SenseInterval: Integer read f_GetSenseInterval write p_SetSenseInterval;
    property OnCurrentTuple: TNotifyTuple read FOnCurrentTuple write FOnCurrentTuple;
  end;
```

// Classe TSensorsGroupList: cria lista de sensores para a aplicação de simulação de nós sensores
// Aplicações usuárias: SensorNodeProcessor.

```
TSensorsGroupList = class(TXmlElement)
  public
    constructor Create; overload;
    constructor Create(v_FileName:String); overload;
  end;
```

// Classe TPDCRSAgent: núcleo do Agente de Rede
// Aplicações usuárias: NetAgent.

```
 TPDCRSAgent = class
  public
    property ComGrafico: Boolean;
    property ConnectedNodes: TConnectedNodes;
    property ExecThreadGerencia: Boolean;
    property FinalizarThreads: Boolean;
    property Finalizou: Boolean;
    property Inicializou: Boolean;
    property ListaConDados: TList;
    constructor Create;
    destructor Destroy; override;
    function f_HaDadosDisponiveis: Boolean; overload;
```

```
  function f_HaDadosDisponiveis(v_Socket:TSock): Boolean; overload;
  function f_NodePosSize(v_PtConexao:T_ptInfoConexao): TRect;
  function f_RecebeMensagem(var v_Socket:TSock;var v_MessageRx:String): Boolean;
  function f_TransmiteMensagem(v_Socket : TSock;v_ResponseMessage:String): Boolean;
  procedure p_GerenciaFilaRecepcao;
  procedure p_Linha(v_PtConexaoOrigem,v_PtConexaoDestino:T_PtInfoConexao);
  procedure p_LogErros(v_Socket:Integer;v_Mensagem : string;v_MsgException :  string);
  procedure p_LogEventos(v_Socket:Integer;v_Mensagem : string); overload;
  procedure p_LogEventos(v_Socket:Integer;v_Mensagem : string;  v_MsgAux:String); overload;
  procedure p_ProcessaSolicitacao(v_Socket:TSock;  v_XMLRequestMessage:TPDCRSMessage);
 end;
```

// Classe TConnectedNodes: gerencia nós conectados no Agente de Rede
// Aplicações usuárias: NetAgent.

```
TConnectedNodes = class
 public
  constructor Create(v_PDCRSAgent:TPDCRSAgent);
  destructor Destroy; override;
  function f_AtualizaTabelaDeConexoes(v_ExecutandoTransacao:boolean;v_Socket  : TSock;
                v_Gravar:boolean): Boolean;
  function f_EnderecoConexao(v_Socket:TSock): t_PtInfoConexao;
  function f_NodeIdExists(v_NodeId:String): Boolean;
  function f_ObtemConexaoPorIndice(v_Indice:Integer): t_ptInfoConexao;
  procedure p_Enxuga;
  procedure p_ExcluiConexao(v_ptConexao:t_PtInfoConexao);
  procedure p_ObtemConexaoDaLista(v_Socket:TSock;var v_ptConexao:t_PtInfoConexao);
  procedure p_ObtemConexaoPorIndice(v_Indice:Integer;var v_ptConexao:t_PtInfoConexao);
  procedure p_PoeConexaoNaLista(v_Socket:TSock);
  procedure p_RemovePtConexaoDaLista(v_Procurado:t_PtInfoConexao;v_Fechar:Boolean);
  property Count: Integer read v_QuantidadeStream;
  property Items[Index: Integer]: t_InfoConexao read f_GetItems write p_SetItems; default;
  property QuantidadeConn: Integer read f_GetQuantidadeConn;
  property QuantidadeStream: Integer read v_QuantidadeStream;
  property QuantidadeUDP: Integer read v_QuantidadeUDP;
 end;
```

// Classe TXmlElement: rotinas para gerar e manter um elemento XML
// Aplicações usuárias: Todas.

```
 TXmlElement = class
 public
  constructor Create;
  constructor CreateName(v_Name:String);
  destructor Destroy; override;
  procedure AppendAttribute(v_Nome,v_Valor:String);
  procedure AppendChild(v_XmlElement:TXmlElement);
  procedure AppendChildren(v_ChildrenElements:TXmlChildrenElements);
  procedure Initialize;
  procedure LoadFromFile(v_NomeArq:String);
  procedure SaveToFile(v_NomeArq:String);
  property Attributes: TXmlAttributes read v_Atributos;
  property CharacterData: string read f_GetDados write p_SetDados;
  property ChildrenElements: TXmlChildrenElements read v_ChildrenElements;
  property Content: string read f_Conteudo write p_CarregaConteudo;
  property Indent: Boolean read v_Endentar write v_Endentar;
  property Name: string read v_Nome write v_Nome;
  property XmlText: string read f_GeraXml write p_CarregaXml;
 end;
```

