ISSN 0103-9741

Monografias em Ciência da Computação

nº 02/10

Macroprogramação em Rede de Sensores Sem Fio

Adriano Branco

Noemi Rodriguez

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 02/10 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena Abril, 2010

Macroprogramação em Rede de Sensores Sem Fio

Adriano Branco Noemi Rodriguez

{abranco , noemi}@inf.puc-rio.br

Resumo. A evolução tecnológica dos dispositivos sensores sem fio, associada à redução do custo dos mesmos, está permitindo um crescimento de aplicações para rede de sensores sem fio (WSN). Essas aplicações cada vez mais tendem a utilizar redes complexas e de grandes dimensões. Com isso, torna-se mais evidente a necessidade de ferramentas que facilitem a programação, possibilitando abstrações no nível de grupo de nós ou de rede, em oposição à trabalhosa programação no nível dos nós sensores. Esses modelos de programação são chamados de macroprogramação.

O objetivo deste trabalho é avaliar diferentes modelos de programação em rede de sensores sem fio. Apesar do interesse especial pelas abordagens que executam em plataformas com recursos limitados denominados motes, estaremos avaliando alguns outros modelos que não se encaixam nessa visão.

A nossa avaliação aborda algumas características dos modelos de forma a podermos comparar critérios que vão desde a complexidade de programação até os tipos de aplicações beneficiadas por cada modelo.

Palavras-chave: macroprogramação, sistemas distribuídos, rede de sensores sem fio, RSSF

Abstract. The area of Wireless Sensors Networks (WSN) has experienced in the last years an increase in applications due to the technological evolution and cost reduction of sensor devices. These new applications tend to use larger and more complex networks. The development of these new applications needs tools that facilitate the programming work. These tools must allow new programming abstractions for group and network as opposed to programming at the level of individual sensor nodes. Such models are being called macroprogramming models.

This work evaluates different programming models for wireless sensors network. Although we focus on platforms with limited resources like Berkeley-Style Motes, we also evaluate some other models that do not fit into that description. The evaluation criteria range from the programming complexity to the application types that benefit from each model.

Keywords: macroprogramming, distributed systems, wireless sensor network, WSN

Responsável por publicações:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

Sumário

1 Introdução **1**

2 Redes de Sensores sem Fio **1**
2.1 Definição e Arquitetura 1
2.2 Sistema Operacional 2

3 Classificando os modelos de programação **3**
3.1 Classificação Geral 3
3.2 Classificação por Plataforma de Execução 3
3.3 Escolha dos Modelos de Macroprogramação 5

4 Critérios de Avaliação **5**

5 Descrição dos trabalhos escolhidos **6**
5.1 Regiment 6
5.2 Pleiades 8
5.3 Cosmos 11
5.4 WADL 15
5.5 TinyDB 18
5.6 ATaG 21

6 Comparativo dos modelos **25**

7 Considerações Finais **25**

Referências **28**

1 Introdução

Nos últimos anos, a evolução tecnológica na área de rede de sensores sem fio (WSN - Wireless Sensor Network) gerou um grande crescimento nos estudos envolvendo esses tipos de rede. A tendência dessa evolução é baratear o hardware e, ao mesmo tempo, prover dispositivos para vários tipos de sensoriamento. Isso torna cada vez mais viável a construção de redes de sensores de diferentes dimensões para variados tipos de aplicações.

Toda essa evolução representa novos desafios para a área de programação de sistemas distribuídos, impulsionando trabalhos de Sistemas Operacionais e Middlewares no nível dos nós da rede, e também trabalhos com ferramentas para programação que abstraem o sistema no nível de grupo de sensores ou no nível de rede de sensores.

O objetivo deste trabalho é estudar modelos de programação que aplicam o conceito de macroprogramação em rede de sensores sem fio. No conceito de macroprogramação, o programador é habilitado a trabalhar com a abstração em que a rede é um sistema composto por grupos de sensores. Essa abstração permite a geração de programas de uma forma mais simples, e que podem ser utilizados em uma grande gama de aplicações.

Como trabalho relacionado, podemos citar [1] que dá uma visão ampla dos modelos de programação para WSN, não se restringindo somente aos modelos de macroprogramação. O autor agrupa os modelos em *Node-Level*, *Group-Level* e *Network-level*, esse último divido em *Database* e *Macroprogramming Language*. Também faz uma análise comparativa identificando as características entre os modelos e considerando os principais requisitos: *Energy-efficiency*, *Scalability*, *Failure-resilience* e *Collaboration*.

Na seção 2, situamos o leitor acerca das redes de sensores e de suas características. Já na seção 3, apresentamos uma classificação dos modelos de programação com o objetivo de explicitar os modelos que iremos detalhar. Na seção 4, identificamos os critérios que selecionamos para avaliar os modelos detalhados na seção 5. Na seção 6, apresentamos o resultado resumido da nossa comparação. Finalmente, na seção 7, apresentamos algumas considerações finais.

2 Redes de Sensores sem Fio

> *Uma rede de sensores é um grupo de pequenos sistemas autônomos denominados de Nós Sensores (sensor nodes). Estes nós cooperam para resolver pelo menos um problema comum. Geralmente suas funções incluem algum tipo de percepção de parâmetros físicos.*

A seguir apresentamos a definição e a arquitetura das WSNs e dos Nós Sensores. Em seguida, citamos características dos Sistemas Operacionais específicos para WSNs. Essas informações foram baseadas na referência [2].

2.1 Definição e Arquitetura

Um Nó Sensor sem Fio (WSN - Wireless Sensor Network) é um sistema que tem capacidade de comunicação, computação, sensoriamento e armazenagem. Esses nós miniaturizados operam com restrições severas em termos de recursos disponíveis, como a energia da bateria, memória RAM, largura de banda disponível e poder de processamento. A

figura 1 mostra um diagrama esquemático dos componentes de um nó sensor. Cada nó é composto por um micro-controlador, fonte de alimentação, transceptor de Rádio Frequência (RF), memória externa e sensores.

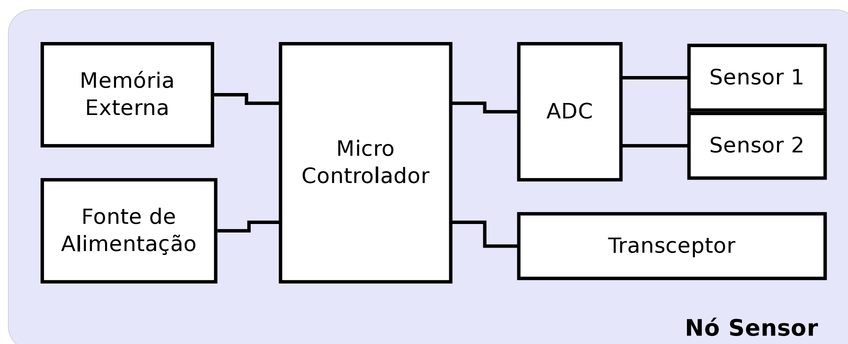

Figura 1: Módulos de um Nó Sensor

Centenas de milhares desses nós sensores são implantados numa larga variedade de aplicações, que vão desde campos vulcânicos até monitoramento ambiental. Esses nós sensores podem se comunicar uns com os outros numa rede ad-hoc usando caminhos de comunicação multi-hop, assim formando uma rede de sensores (WSN).

Em vários tipos de aplicações, após a implantação, os nós sensores são de difícil acesso. Por esse motivo essas redes devem ser autônomas e apresentar longa duração. Quase sempre os nós sensores precisam sobreviver às duras condições ambientais e conservar o máximo de energia possível.

2.2 Sistema Operacional

A funcionalidade básica de um sistema operacional para WSN é esconder, da aplicação, os detalhes de uso e acesso aos componentes de hardware do nó sensor, proporcionando uma interface clara para os usuários do mesmo. O Sistema Operacional deve fornecer serviços básicos como gerenciamento do processador, gerenciamento de memória, gerenciamento de dispositivos, políticas de escalonamento, multi-threading e multitasking. Além dos serviços acima mencionados, o sistema operacional também deve prover serviços específicos, tais como mecanismos apropriados para concorrência, Application Programming Interface (API) para acesso ao hardware, e também reforçar as políticas de boa gestão de energia. Embora alguns desses serviços sejam semelhantes aos serviços dos sistemas operacionais tradicionais, a realização destes serviços em WSN é um problema não-trivial, devido às limitações das capacidades de recursos.

A Figura 2 mostra onde o sistema operacional se situa nas camadas de software da WSN. O *Core kernel* do sistema operacional executa em cada nó individual. Em cima dele, são executados o middleware e as aplicações que interagem através dos nós. Há uma distinção clara entre os serviços que devem ser providos pelo sistema operacional e pelo middleware em sistemas tradicionais. Isso é mascarado em WSN devido à necessidade de interações entre camadas, característica importante para esse tipo de sistemas.

Nós sensores em WSN são de baixo acoplamento e algumas vezes implantados em uma grande área geográfica. A escala da rede, algumas vezes, é da ordem de milhares de nós sensores. Cada nó individual tem seu próprio poder de processamento e um sistema de software para ser executado. Nós cooperam entre si através da troca de mensagens. Em um ambiente distribuído, o objetivo de um sistema operacional deve ser o de gerir

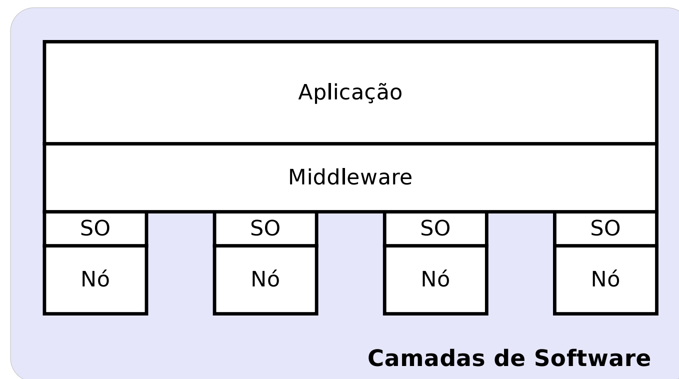

Figura 2: Camadas de software

os diferentes nós espalhados em regiões, fazendo o conjunto parecer como uma única entidade virtual. Isso envolve prover transparência de comunicação, transparência de falhas, e suporte para aplicações heterogêneas e escaláveis.

3 Classificando os modelos de programação

3.1 Classificação Geral

Vamos utilizar como referência a taxonomia proposta em [1] que apresenta uma classificação para os modelos de programação. Os autores apresentam na figura 3 uma divisão dos modelos de programação aplicados em WSN dividindo em baixo-nível (*Platform-centric*) e alto-nível (*Application-centric*). O primeiro está relacionado ao nível de nós, enquanto que o segundo está relacionado à programação no nível de grupo ou de rede. Dentro da divisão de programação no nível de rede são identificados os subgrupos *Database Abstraction* e *Macroprogramming language*.

Considerando que muitas vezes as redes de sensores são utilizadas para coletar dados de sensoriamento remoto, a abstração de banco de dados (*Database Abstraction*) é intuitiva, embora essa abordagem em rede de sensores tenha problemas diferentes de uma aplicação convencional de banco de dados. Um exemplo de problema é que uma simples consulta gera um "afunilamento" de mensagens trafegadas pela rede até o servidor central. A outra abordagem é proporcionar novas linguagens de macroprogramação (*Macroprogramming language*), que possuem uma cobertura de aplicações mais ampla que a abordagem de dados. O objetivo das linguagens de macroprogramação é efetuar a programação do ponto de vista macroscópico em que cada nó e os respectivos dados podem ser acessados sem considerar as comunicações de baixo nível entre os nós.

Linguagens de macroprogramação são uma abordagem complementar em relação à abordagem de banco de dados e se destinam a proporcionar maior flexibilidade.

3.2 Classificação por Plataforma de Execução

Criamos uma classificação adicional para identificar a plataforma de execução do modelo de macroprogramação. Iremos dividir em três grupos.

O primeiro grupo contém os trabalhos que utilizam plataformas com recursos bem

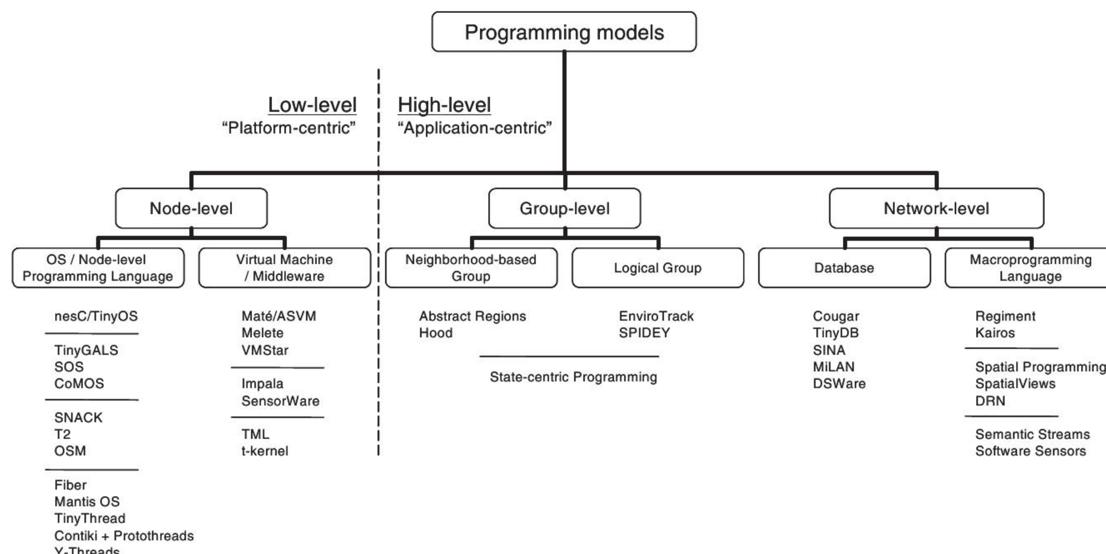

Figura 3: Taxonomia proposta em [1] para Modelos de Programação em RSSF.

limitados e que são dedicados para aplicações de rede de sensores sem fio. Essas plataformas são denominadas *Motes*[1].

O segundo grupo contempla os trabalhos que utilizam plataformas com mais recursos e com capacidade de execução de aplicações gerais, aqui representado por HP-iPAQ[2], MiniPCs[3] e Servidor de Aplicação [4].

O terceiro grupo contempla plataformas com recursos intermediários entre o primeiro e o segundo grupo, mas ainda com foco em rede de sensores. Esse grupo é representado por plataformas do tipo Sun-SPOT[5] e Stargate[6].

Como exemplos de trabalhos do primeiro grupo, temos Regiment [3, 4], Kairos [5], Pleiades[7] [6] e TinyDB [7]. Os quatro utilizam o sistema operacional TinyOS [8] em plataformas do tipo Mote. Também temos Cosmos [9], que executa em multiplataforma utilizando mOS em Mote.

No segundo grupo, utilizando Linux no HP-iPAQ, temos Spatial Programming [10], SpatialViews [11] e DRN [12]. Utilizando MiniPC, temos Semantic Streams [13], e já executando em Servidores de Aplicação, teremos WADL[8] [14].

Para o terceiro grupo, temos ATaG[9] [15], utilizando JVM na plataforma Sun-SPOT. Também consideramos Cosmos [9], que executa em multiplataforma podendo utilizar Linux no Stargate.

[1]Motes - Temos como exemplo o modelo MICA Estilo-Berkeley que é fabricado pela empresa Crossbow Technology com os nomes Mica2 e MicaZ.

[2]HP-iPAQ - PDAs (*Personal Digital Assistants*) fabricados pela HP (Hewlett-Packard)

[3]MiniPC - Sistema é centralizada num miniserver como o modelo "Upont Cappuccino TX-3 Mini PC"

[4]Servidor de Aplicação - utilização de máquinas mais genéricas com maior capacidade de recursos de processamento.

[5]Sun-SPOT - *Sun Small Programmable Object Technology* é um mote baseado em J2ME desenvolvido pela Sun Microsystems.

[6]Stargate - É uma processador mais potente para ser utilizado como *gateway* numa rede de sensores do tipo mote. É fabricado pela Crossbow e utiliza o processador PXA255 da Intel.

[7]Pleiades é o sucessor de Kairos.

[8]Por ser mais recente, WADL não é listado na taxonomia proposta em [1]

[9]ATaG também não foi citado na taxonomia proposta em [1]

3.3 Escolha dos Modelos de Macroprogramação

Temos interesse especial em trabalhos aplicados diretamente em rede de sensores sem fio que utilizam motes. O primeiro critério de seleção dos modelos de programação para nossa avaliação foi a escolha de trabalhos incluídos no primeiro grupo da classificação das plataformas de execução. Incluímos nessa seleção mais alguns modelos fora do critério original de forma a aumentar a diversidade de modelos da nossa avaliação.

Com isso, estaremos avaliando os seguintes trabalhos:

- Regiment [3,4] (TinyOS/Mote)
- Pleiades [6] (TinyOS/Mote)
- Cosmos [9] (mOS/Mote)
- TinyDB [7] (TinyOS/Mote)
- WADL [14] (Servidor de Aplicação)
- ATaG [15] (JVM/Sun-SPOT)

4 Critérios de Avaliação

O objetivo principal deste trabalho é identificar e comparar as principais características de alguns modelos de programação aplicados em rede de sensores sem fio. Para isso, selecionamos os seguintes critérios de avaliação:

Base do Modelo - Origem ou a base do modelo utilizado na linguagem de macroprogramação.

Estilo de programação - Classificação do estilo de programação, podendo ser Imperativo ou Declarativo.

Linguagem do usuário - Identifica qual é a principal linguagem de programação disponibilizada para o usuário.

Visão da Rede - A visão que o usuário tem da rede, identificando as abstrações disponibilizadas.

Pontos Positivos - Pontos positivos e benefícios do modelo proposto.

Tipos de Aplicações - Para quais tipos de aplicações o modelo é mais adequado. Aqui dividimos em:

1. Envio periódico de dados da rede para a estação base - utilizado em aplicações de coleta de dados.
2. Leitura eventual de dados - utilizado em aplicações de controle.
3. Interação dentro da rede - um exemplo de aplicação é o caso do carro procurando a vaga mais próxima.
4. Envio eventual de dados da rede para a estação base - aplicação do tipo alarme, por exemplo, o alarme de incêndio numa floresta.

Aplicação Exemplo - Quais aplicações foram usadas como exemplo na apresentação do trabalho.

Pontos Críticos - Identifica os principais pontos críticos do modelo no contexto de rede de motes.

5 Descrição dos trabalhos escolhidos

Nesta seção, iremos descrever cada trabalho escolhido, colocando ênfase nos critérios de avaliação pré-selecionados.

5.1 Regiment

Regiment [3,4] é composto por uma linguagem de alto nível nos moldes das linguagens funcionais similares à *Haskell*. Especificamente Regiment utiliza o modelo de Linguagens Funcionais Reativas (FRP). Regiment implementa "funções" para atender as necessidades de macroprogramação em rede de sensores sem fio.

O programador vê a rede como um conjunto de fluxos de dados espacialmente distribuídos. O programador pode manipular esses conjuntos de fluxos, em que a relação entre os nós pode ser definida como topológica ou geográfica. Regiment fornece um conjunto de primitivas para processamento de dados sobre fluxos individuais, manipulando as regiões, realizando agregação sobre uma região, e disparando novos processamentos na rede.

Bons exemplos de aplicações em Regiment incluem a estimativa de um gradiente ou o contorno no campo de sensores, ou o uso da comunicação local entre vizinhos para estimar a localização de um veículo específico.

5.1.1 Arquitetura

Na figura 4, apresentamos os módulos que compõem a arquitetura do sistema Regiment, indicando o fluxo de transformação do código escrito em Regiment até o binário para ser executado no Mote.

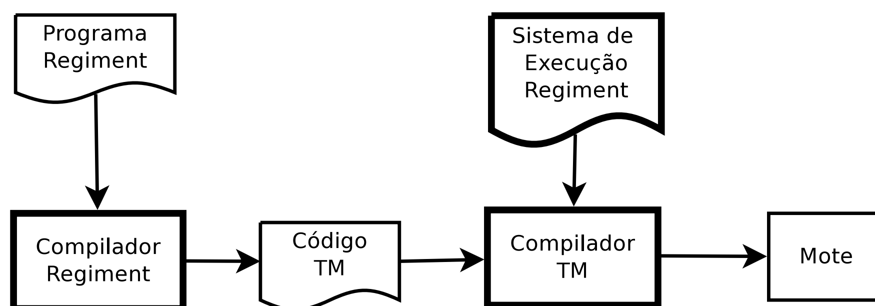

Figura 4: Arquitetura do sistema Regiment

Regiment é composto por um compilador que, principalmente, mapeia as estruturas globais da macroprogramação para o nível de nó. Para isso, é utilizada uma linguagem intermediária entre Regiment e a linguagem nativa e o ambiente de execução do nó sensor. Essa linguagem intermediária é chamada de *Token Machines*. O modelo de *Token Machines* adapta-se ao modelo de eventos do TinyOS, possuindo mecanismos simples baseados no conceito de tokens para chamadas locais, chamadas remotas e armazenamento de dados.

A propagação controlada de tokens também é utilizada para a determinação de grupos de nós, por exemplo, para determinar os vizinhos *n-hops* a partir de um determinado nó ancora.

5.1.2 Linguagem

O modelo de Regiment está baseado no conceito de *Signals* e *Regions*. Os *Signals* representam valores de sensores para um determinado tempo *t* e são representados por uma sequência de leituras discretas no tempo (*stream*). As *Regions* representam um conjunto de *Signals*, esse conjunto pode ser variável no tempo, como por exemplo o conjunto de nós com a temperatura maior que um determinado valor. O conjunto também pode variar por falhas na comunicação, defeitos nos sensores ou com a adição de novos sensores.

Considerando *Signals* e *Regions* como tipos básicos, Regiment tem um pequeno conjunto de operações para esses tipos:

- **smap** *f stream* - aplica a função *f* para cada dado de *stream*, retornando uma nova *stream*.
- **rmap** *f region* - aplica a função *f* para cada stream de *region*, retornando uma nova *region*.
- **rfilter** *p region* - retorna uma nova *region* composta somente pelos *signals* que atendam a condição testada na função *p*.
- **rfolder** *f region* - retorna um novo *signal* formado pela agregação na função *f* para os *signals* que compõem *region*.
- **khood**, **circle** e **knearest** - são utilizados para formação de *Regions* que iniciam a partir de um nó âncora, respectivamente *k* radio hops, distancia física radial e *k* mais próximos geograficamente.
- **gossip** e **table_gossip** - são utilizados para formação de *Regions* baseados no alcance broadcast.

Regiment também provê um conjunto básico de recursos de linguagem encontrado em outras linguagens funcionais. Além das operações primitivas sobre *Signals* e *Regions*, o programador pode utilizar condicionais e passagem de funções por valor.

5.1.3 Programa Exemplo

O programa exemplo utilizado pelos autores em [3], calcula a temperatura média entre todos sensores da rede.

```
fun dosum(temp, (sumtemp, count)) {
    (sumtemp+temp, count+1)
}
tempreg = rmap(fun(nd){sense("temp",nd)}, world);
sumsig = rfold(dosum, (0,0), tempreg);
avgsig = smap( fun((sum,cnt)) {sum / cnt},sumsig);
BASE <- avgsig
```

O programa inicia aplicando *rmap* em *world* para obter uma *Region* com todas as temperaturas lidas dos sensores da rede. A região *world* é uma região especial que representa todos os sensores da rede.

Em seguida, utiliza-se *rfold* para agregar esse conjunto em um *signal* chamado `sumsig`. Cada valor de `sumsig` é composto por 2 componentes, um contém a soma da temperatura e o outro é um contador dos valores que contribuíram nessa soma. Essa agregação é feita com auxilio da função `dosum`.

Finalmente, *smap* é aplicada para dividir o soma das temperaturas pelo contador. O valor resultante é enviado para a estação base com o comando `BASE <- avgsig`.

5.1.4 Pontos críticos

Duas características básicas do modelo de Regiment implicam em algumas limitações.

Em primeiro lugar, a linguagem compilada associada ao modelo de execução baseado numa biblioteca específica para *spanning tree*, dificulta a criação de aplicações com comportamento altamente dinâmico, com múltiplos modos de operação e que mudam os caminhos de dados.

Em segundo lugar, aplicações que requerem outros serviços de comunicação (como roteamento *any-to-any*) não podem fazer uso de Regiment.

5.2 Pleiades

Pleiades [6] é uma linguagem que permite uma visão centralizada de rede de sensores com programas escritos de forma sequencial. Baseado no modelo SPMD, Pleiades estende a linguagem C com construções que permitem aos programadores especificar formas simples de execução concorrente, com suporte a variáveis locais aos nós individuais e variáveis compartilhadas entre grupos de nós. Pleiades também fornece uma abstração síncrona para sensores e timers.

Pleiades, basicamente, disponibiliza o comando `cfor`, que é uma extensão do comando `for`, e que permite a execução concorrente para um grupo de nós. Quando necessário, o sistema de execução de Pleiades mantém a serialização e o controle de *locks* e *deadloks*.

O modelo de paralelização de Pleiades pode ser aplicado tanto para processamento entre um grupo de nós, como para toda rede. Esse modelo atende às aplicações que necessitem de interações dentro da rede e aplicações de envio periódico de dados.

5.2.1 Arquitetura

Na figura 5 apresentamos os módulos que compõem a arquitetura do sistema Pleiades, indicando o fluxo de transformação do código escrito em Pleiades até o binário para ser executado no Mote.

Pleiades é composto por um compilador que transforma um programa Pleiades para um código NesC. Esse código pode então ser compilado e linkado com o TinyOS e com o sistema de execução de Pleiades. O compilador Pleiades particiona o programa original em unidades de trabalhos em NesC, e o sistema de execução orquestra a execução dessas unidades através da rede de sensores. Essa orquestração define o que será executado em cada nó, otimizando a processamento local e a troca de eventos entre os nós.

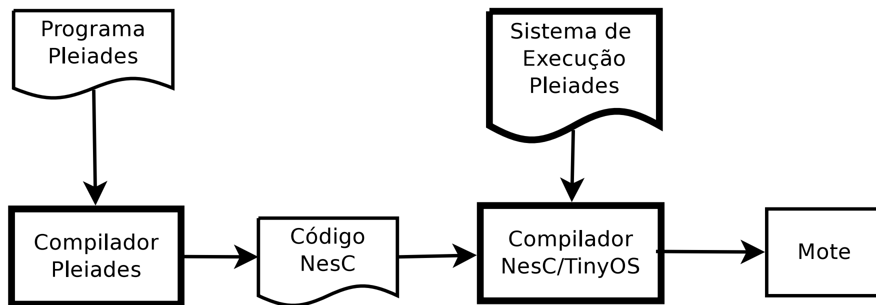

Figura 5: Arquitetura do sistema Pleiades

5.2.2 Linguagem

O programador escreve um programa Pleiades na forma sequencial e utiliza os novos construtores para definir o comportamento de processamento concorrente entre os nós da rede.

Pleiades estende a linguagem C com os seguintes construtores:

- Definição de variáveis
  - `node` - tipo de variável para abstração de um nó simples da rede
  - `nodeset` - tipo de variável para abstração de um iterador para um conjunto desordenado de nós.
  - `nodelocal` - atributo na definição para indicar que a variável terá uma versão para cada nó da rede.
  - Variável central - uma variável que não é definida com o atributo `nodelocal`, será tratada como uma variável compartilhada por todos nós da rede.
  - `loose` - atributo na definição para indicar que a variável terá um modelo de consistência relaxado. Utilizado nos casos em que a variável não precisa da semantica de serialização.
- Atribuição e Funções gerais
  - `<var>@<node>` - Endereça a variável `<var>` no nó indicado por `<node>`.
  - `get_network_nodes()` - Retorna um *nodeset* com os nós disponíveis na rede.
  - `get_neighbors(n)` - Retorna um *nodeset* com os *n*'s vizinhos a one-hop. Implementado com envio de um broadcast a partir do nó.
  - `wait(var)` - Dispara a leitura de uma variável de sensor ou timer e aguarda o final para continuar a execução. Utilizado como abstração síncrona para sensores e timers.
- Concorrência
  - `cfor` - Loop para iteração sobre um *nodeset*. Executa o corpo do loop de forma concorrente para cada nó de *nodeset*. Quando necessário, garante a serialização, na execução.

5.2.3 Programa Exemplo

Como exemplo, os autores em [6] utilizam uma aplicação para ajudar motoristas a achar uma vaga de estacionamento nas ruas de uma cidade. A aplicação considera que cada vaga terá um sensor de baixo custo que identifica se a vaga está ocupada ou não. O objetivo é achar uma vaga livre mais próximo do destino do motorista. Para facilitar a explanação será considerado como distância o contador de hops na rede.

obs: O exemplo a seguir omite algumas das funções utilizadas.

```
1: #include "pleiades.h"
2: boolean nodelocal isfree=TRUE;
3: nodeset nodelocal neighbors;
4: node nodelocal neighborIter;

5: void reserve(pos dst) {
6:  boolean reserved=FALSE;
7:  node nodeIter,reservedNode=NULL;
8:  node n=closest_node(dst);
9:  nodeset loose nToExamine=add_node(n, empty_nodeset());
10: nodeset loose nExamined=empty_nodeset();

11: if(isfree@n) {
12:  reserved=TRUE; reservedNode=n;
13:  isfree@n=FALSE;
14:  return;
15: }

16:  while(!reserved && !empty(nToExamine)){
17:  cfor(nodeIter=get_first(nToExamine);nodeIter!=NULL;
          nodeIter = get_next(nToExamine)){
18:   neighbors@nodeIter=get_neighbors(nodeIter);
19:   for(neighborIter@nodeIter=get_first(neighbors@nodeIter);
          neighborIter@nodeIter!=NULL;
          neighborIter@nodeIter=get_next(neighbors@nodeIter)){
20:    if(!member(neighborIter@nodeIter,nExamined))
21:     add_node(neighborIter@nodeIter,nToExamine);
22:   }
23:   if(isfree@nodeIter){
24:    if(!reserved){
25:     reserved=TRUE; reservedNode=nodeIter;
26:     isfree@nodeIter=FALSE;
27:     break;
28:    }
29:   }
30:   remove_node(nodeIter,nToExamine);
31:   add_node(nodeIter,nExamined);
32:  }
33: }
34:}
```

Nesse programa Pleiades, quando um carro chega perto de uma área desejada, uma vaga perto do destino indicado é achada e reservada. Faz-se isso invocando `reserve` e passando a localização desejada. O procedimento `reserve` acha o nó sensor mais próximo e verifica se a vaga está livre. Se sim, a vaga é reservada para o carro. Se não, os nós vizinhos são verificados recursivamente e concorrentemente.

Na linha 7, a variável `n` identifica o nó mais próximo do destino, e o código nas linhas 11 até 15 encerra o processamento se essa vaga estiver livre. O `cfor` da linha 17 e o `for` da linha 19 executam o processamento concorrente e recursivo para os vizinhos de cada nó sem vaga livre. Esses loops são executados até que se ache uma vaga livre, ou até que não existam mais nós para verificar.

5.2.4 Pontos críticos

O modelo de Pleiades aplica-se muito bem para o caso de aplicações que precisem construir uma árvore de roteamento pelos nós da rede, sempre partindo de um determinado nó. Programas Pleiades não se aplicam muito bem para o caso de aplicações de coleta em redes de sensores que exigem diversos processamentos eventuais e não contíguos, como por exemplo as aplicações de monitoração com múltiplos alarmes.

O modelo de concorrência e compartilhamento de variáveis gera uma necessidade intensiva de troca de eventos entre os nós da rede. Outro ponto crítico é a centralização do processamento em cima do loop *cfor*. Uma vez que haja uma falha no loop *cfor*, consequentemente, teremos uma falha na aplicação.

5.3 Cosmos

COSMOS [9] é uma arquitetura para macroprogramação de redes de sensores heterogêneos que se baseia na abstração de fluxo de dados distribuído. É composto por uma linguagem de programação mPL e um sistema operacional mOS.

A macroprogramação em mPL é feita pela definição de um grafo, onde os vértices são *componentes funcionais* (FCs) ou dispositivos, e as arestas são definidas por *associações de interação* (IAs).

mPL é uma linguagem declarativa, onde o programador deve declarar os componentes funcionais (FCs) que vai utilizar, e, em seguida, deve declarar as "ligações" (IAs) entre os componentes, formando uma hierarquia do fluxo de dados.

A codificação dos componentes funcionais deve ser feita previamente utilizando a linguagem de programação C, com base em regras e restrições impostas pela arquitetura de Cosmos. As IAs são canais de comunicação criados na inicialização da aplicação e podem estar interligando componentes locais, ou também interligando componentes de diferentes nós. As ligações feitas pelas IAs podem ser 1-1, 1-n ou n-1, permitindo, dessa forma, a definição da hierarquia desejada no fluxo de dados.

O compilador mPL particiona o grafo do sistema em sub-grafos relativos a cada mote da rede. O sistema de execução trata de forma transparente as IAs locais e as IAs entre motes.

A flexibilidade de Cosmos permite a utilização para diferentes tipos de aplicações, necessitando somente da criação dos componentes funcionais necessários.

5.3.1 Arquitetura

Na figura 6, apresentamos os módulos que compõem a arquitetura Cosmos, indicando o fluxo de transformação do código escrito em mPL até o binário para ser executado no Mote.

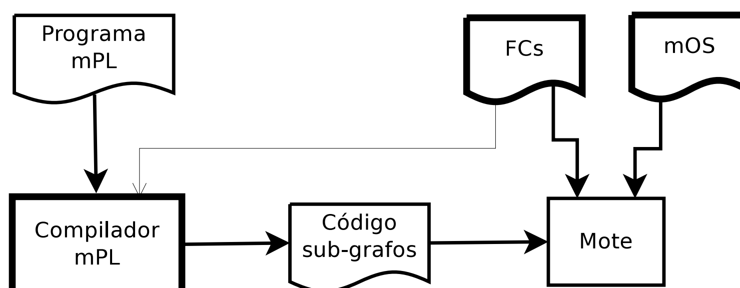

Figura 6: Arquitetura Cosmos

A arquitetura de Cosmos divide em 3 níveis o código carregado no mote. No primeiro nível, deve-se carregar o mOS, em seguida, os componentes funcionais e, finalmente, a informação do sub-grafo da aplicação mPL para o respectivo mote. Essa carga pode ser feita toda de uma vez na operação de carga da memória flash, ou os motes podem estar previamente carregados com o mOS e se utilizarem dos recursos de carga via mensagens para as FCs e Sub-grafos.

Os componentes funcionais podem ser dos seguintes tipos:

SFCs - São componentes que implementam serviços de sistema (por exemplo: serviços de rede), e existem independentemente da aplicação.

LFCs - São abstrações de programação em alto nível em mPL e não são visíveis ao programador. O compilador automaticamente seleciona as LFCs conforme as abstrações utilizadas no programa mPL.

FCs - São os componentes criados ou reutilizados para a aplicação mPL.

5.3.2 Linguagem

A programação em mPL é feita de forma declarativa. Antes o programador cria em C os componentes funcionais. A codificação dos componentes deve seguir algumas regras que possibilitam a integração com mPL. No programa mPL, o programador define as regras de instanciação dos componentes funcionais e, em seguida, através das IAs, define o fluxo de dados entre as instâncias dos componentes.

Um macroprograma em mPL é composto pelos seguintes tipos de expressões:

Enumerations - Associação de identificadores a valores inteiros. Normalmente utilizado para fornecer identificadores globais únicos. Têm a mesma sintaxe de *enum* de C e o compilador importa automaticamente dos arquivos de COSMOS.

Declarations - Essas expressões permitem declarações de dispositivos e FCs. Também são importados diretamente dos arquivos de Cosmos.

Instances - Muito parecido com a declaração de objetos como instâncias de classes, essas expressões permitem a criação de instâncias lógicas de FCs e dispositivos, fornecendo parâmetros de instanciação.

Capability constraints - Instâncias de FCs ou Dispositivos que herdam as restrições de capacidade das declarações de FCs ou Dispositivos.

IA description - Essa seção descreve de forma compreensiva as conexões de entrada e saída das instâncias dos FCs e os Dispositivos utilizados no macroprograma. O compilador identifica erros de tipos e ligações.

Contract predicates - Permite a utilização de abstrações de alto nível nas descrições de IAs.

A criação de regiões é possível com a utilização de expressões que definem a comunicação n-n para o grupo de vizinhos n-hops.

5.3.3 Programa Exemplo

O programa exemplo utilizado pelos autores em [9] executa uma monitoração estrutural. Esse programa exibe valores de aceleração máxima e avalia a resposta de frequência do edifício. Apresenta também o espectrograma resultante caso a aceleração esteja acima de um limite especificado. Um controlador (Ctrl) realimenta o valor limite, com base em dados agregados da rede. Isso economiza recursos do sistema até que um evento interessante desencadeie a necessidade de uma visão mais detalhada.

Na figura 7, apresentamos o diagrama do fluxo de dados do programa exemplo. Cada bloco representado pela área pontilhada é instanciado nos respectivos tipos de motes ou no servidor.

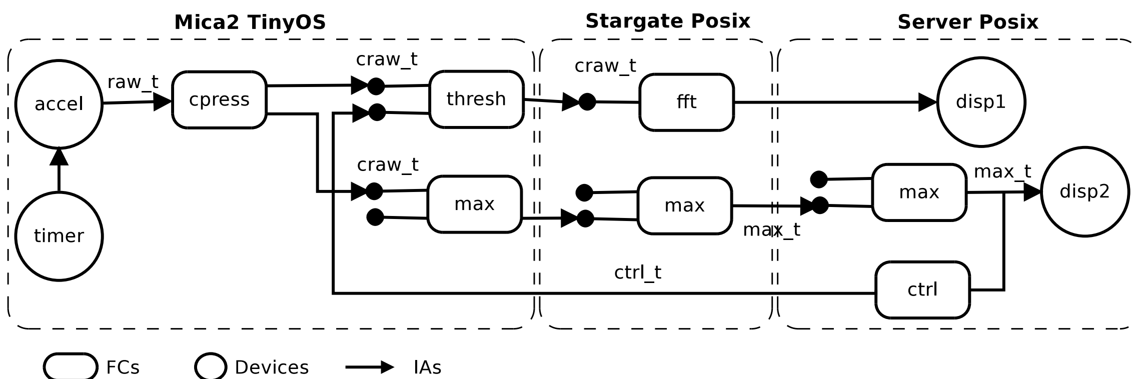

Figura 7: Programa exemplo - Diagrama do fluxo de dados

A seguir apresentamos a listagem do programa exemplo mPL.

```
// declarations (auto import)
%accel_x    : mcap = MCAP_ACCEL_SENSOR,
              device = ACCEL_X_SENSOR, out[raw_t];
%cpress_fc : { mcap = MCAP_ANY, fcid = FCID_CPRESS,
               in[raw_t], out[craw_t] };
%thresh_fc : { mcap = MCAP_ANY, fcid = FCID_THRESH,
               in[craw_t, ctrl_t], out[craw_t] };
%ctrl_fc    : { mcap = MCAP_ANY, fcid = FCID_CTRL,
               in[max_t], out[ctrl_t] };
%max_fc     : { mcap = MCAP_ANY, fcid = FCID_MAX,
```

```
               in[craw_t, max_t], out[max_t]};
%disp       : mcap = MCAP_UNIQUE_SERVER,
              device = DISPLAY, in[ * ];
%fft_fc     : { mcap = MCAP_FAST_CPU, fcid = FCID_FFT,
               in[craw_t], out[freq_t] };

// logical instances
accel_x   : accel(12);
disp      : disp1, disp2;
cpress_fc : cpress;
thresh_fc : thresh(250);
max_fc    : max;
fft_fc    : fft;
ctrl_fc   : ctrl;

// refining capability constraints
@ on_mote = MCAP_ACCEL_SENSOR : thresh, cpress;
@ on_srv = MCAP_UNIQUE_SERVER : ctrl;

start_ia
timer(30) -> accel;
accel     -> cpress[0];
cpress[0] -> thresh[0], max[0];
thresh[0] -> fft[0];
fft[0]    -> disp1;
max[0]    -> ctrl[0], disp2 | max[1];
ctrl[0]  --> thresh[1];
end_ia
```

Para dar mais clareza, apresentamos as declarações dos Dispositivos e FCs. De forma geral, essas declarações não são escritas no arquivo de código mPL. Ainda na parte das declarações, `mcap` define as restrições para o FC ou Dispositivo. Por exemplo, para o componente *fft* (*Fast Fourier Transform*) é definido `mcap = MCAP_FAST_CPU`, o que significa que essa FC somente pode ser instanciada em máquinas com CPU rápidas como os nós Stargate.

Na parte da criação das instâncias lógicas, note que algumas instanciações passam parâmetros para inicialização. Por exemplo, em "`thresh_fc : thresh(250);`", cada instância do componente *Threshold* é inicializada com um valor inicial 250.

Na parte de descrição da IA, temos a representação do grafo IA. Lembramos que `timer` é uma palavra chave em mPL e o seu parâmetro define um intervalo de disparo em milissegundos. Uma instância FC à esquerda da seta precisa fornecer o índice do valor de saída, enquanto que a instância do FC à direita deverá fornecer o índice de entrada. No nosso exemplo, a primeira saída de `ctrl[0]` será conectada na segunda entrada de `thresh[1]`. Dispositivos não precisam desses índices porque eles têm somente uma entrada e uma saída.

O uso de "->" indica comunicação local ou pela rede 1-1. A melhor escolha é feita automaticamente pelo compilador. Já o uso de "-->" indica uma comunicação 1-n.

O uso de "|" permite mesclar e atualizar os dados. Na geração dos sub-grafos para diferentes tipos de nós, o compilador mPL verifica a localização dos FCs envolvidos na

operação para estabelecer a comunicação local ou via rede.

5.3.4 Pontos críticos

A criação de novas FCs está fora do escopo do programador mPL, necessitando de um desenvolvedor mais especializado e que conheça as regras e restrições impostas pela arquitetura de Cosmos.

5.4 WADL

A linguagem declarativa WADL [14] é uma variação de uma linguagem de descrição de arquitetura (ADL-Architecture Description Language) com foco em aplicações dinâmicas baseadas em sensores. A linguagem utiliza o conceito Produtor-Consumidor como padrão de comunicação, no qual os sensores produzem dados e os módulos de processamento consomem os dados produzidos.

A característica "dinâmica" de WADL é permitir que os módulos produtores e consumidores possam ser introduzidos ou retirados em tempo de execução. Para isso, utilizam o conceito de descoberta de serviços da arquitetura SOA.

O programador WADL define o fluxo de dados entre os módulos especificando as características dos módulos, ao invés de especificar um módulo individual. Em tempo de execução, é feita a "ligação" dos módulos instanciados, inclusive permitindo a reconfiguração automática com a inserção de novos módulos ou a exclusão de módulos existentes.

O modelo produtor-consumidor de WADL favorece aplicações típicas de interação dentro da rede ou de coleta de dados.

5.4.1 Arquitetura

Na figura 8, apresentamos os módulos que compõem a arquitetura WADL. Todo sistema está baseado na especificação OSGi (*Open Services Gateway Initiative*) [16].

O padrão OSGi especifica uma arquitetura de componentes dinâmicos para Java. Essa arquitetura funciona como um middleware orientado a serviço para desenvolvimento de aplicações modulares em Java (SOA). As principais implementações do OSGi são para servidores de aplicações.

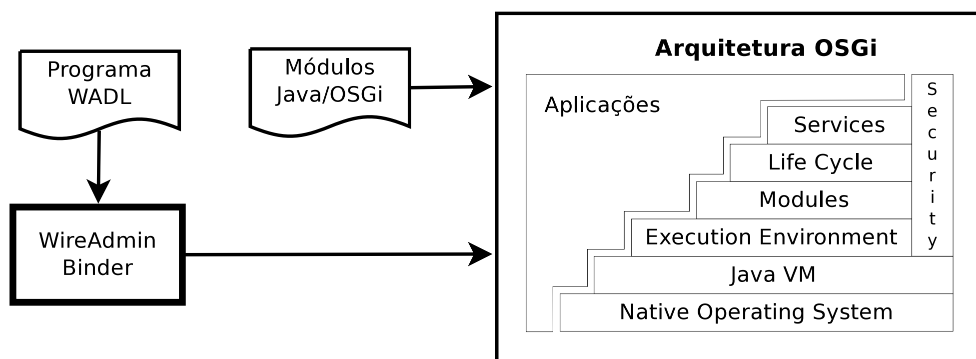

Figura 8: Arquitetura WADL

Na programação WADL, é assumido que os módulos sensores e de processamento de dados são módulos aplicativos desenvolvidos na arquitetura OSGi. O programador WADL então define, no programa WADL, as regras de "ligação" entre esses módulos, isto é, as regras para a criação dos canais de comunicação entres os módulos. Para isso, foi desenvolvido especialmente um módulo OSGi (*WireAdminBinder*) para a interpretação do programa WADL e a criação automática dos canais.

A arquitetura OSGi prevê a possibilidade de instanciação dinâmica de novos módulos ou de desativação de módulos existentes. O *WireAdminBinder* aproveita essa característica para providenciar a manutenção das "ligações" (canais) em função da manutenção dos módulos.

5.4.2 Linguagem

Basicamente a linguagem WADL é composta por três conceitos:

- **WireApp** - Representa uma aplicação composta pelos módulos produtores (*producers*), módulos consumidores (*consurmers*) e as respectivas conexões (*wires*).
- **WireSet** - Representa a definição das conexões entre os produtores e consumidores. Essas definições são baseadas em filtros, permitindo que a identificação de produtores e consumidores possa ser feita por atributos.
- **Property** - Define as propriedades de QoS para *Wires* e *Producers*. Essas propriedades ajudam a controlar o fluxo de dados e a carga dos Consumidores.

Por causa da característica de instanciação dinâmica, o ciclo de vida de uma aplicação WADL está em função dos módulos ativos e das respectivas conexões. Para o controle do ciclo de vida das conexões, deve-se definir políticas de comportamento para quando um módulo é removido da aplicação.

As políticas para destruição de uma conexão (*wire*) são:

- IF_DISCONNECTED - Destrói uma conexão se um produtor ou consumidor é removido.
- WHILE_PRODUCER - Destrói uma conexão se somente o Produtor é removido.
- WHILE_CONSUMER - Destrói uma conexão se somente o Consumidor é removido.
- KEEP_ALIVE - A conexão será persistente.

5.4.3 Programa Exemplo

A seguir, apresentamos a listagem do programa exemplo utilizado pelos autores em [14]. Esse exemplo é uma aplicação de central de incêndio.

O programa gera mensagens de alerta quando forem detectados valores anormais para os medidores de temperatura ou de nível de fumaça de qualquer sala de um prédio.

```
<?xml version="1.0" encoding="UTF-8"?>
  <wireapp id="building.FireCentral"
          description="A Fire central wired application"
```

```
        acyclic="true">
  <!-- a many-to-one wireset without wire properties -->
  <!-- connects temperature sensors to the fire central -->
  <!-- + keepAlive remove policy -->
  <wireset
    id="temperature2central"
    description="temperatures consumed by the fire central"
    producers-filter="(&(|(wireadmin.producer.flavors=
                      *org.osgi.util.measurement.Measurement)
                      (wireadmin.producer.flavors=
                      *javax.measure.Measure))(unit=SI.K))"
    consumers-filter="(service.pid=building.firecentral.temperature)"
    mandatory="false"
    removepolicy="KEEP_ALIVE"
  />
  <!-- current rooms smoke level to the fire central -->
  <!-- + whileConsumer remove policy -->
  <wireset
    id="smoke2central"
    description="smoke level producers consumed by the fire central"
    producers-filter="(wireadmin.producer.flavors=
                      *com.acme.data.SmokeLevel)"
    consumers-filter="(service.pid=building.firecentral.smoke)"
    removepolicy="WHILE_CONSUMER"
    mandatory="true"
  />
    <property
      name="wireadmin.filter"
      value="(wirevalue.elapsed>=2000)"
      type="java.lang.String"
    />
  </wireset>
</wireapp>
```

O programa WADL descreve a topologia da aplicação, nesse exemplo o *WireApp* é composto por dois *WireSet*.

O primeiro *WireSet* (`temperature2central`) liga o consumidor `building.firecentral.temperature` aos produtores `Measurement` ou `Measure` com a unidade de temperatura kelvin (`SI.K`). Também utiliza a política `KEEP_ALIVE` para que a "ligação" persista, mesmo havendo intermitência nos módulos.

O segundo *WireSet* (`smoke2central`) trata de forma semelhante os sensores de fumaça, com a diferença que a "ligação" pode ser destruída se o módulo consumidor for desativado (`WHILE_CONSUMER`).

A propriedade utilizada (`wirevalue.elapsed>=2000`) força a atualização dos valores em pelo menos 2000 milissegundos.

5.4.4 Pontos críticos

O sistema foi desenvolvido originalmente para aplicações de sensores, não necessariamente para aplicações de redes de sensores sem fio. A programação tem foco somente

nas conexões. O desenvolvimento dos módulos tem que ser feito em Java diretamente sobre a arquitetura OSGi.

As implementações disponíveis da arquitetura OSGi são, normalmente, para servidores de aplicações utilizando Java. A complexidade desse tipo de implementação inviabiliza a utilização em rede de sensores com dispositivos do tipo Mote.

As características do OSGi são dedicadas para aplicações mais tradicionais em rede de computadores. Dessa forma, não atende pontos críticos específicos para rede de sensores sem fio, principalmente para aplicações mais densas e complexas.

5.5 TinyDB

TinyDB [7] é um sistema de processamento de queries para extração de dados de rede de sensores. TinyDB implementa uma versão simplificada da linguagem SQL e adiciona algumas extensões para adaptação ao contexto de rede de sensores sem fio.

TinyDB abstrai a rede de sensores como se fosse uma tabela (*sensors*) composta por "tuplas" dos valores de cada nó para cada instante no tempo. Na realidade, esses valores somente serão "materializados" quando for necessário satisfazer uma consulta.

O controle principal do TinyDB é feito num servidor interligado a rede de sensores através do nó raiz. As consultas são disparadas no servidor e propagadas de forma hierárquica para cada nó da rede. Os resultados são retornados na forma de *streams* pelo caminho inverso. No retorno dos dados, é possível a utilização de filtros e funções agregadoras.

É possível parametrizar uma consulta para funcionar de forma pontual ou periódica, ou ainda definir se vai ser disparada a partir de um evento customizado no mote.

TinyDB inclui também algoritmos para utilização eficiente de energia. Esses algoritmos otimizam o consumo dos dispositivos, influenciando o intervalo e o caminho da troca de mensagens. Uma das opções na programação da consulta é permitir ao sistema determinar o melhor período de coleta, em função da disponibilidade de energia e do tempo de funcionamento requerido para o sensor.

5.5.1 Arquitetura

Na figura 9, apresentamos os módulos que compõem a arquitetura TinyDB, indicando o fluxo de transformação das consultas até o binário para ser executado no Mote.

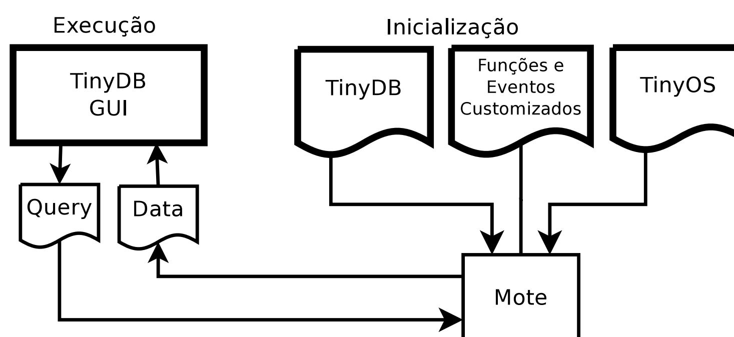

Figura 9: Arquitetura TinyDB

O TinyDB é composto por um módulo que é executado no mote utilizando o TinyOS,

e por uma interface gráfica que é executada no servidor para a construção das consultas.

A parte que é executada no mote é responsável pelo roteamento das mensagens e pelo processamento local das consultas. Deve-se incluir nessa parte as funções e os eventos customizados para serem utilizados nas queries.

A interface gráfica dá suporte para criação das consultas, envio de comandos para os motes e recebimento dos resultados. Também existem as opções de criação direta de scripts e utilização de API cliente para processamento e armazenamento de dados no servidor.

5.5.2 Linguagem

A linguagem utilizada é uma simplificação do SQL tradicional utilizado em sistemas de banco de dados. Foram criadas algumas variações para atender ao contexto de rede de sensores sem fio.

Abaixo segue a definição da sintaxe SQL do TinyDB.

```
[ON [ALIGNED] EVENT event-type[{paramlist}]
                    [boolop event-type{paramlist} ... ]]
  SELECT [NO INTERLEAVE] <expr>| agg(<expr>) |
                       temporal agg(<expr>), ...
    FROM [sensors | storage-point], ...
    [WHERE {<pred>}]
    [GROUP BY {<expr>}]
    [HAVING {<pred>}]
  [OUTPUT ACTION [command |
                  SIGNAL event({paramlist}) |
                  (SELECT ... ) ] |
  [INTO STORAGE POINT bufname]]
  [SAMPLE PERIOD seconds
         [[FOR n rounds] |
          [STOP ON event-type [WHERE <pred>]]]
         [COMBINE { agg(<expr>)}]
         [INTERPOLATE LINEAR]] |
    [ONCE] |
    [LIFETIME seconds [MIN SAMPLE RATE seconds]]
```

Os principais elementos do SQL utilizado no sistema TinyDB são:

- ON EVENT - Indica que a query será disparada por um evento de baixo nível ou por um evento gerado por outra query. O código do evento de baixo nível deverá ser compilado no TinyOS.
- SELECT-FROM-WHERE - Basicamente é o mesmo que no SQL tradicional. A clausula FROM pode fazer referência à tabela *sensors* ou à alguma tabela local ao mote gerada por outra query (*materialization point*).
- GROUP BY, HAVING - Cláusulas para as funções de agregação.

- OUTPUT ACTION - No lugar de retornar valores para o nó raiz, executa a chamada de uma função de baixo nível. Pode ser utilizado para acionamentos externos.
- SIGNAL - Utilizado para uma query gerar um evento no mote.
- INTO STORAGE POINT - Cria uma tabela local (*materialization point*) com o resultado da query.
- SAMPLE PERIOD - Define o intervalo periódico de coleta para uma query.
- FOR - Define a duração total de coleta para uma query.
- STOP ON - Indica que uma query periódica será parada por um evento.
- COMBINE - Especifica uma função de agregação para os casos de *Joins* com diferentes taxas de coleta. Um *Join* somente pode ser feito entre uma query normal e uma tabela local.
- ONCE - Utilizado no lugar de SAMPLE PERIOD para indicar uma coleta simples e não periódica.
- LIFETIME - Utilizado no lugar de SAMPLE PERIOD para indicar que o período de coleta deve ser ajustado automaticamente em função da energia disponível, de forma a durar o tempo de LIFETIME indicado.

5.5.3 Programa Exemplo

A seguir, apresentamos alguns exemplos de queries para o TinyDB. Esses exemplos fora retirados de [7].

Query Básica

```
SELECT nodeid, light, temp
  FROM sensors
  SAMPLE PERIOD 1s FOR 10s
```

Criação de tabela local (*recentLight*)

```
CREATE
  STORAGE POINT recentlight SIZE 8
  AS (SELECT nodeid, light FROM sensors
  SAMPLE PERIOD 10s)
```

Join da tabela *Sensors* com a tabela local *recentLight*

```
SELECT COUNT(*)
  FROM sensors AS s, recentLight AS rl
  WHERE rl.nodeid = s.nodeid
  AND s.light < rl.light
  SAMPLE PERIOD 10s
```

Agregação calculando valor médio

```
SELECT AVG(volume),room FROM sensors
  WHERE floor = 6
  GROUP BY room
  HAVING AVG(volume) > threshold
  SAMPLE PERIOD 30s
```

Query disparada pelo evento *bird-detect*

```
ON EVENT bird-detect(loc):
  SELECT AVG(light), AVG(temp), event.loc
  FROM sensors AS s
  WHERE dist(s.loc, event.loc) < 10m
  SAMPLE PERIOD 2 s FOR 30 s
```

Query que dispara o evento *hot*

```
SELECT nodeid,temp
  WHERE temp > thresh
  OUTPUT ACTION SIGNAL hot(nodeid,temp)
  SAMPLE PERIOD 10s
```

Query com a opção LIFETIME

```
SELECT nodeid, accel
  FROM sensors
  LIFETIME 30 days
```

5.5.4 Pontos críticos

As funções customizadas de agregação e de eventos de baixo nível devem ser escritas em NesC, dessa forma perdem-se as facilidades da linguagem SQL.

Apesar da possibilidade de criação de funções customizadas e da utilização de eventos locais, a arquitetura cliente-servidor baseada no modelo de banco de dados inviabiliza a construção de aplicações mais complexas que necessitem de processamento e interações localizadas.

O modelo de propagação genérico com a utilização de filtros locais impõe atividades desnecessárias em grupos de motes quando se utilizam queries seletivas.

5.6 ATaG

ATaG (Abstract Task Graph) [15] é um modelo de programação orientada a dados. ATaG utiliza-se da programação declarativa para facilitar a especificação da aplicação de forma macro, enquanto que as funções detalhadas são implementadas na linguagem imperativa da plataforma de execução.

A parte declarativa de ATaG é composta de dois módulos, um módulo define o fluxo de processamento de dados e o outro módulo define as características da rede de sensores. Durante a compilação, o sistema combina esses dois módulos com as funções desenvolvidas na linguagem imperativa para gerar a configuração final a ser implantada.

O programador ATaG utiliza uma ferramenta visual para definir o diagrama de fluxo de dados da aplicação. Nesse diagrama, são representados os componentes funcionais relativos às atividades (*abstract task*), aos componentes de dados (*abstract data*) e aos canais de comunicação entre os componentes funcionais e de dados (*abstract channel*). Para cada componente de atividade e de dado, deve existir uma especificação na linguagem imperativa suportada pela plataforma de execução.

A característica de fluxo de dados de ATaG facilita o desenvolvimento de aplicações para coleta de dadaos, geração de eventos ou interação dentro da rede.

5.6.1 Arquitetura

Na figura 10, apresentamos os módulos que compõem a arquitetura ATaG, indicando o fluxo de transformação do código até o binário para ser executado no Mote.

Em [17], é apresentada uma ferramenta gráfica chamada de "Srijan". Essa ferramenta permite ao programador executar todas as etapas de desenvolvimento, desde a programação até a implantação nos motes. Nesse caso, foi utilizado Java como linguagem imperativa na plataforma Sun-SPOT.

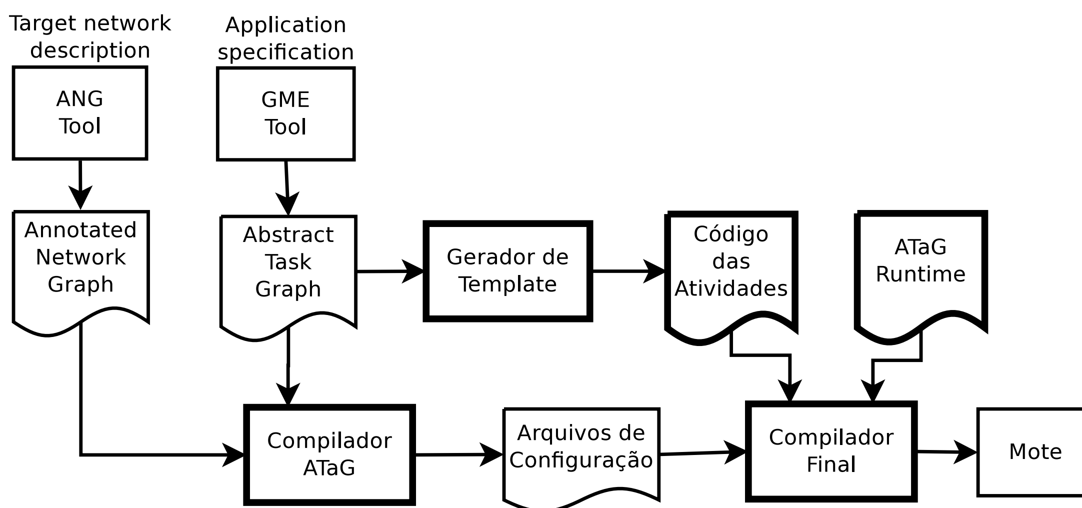

Figura 10: Arquitetura ATaG

Seguem algumas definições para o entendimento da arquitetura.

- ANG Tool - Ferramenta para definição da rede de sensores. O arquivo ANG contém informações como número de nós, o coordenador de cada nó, conectividade de rede, etc...
- GME Tool - Ferramenta para definição da aplicação. É baseada numa ferramenta de modelagem genérica GME [18], que foi devidamente configurada para a sintaxe do ATaG. Essa mesma ferramenta dá suporte para a implementação dos compiladores.
- É importante ressaltar que a partir da programação declarativa nas ferramentas

gráficas, são gerados os templates para a programação imperativa em Java das funcionalidades (*tasks*) da aplicação.

- Os arquivos de configuração contém informações customizadas para cada nó da rede de acordo com o papel na rede.

5.6.2 Linguagem

Vamos focar aqui na programação ATaG específica do módulo declarativo que define a aplicação.

Um programa ATaG é um conjunto de declarações abstratas (*abstract declarations*) que podem ser de 3 tipos:

Abstract task - Representa um tipo de processamento na aplicação. Para cada *task* deve existir um programa em JAVA. As tabelas 1 e 2 descrevem as anotações que podem ser associadas as declarações das *tasks*.

Abstract data - Representa o tipo de dados que podem ser trocados entre as *tasks*. Os itens *abstract data* não possuem anotações.

Abstract channel - Define a associação entre *task* e *data* indicando quem é o produtor e/ou consumidor. As tabelas 3 e 4 descrevem as anotações que podem ser associadas as declarações dos *channels*.

Value[:parameter]	**Descrição**
one-on-node-ID:*id*	Cria uma instância da *task* no nó *id*.
one-anywhere	Cria uma instância da *task* em qualquer nó da rede.
nodes-per-instance:[/]*n*	Cria uma instância da *task* para cada *n* nós da rede. Se for utilizado "/", cria domínios proporcionais de nós para cada instância.
area-per-instance:[/]*area*	O mesmo de "nodes-per-instance". Os domínio são definidos por área da implantação
spatial-extent:$x_1, y_1, x_2, y_2, ...$	Cria uma instância da *task* em todos os nós implantados dentro do polígono definido pelas coordenadas.

Tabela 1: Anotações para Abstract Task - Instanciação

Value[:parameter]	**Descrição**
periodic:*p*	Agenda a *task* para execução periódica de *p* segundos.
any-data	Agenda a *task* para execução quando pelo menos um dos dados de entrada estiver disponível.
all-data	Agenda a *task* para execução somente quando todos dados de entrada estiverem disponíveis.

Tabela 2: Anotações para Abstract Task - Regra de Disparo

Value	Descrição
push	O sistema de execução dos nós origem tem a responsabilidade de enviar os dados para os nós destinos.
pull	O sistema de execução dos nós destinos deverá requisitar os dados aos nós origem.

Tabela 3: Anotações para Abstract Channel - Inicialização

Value[:parameter]	Descrição
[¬]local	O canal se aplica ao pool local de dados. O símbolo de negação "¬" exclui o pool local e pode ser usado em conjunto com outros qualificadores.
neighborhood-hops:*n*	O canal inclui todos os nós vizinho a *n*-hop do nó com a *task* instanciada.
neighborhood-distance:d	O canal inclui todos os nós na distância *d* do nó com a *task* instanciada.
all-nodes	O canal inclui todos os nós da rede.
domain	O canal inclui os nós definidos no domínio atribuído em "nodes-per-instance" ou "area-per-instance".
parent	O canal inclui o nó "pai" definido na hierarquia da topologia virtual da rede.
children	O canal inclui os nós "filhos" definidos na hierarquia da topologia virtual da rede.

Tabela 4: Anotações para Abstract Channel - Associação de Nós

5.6.3 Programa Exemplo

Um programa ATaG é, na realidade, um diagrama representativo do fluxo de dados da aplicação. Na figura 11, apresentamos um diagrama ATaG de uma aplicação para monitoração ambiental utilizando sensores de temperatura. Esse exemplo foi utilizado pelos autores em [15].

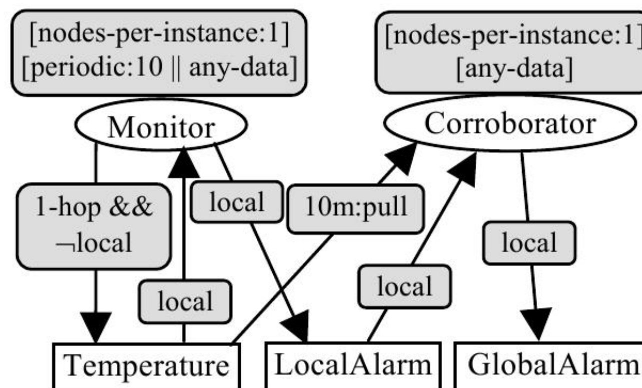

Figura 11: Programa ATaG para monitoração ambiental

As *abstract tasks* são representadas por elipses brancas, os *abstract datas* por retângulos brancos e os *abstract channels* pelas setas. Os retângulos cinzas representam as anotações

das *tasks* e dos *channels*.

A ideia básica é que cada nó verifique se o gradiente de temperatura de seus vizinhos é maior que um determinado *threshold*, se sim, o nó gera um alarme local para que seja avaliada a temperatura de uma área maior.

Nesse caso, as abstrações de dados são: `Temperature`, que representa leituras do sensor de temperatura, `Local Alarm` e `Global Alarm`, que representam as condições em que o *threshold* é atingido.

As abstrações de atividades são o `Monitor`, que calcula o gradiente local, e o `Corroborator`, que analisa as leituras de uma área maior.

As anotações são fundamentais para definir o correto funcionamento da aplicação.

Por exemplo, a anotação "`[node-per-instance:1] [periodic:10 ||any-data]`" indica que `Monitor` será instanciado em todos os nós do sistema, executado periodicamente e executado também quando uma instância de `Temperature` for disponibilizada.

A anotação "`1-hop && ¬local`" indica que a instância `Temperature`, produzida por `Monitor`, é transmitida para todos vizinhos 1-hop e não adicionada no pool de dados local.

A atividade `Corroborator` será disparada pela produção de um `Local Alarm` gerado por `Monitor`, para então ler (`pull`) todas as instâncias de `Temperature` dentro do raio de 10 metros. Se necessário, poderá gerar um `Global Alarm`.

5.6.4 Pontos críticos

A implementação atual utilizando Java assume uma rede de sensores com mais recursos do que uma rede de sensores do tipo mote. Para o caso da utilização de motes com TinyOS e NesC, o trabalho de programação imperativa será bem mais complexo.

O trabalho estudado não aborda os detalhes do suporte *runtime* e da programação da topologia da rede.

6 Comparativo dos modelos

Na tabela 5, apresentamos um resumo comparativo da nossa avaliação para os 6 trabalhos.

7 Considerações Finais

Selecionamos um conjunto de diferentes modelos de programação para rede de sensores. A maioria dos modelos é para aplicações de rede de sensores sem fio utilizando nós do tipo mote ou um pouco mais potentes. Além disso, incluímos mais um modelo de programação que utiliza nós com mais recursos de processamento, porém com uma visão diferente dos modelos anteriores. Comparamos esses modelos conforme alguns critérios de avaliação.

De uma forma geral, identificamos que, para atender aos requisitos de macroprogramação, os modelos avaliados disponibilizam facilidades para algum tipo de abstração no nível de grupos e de rede, e também implementam abstrações para grupo de vizinhos.

Critério	Regiment	Pleiades	Cosmos	WADL	TinyDB	ATaG
Modelo Base	Programação Funcional Reativa	Modelo de programação SPMD	Linguagem de programação de fluxo de dados	Linguagem para descrição de arquitetura (ADL)	SQL em banco de dados distribuídos	Diagrama de Fluxo de Dados
Estilo de programação	Declarativo	Imperativo	Declarativo + Imperativo	Declarativo	Declarativo	Declarativo + Imperativo
Linguagem do usuário	Regiment	Extensão da linguagem C	mPL + C	WADL	SQL adaptado	Diagrama ATaG + Java
Visão da Rede	Distribuição espacial e variável no tempo. Agregação por região.	Nó e grupo de nós vizinhos	Componentes funcionais distribuídos num grafo direcionado, com opção de agregação nos vizinhos n-hops.	Grupos de nós com características semelhantes	Banco de dados com elementos hierarquizados na topologia da rede. Vizinhos n-hops	Componentes e canais distribuídos pela topologia da rede. Comunicação entre vizinhos n-hops, por raio ou por partição.
Pontos Positivos	Facilita a abstração para grupos geográficos e topológicos; Bom desempenho na comunicação.	Processamento paralelo entre um grupo de nós vizinhos, com controle de locks e deadlocks;	Modelo flexível e configurável; Permite atualização de código. *over-the-air*.	Abstrai a rede com base nas características dos componentes.	Solução ideal para aplicações de coleta de dados	Separa o desenho da aplicação do desenho da rede e converge os dois no processo de compilação.
Tipos de Aplicações	Envio periódico ou eventual;	Interação dentro da rede; Envio periódico;	Depende da implementação dos componentes funcionais;	Interação dentro da rede; Envio periódico;	Envio periódico ou eventual;	Envio periódico ou eventual; Interação dentro da rede;
Aplicação exemplo	Alarme de nuvem química	Estacionamento de carros	Medidas estruturais dinâmicas	Central de incêndio predial	Queries para coleta de dados	Monitoração Ambiental com Alarmes
Pontos Críticos	Não atende aplicações com comportamento dinâmico; Feito para consultas de longa duração.	Baixa eficiência nas trocas de mensagens	Depende da implementação prévia dos componentes funcionais escritos em C;	Utiliza plataforma SOA que é muito complexa e não adequada para aplicações em motes. Desenvolvimento dos módulos em Java.	Modelo cliente-servidor para coleta de dados. Customizações em NesC. Modelo de broadcast gera baixa eficiência na troca de mensagens.	Depende da implementação de componentes na linguagem imperativa do sistema

Tabela 5: Resumo Geral da Avaliação

Para apoiar essas abstrações, os modelos avaliados implementam alguma topologia pré-configurada de comunicação entre os nós.

Regiment implementa uma linguagem funcional com abstrações de nó e grupo. A topologia da comunicação é baseada no modelo de spanning tree.

Pleiades estende a linguagem C para apoiar as abstrações de nó e grupo com facilidades de concorrência e paralelização do tipo SPMD. O procedimento de comunicação é definido dinamicamente e depende da topologia pré-configurada.

COSMOS é uma arquitetura que contém uma linguagem de macroprogramação mPL que depende da implementação de componentes funcionais escritos em C. A topologia de comunicação é baseada num grafo direcionado.

WADL é uma linguagem de definição de arquitetura que define os canais de comunicação entre componentes da aplicação. Os componentes são identificados por suas características e devem ser desenvolvidos em Java para uma plataforma SOA.

TinyDB utiliza uma linguagem baseada em SQL para abstrair a rede de sensores como se fosse um banco de dados. São fornecidos alguns algoritmos para agregação, mas é possível implementar novas funções utilizando NesC. A topologia da comunicação é baseada num modelo hierárquico.

ATaG é uma linguagem declarativa com uma interface visual que permite ao programador diagramar o fluxo de dados entre componentes funcionais e de dados. Tem algumas facilidades para definição de grupos e fluxo de dados. Os componentes referenciados no diagrama devem ser implementados em Java.

De uma forma geral, podemos concluir que cada tipo de aplicação pode se beneficiar mais ou menos da combinação do procedimento de programação, do tipo de abstração fornecido e da topologia de comunicação implementada.

Apesar da simplificação no processo de programação de cada modelo avaliado, modelos como COSMOS, WADL e ATaG apresentam um modelo declarativo extremamente simplificado, adicionando grande flexibilidade com a composição por meio dos componentes funcionais. Vale ressaltar que existe um custo adicional para a programação dos componentes funcionais.

Outro ponto de atenção é o sistema de execução disponibilizado por cada modelo. Quanto mais facilidades e recursos disponíveis para simplificar a macroprogramação, menor será a possibilidade de ser executado em dispositivos de poucos recursos do tipo mote.

Baseado numa arquitetura de componentes funcionais, entendemos que existe um espaço importante para estudos que identifiquem os principais componentes funcionais e que atenderiam a maioria dos tipos de aplicações para rede de sensores sem fio. Disponibilizando esses conjuntos de componentes numa forma padronizada, podemos afirmar que o esforço de programação de uma nova aplicação se resumiria ao trabalho de macroprogramação numa linguagem declarativa.

Referências

[1] SUGIHARA, R.; GUPTA, R. K.. **Programming models for sensor networks: A survey**. ACM Trans. Sen. Netw., 4(2):1–29, 2008.

[2] REDDY, A. M. V.; KUMAR, A. P.; JANAKIRAM, D. ; KUMAR, G. A.. **Wireless sensor network operating systems: a survey**. Int. J. Sen. Netw., 5(4):236–255, 2009.

[3] NEWTON, R.; MORRISETT, G. ; WELSH, M.. **The regiment macroprogramming system**. In: IPSN '07: PROCEEDINGS OF THE 6TH INTERNATIONAL CONFERENCE ON INFORMATION PROCESSING IN SENSOR NETWORKS, p. 489–498, New York, NY, USA, 2007. ACM.

[4] NEWTON, R.; WELSH, M.. **Region streams: functional macroprogramming for sensor networks**. In: DMSN '04: PROCEEEDINGS OF THE 1ST INTERNATIONAL WORKSHOP ON DATA MANAGEMENT FOR SENSOR NETWORKS, p. 78–87, New York, NY, USA, 2004. ACM.

[5] GUMMADI, R.; GNAWALI, O. ; GOVINDAN, R.. **Macro-programming wireless sensor networks using kairos**. Distributed Computing in Sensor Systems, p. 126–140, 2005.

[6] KOTHARI, N.; GUMMADI, R.; MILLSTEIN, T. ; GOVINDAN, R.. **Reliable and efficient programming abstractions for wireless sensor networks**. PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, p. 200–210, 2007.

[7] MADDEN, S. R.; FRANKLIN, M. J.; HELLERSTEIN, J. M. ; HONG, W.. **Tinydb: an acquisitional query processing system for sensor networks**. ACM Trans. Database Syst., 30(1):122–173, 2005.

[8] LEVIS, P.; MADDEN, S.; POLASTRE, J.; SZEWCZYK, R.; WHITEHOUSE, K.; WOO, A.; GAY, D.; HILL, J.; WELSH, M.; BREWER, E. ; CULLER, D.. **Tinyos: An operating system for sensor networks**. In: IN AMBIENT INTELLIGENCE. Springer Verlag, 2004.

[9] AWAN, A.; JAGANNATHAN, S. ; GRAMA, A.. **Macroprogramming heterogeneous sensor networks using cosmos**. EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, p. 159–172, 2007.

[10] BORCEA, C.; INTANAGONWIWAT, C.; KANG, P.; KREMER, U. ; IFTODE, L.. **Spatial programming using smart messages: Design and implementation**. In: ICDCS '04: PROCEEDINGS OF THE 24TH INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS (ICDCS'04), p. 690–699, Washington, DC, USA, 2004. IEEE Computer Society.

[11] NI, Y.; KREMER, U.; STERE, A. ; IFTODE, L.. **Programming ad-hoc networks of mobile and resource-constrained devices**. PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, p. 249–260, 2005.

[12] INTANAGONWIWAT, C.; GUPTA, R. ; VAHDAT, A.. **Declarative resource naming for macroprogramming wireless networks of embedded systems**. in International Workshop on Algorithmic Aspects of Wireless Sensor Networks (ALGOSENSORS), 2005.

[13] WHITEHOUSE, K.; ZHAO, F. ; LIU, J.. **Semantic streams: A framework for composable semantic interpretation of sensor data**. Wireless Sensor Networks, p. 5–20, 2006.

[14] CERVANTES, H.; DONSEZ, D. ; TOUSEAU, L.. **An architecture description language for dynamic sensor-based applications**. In: CONSUMER COMMUNICATIONS AND NETWORKING CONFERENCE, 2008. CCNC 2008. 5TH IEEE, p. 147–151, Jan. 2008.

[15] BAKSHI, A.; PRASANNA, V. K.; REICH, J. ; LARNER, D.. **The abstract task graph: a methodology for architecture-independent programming of networked sensor systems**. EESR '05: Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services, p. 19–24, 2005.

[16] OSGI-ALLIANCE. **Osgi - the dynamic module system for java**. http://www.osgi.org, October 2005.

[17] PATHAK, A.; GOWDA, M. K.. **Srijan: a graphical toolkit for sensor network macroprogramming**. In: ESEC/FSE '09: PROCEEDINGS OF THE 7TH JOINT MEETING OF THE EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND THE ACM SIGSOFT SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING ON EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND FOUNDATIONS OF SOFTWARE ENGINEERING SYMPOSIUM, p. 301–302, New York, NY, USA, 2009. ACM.

[18] LEDECZI, A.; MAROTI, M.; BAKAY, A.; KARSAI, G.; GARRETT, J.; THOMASON, C.; NORDSTROM, G.; SPRINKLE, J. ; VOLGYESI, P.. **The generic modeling environment**. In: WORKSHOP ON INTELLIGENT SIGNAL PROCESSING, 2001.