```
// Classe TXmlChildrenElements: lista de elementos filhos de um elemento XML
// Aplicações usuárias: Todas.
TXmlChildrenElements = class(TObjectList)
  public
    function Add(v_XmlElement:TXmlElement): Integer; reintroduce; virtual;
    property Items[Index: Integer]: TXmlElement read f_GetItems write p_SetItems; default;
    property ItemsByKey[AttribName:String;KeyValue:Variant]: TXmlElement read f_GetItemsByKey;
    property ItemsByName[Name:String]: TXmlElement read f_GetItemsByName write
              p_SetItemsByName;
  end;

// Classe TXmlAttributes: lista de atributos de um elemento XML
// Aplicações usuárias: Todas.
TXmlAttributes = class(TStringList)
  public
    property StrFromValues: string read f_GetStrFromValues;
    property Values[Name:String]: string read f_GetValues write p_SetValues;
    property ValuesByIndex[Index:Integer]: string read f_GetValuesByIndex write p_SetValuesByIndex;
  end;
```

Apêndice E. Estrutura sintática da SNQL

E.1 Introdução

Um comando SNQL é composto por uma seqüência de símbolos (*tokens*[15]) terminada por um ponto-e-vírgula (;)[16].

Um símbolo pode ser uma palavra chave, um identificador, um literal (ou constante), ou um caractere especial. Geralmente os símbolos são separados por espaço em branco (espaço, tabulação ou nova-linha), mas não há necessidade se não houver ambigüidade (o que geralmente só acontece quando um caractere especial está adjacente a outro tipo de símbolo). Além disso, podem existir comentários na entrada SNQL. Os comentários não são símbolos, mas equivalentes a espaço em branco.

```
USE GLOBALSCHEMA;
CREATE SENSORS TEMPERATURA(SensorId Integer, ReadValue float, RegionId
Integer);
SELECT * FROM TEMPERATURA;
```

Figura E.1 Comandos escritos em SNQL.

A seqüência apresentada na Figura E.1 mostra três comandos SNQL, um em cada linha. Pode haver mais de um comando na mesma linha, e um único comando pode ocupar várias linhas. Cada comando deve ser concluído por um caractere ponto-e-vírgula (;).

A sintaxe do SNQL, seguindo o padrão SQL, não define explicitamente quais símbolos identificam comandos e quais são operandos ou parâmetros. Geralmente os primeiros símbolos são o nome do comando e, portanto, no exemplo mostrado acima se pode dizer que estão presentes os comandos "*USE*", “*CREATE*” e "*SELECT*".

[15] *Token*, neste sentido, é um segmento de texto ou símbolo que será manipulado pelo interpretador, fornecendo um significado ao texto. Ou seja, um token pode ser formado por um ou mais caracteres.
[16] Para efeito de facilitar a descrição da função de símbolos gráficos e caracteres especiais, neste trabalho os apresentaremos entre parênteses.

E.2 Identificadores e palavras-chave

Os símbolos, como *SELECT*, *FROM* e *USE* presentes na Figura E.1 são exemplos de palavras-chave, ou seja, palavras que possuem o significado definido na linguagem aqui mostrada. Os símbolos TEMPERATURA e SensorId são exemplos de identificadores, os quais podem identificar nome de categoria de sensores, colunas ou outros itens do esquema da rede de sensores, dependendo do comando onde são utilizados. A relação completa das palavras-chave pode ser encontrada na Tabela 4.6 (Palavras chave da SNQL).

Os identificadores e as palavras-chave da SNQL, tal como os da SQL padrão, devem iniciar por uma letra do alfabeto latino (a..z, A..Z), ou o caractere sublinhado (_). Os demais caracteres de um identificador, ou da palavra chave, podem ser letras do alfabeto latino, sublinhados e dígitos (0..9). O tamanho máximo em caracteres para o nome de um identificador é 255. Os identificadores e as palavras-chave não fazem distinção entre letras maiúsculas e minúsculas. Portanto,

```
SELECT * FROM TEMPERATURA WHERE REGIONID = 2;
```

pode ser escrito, de forma equivalente, como

```
SelecT * From Temperatura WherE RegionId = 2;
```

E.3 Constantes

Há quatro tipos de constantes com tipo implícito no SNQL: texto, numérica, data e hora e lógicas.

E.3.1 Constantes de texto

Uma constante de texto no SNQL é uma seqüência arbitrária de caracteres delimitada por apóstrofos (') ou aspas duplas (“), como, por exemplo, `'Esta é uma cadeia de caracteres'` ou `“Este é mais um exemplo”`.

Uma forma de escrever um delimitador dentro de uma constante envolta pelo próprio delimitador é colocando o delimitador duplicado dentro da cadeia como, por exemplo, `'Olho D’’Água'` ou `“Este é outro “”exemplo”” do delimitador fazendo parte da cadeia”`.

Outra forma seria delimitar a cadeia com um tipo de delimitador, enquanto se usaria outro delimitador dentro da cadeia, como em `"Esta é 'outra forma' de delimitar cadeias"`.

E.3.2 Constantes numéricas

São aceitas constantes numéricas nas seguintes formas gerais:

```
dígitos
dígitos.[dígitos][e[+-]dígitos]
[dígitos].dígitos[e[+-]dígitos]
dígitose[+-]dígitos
```

Onde dígitos são um ou mais dígitos decimais (0 a 9). Deve haver pelo menos um dígito antes ou depois do ponto decimal, se este for usado. Deve haver pelo menos um dígito após a marca de expoente (`e`), caso esteja presente. Não podem existir espaços ou outros caracteres incorporados à constante.

Deve ser observado que os sinais menos (-) e mais (+) que antecedem a constante não são, na verdade, considerados parte da constante, e sim um operador aplicado à constante. Abaixo são mostrados alguns exemplos de constantes numéricas válidas:

```
42
3.5
-15
4.
.001
5e2
1.925e-3
```

Uma constante numérica não contendo o ponto decimal e nem um indicador de expoente, é presumida, inicialmente, como sendo do tipo *integer* (número inteiro); caso contrário, é assumida como sendo do tipo *float* (número real com ponto flutuante). As constantes que contêm pontos decimais e/ou expoentes são sempre presumidas como sendo do tipo *float*.

O tipo de dado atribuído inicialmente para a constante numérica é apenas o ponto de partida para os algoritmos de resolução de tipo. Na maioria dos casos, a constante é automaticamente convertida no tipo mais apropriado conforme o contexto corrente.

E.3.3 Constantes de data e hora

A interpretação de uma constante do tipo data e hora depende da presença dos caracteres (/) e (:) dentro da cadeia lida. A formação a ser aceita para a data e hora dependerá do formato configurado da data e hora no sistema operacional. Abaixo são mostrados alguns exemplos de constantes de data e hora válidas:

```
01/01/1980
25/12/2000 00:01
00:00:00
01/fevereiro/99 12:00
```

E.3.4 Constantes lógicas

As palavras reservadas *TRUE* E *FALSE* são constantes lógicas da SNQL, que podem ser usada em comparações envolvendo variáveis lógicas em filtros de seleção. *TRUE* representa um resultado verdadeiro, e *FALSE* representa um resultado falso. No exemplo a seguir, a cláusula *Where* testa se a variável *CheckExiste* tem valor igual a verdadeiro:

```
SELECT * FROM Relacao WHERE CheckExiste = TRUE;
```

E.4 Caracteres especiais

Alguns caracteres possuem significado especial para a SNQL. Os detalhes da utilização podem ser encontrados nos locais onde a sintaxe do respectivo elemento é descrita. A seguir será informado um resumo das finalidades destes caracteres, os quais fazem sentido quando são encontrados fora de constantes de texto (identificadores entre aspas).

- Os parênteses (`()`) possuem seu significado usual de agrupar expressões e impor a precedência. Em alguns casos, os parênteses são requeridos como parte da sintaxe fixada para um determinado comando SNQL.
- Vírgulas (,) são utilizadas em algumas construções sintáticas para separar elementos da lista.
- O ponto-e-vírgula (;) finaliza um comando SNQL.

- O asterisco (*) é utilizado em alguns contextos para denotar todos os campos da linha de uma tabela ou de um valor composto. Também podem possuir um significado especial quando utilizado como argumento de algumas funções de agregação, ou quando utilizado antes ou depois do caractere de barra (/) (ver adiante, Comentários), ou dentro de expressões da cláusula de seleção.
- O ponto (.) é utilizado nas constantes numéricas, e para separar os nomes de esquemas, tabelas e colunas.
- Os caracteres (=), (<), (>) servem para compor operadores relacionais utilizados em expressões da cláusula de seleção de uma consulta.
- Os caracteres (+), (-), (/) e (*) fazem o papel de operadores aritméticos nos termos das expressões da cláusula de seleção de uma consulta.

E.5 Tipos de dados

A SNQL possui um conjunto nativo de tipos de dados que são adequados à maioria das aplicações. Estes tipos de dados são utilizados na tipificação de atributos de relações, durante a definição de seus esquemas. A Tabela E.1 mostra os tipos de dado existentes na SNQL.

Tabela E.1 Tipos de dado SNQL

Nome	**Descrição**	**Similaridade de tipos**	
		C/C++	**Object Pascal**
integer	número inteiro com sinal	*int*	*Integer*
float	número real de ponto flutuante	*float*	*Double*
time	data e hora	*double*	*Double*
boolean	lógico (verdade/falso)	*bool*	*Boolean*
text	cadeia de caracteres de comprimento variável	*char**	*String*
textRegionId	equivalente a *text*, mas denota campo de identificação de região, usado na heurística de disseminação de fragmentos de consultas.	*char**	*String*

E.6 Condições e operadores lógicos

Na SNQL os operadores lógicos são utilizados para separar condições de uma lista de condições da cláusula *WHERE* de uma consulta. Essa lista de condições é aqui denominada de filtro de seleção da consulta. Estão disponíveis 3 (três) operadores lógicos: *AND*, *OR* e *NOT*.

Assim, filtro de condições é uma lista de condições separadas por operadores binários *AND* ou *OR*. Adicionalmente, uma condição pode ser precedida pelo operador unário *NOT*. Os operadores *AND* e *OR* são comutativos, ou seja, pode-se trocar a ordem dos operandos esquerdo e direito sem afetar o resultado. Exemplo:

```
condicao1 AND condicao2
```

Condição é uma expressão lógica formada por dois termos. Os dois termos são separados por um operador relacional. Os operadores relacionais são (=), (<), (>), (<>), (<=) e (>=). Exemplo:

```
termo1 = termo2
```

Termo é uma expressão formada por um ou mais itens. Podem ser utilizados neste tipo de expressão os operadores: (+), (-), (*) e (/). Exemplo:

```
item1 + item2
```

Item é um identificador que pode ser um atributo de uma relação ou uma constante. Exemplo:

```
temp.VlrTemp
```

Para sumarizar a definição de filtro de condição, dada a cláusula de seleção abaixo, a Figura E.2 resume a categorização de cada uma de suas partes:

```
… WHERE VlrTemp > 30  AND CodReg = 2  OR Percent*100 < 80
```

<table>
<tr><th>VlrTemp</th><th>></th><th>30</th><th>AND</th><th>CodReg</th><th>=</th><th>2</th><th>OR</th><th>Percent</th><th>*</th><th>100</th><th><</th><th>80</th></tr>
<tr><td>Item</td><td rowspan="2">Op. Rel.</td><td>Item</td><td rowspan="3">Op. Lóg.</td><td>Item</td><td rowspan="2">Op. Rel.</td><td>Item</td><td rowspan="3">Op. Lóg.</td><td>Item</td><td>Op. Arit.</td><td>Item</td><td rowspan="2">Op. Rel.</td><td>Item</td></tr>
<tr><td>Termo</td><td>Term</td><td>Termo</td><td>Term</td><td colspan="3">Termo</td><td>Term</td></tr>
<tr><td colspan="3">Condição</td><td colspan="3">Condição</td><td colspan="5">Condição</td></tr>
<tr><td colspan="13">Filtro de Seleção</td></tr>
</table>

Figura E.2 Elementos de um filtro de seleção

E.7 Comentários

Um comentário é uma seqüência de caracteres dentro de um script de comandos SNQL, começando pelos caracteres (/*) e terminando pelos caracteres (*/), tal como o exemplo a seguir:

```
/* comentário de várias linhas
 * continuação do comentário
 */
```

Os comentários são removidos da cadeia de entrada antes de o interpretador SNQL prosseguir com a análise sintática de uma cadeia de comando.

E.8 Palavras reservadas da SNQL

A Figura E.3 lista todos os símbolos (tokens) que são palavras chave no padrão SNQL. A SNQL faz distinção entre palavras reservadas e não reservadas. De acordo com o padrão, as palavras reservadas são as únicas palavras chave reais, pois nunca são permitidas como identificadores. Como regra geral, se acontecerem erros indevidos do analisador em comandos contendo como identificador qualquer uma das palavras chave listadas, deve-se alterar o nome do identificador para que a ambigüidade identificada não mais ocorra.

AFTER	ALL	AND	AS
ASC	AVG	BEFORE	BOOLEAN
BY	CONTINUOUS	COUNT	CREATE
CYCLE	DATA	DAY	DAYLY
DESC	DISTINCT	DROP	EXISTS
FALSE	FLOAT	FLOATCONT	FLOATTEMP
FROM	GLOBAL	GROUP	HAVING
HOUR	IF	IN	INTEGER
INTERVAL	JOIN	MAX	MILISECOND
MIN	MINUTE	MONTH	MONTHLY
NOT	OR	ORDER	OTHERWISE
RELATION	SCHEDULE	SCHEMA	SECOND
SELECT	SEND	SENSE	SENSORS
START	SUM	TABLE	TEXT
TEXTREGIONID	TIME	TRUE	UNION
UNKNOW	USE	WAIT	WEEKLY
WHERE	WINDOW	YEAR	YEARLY

Figura E.3 Palavras reservadas da SNQL.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo