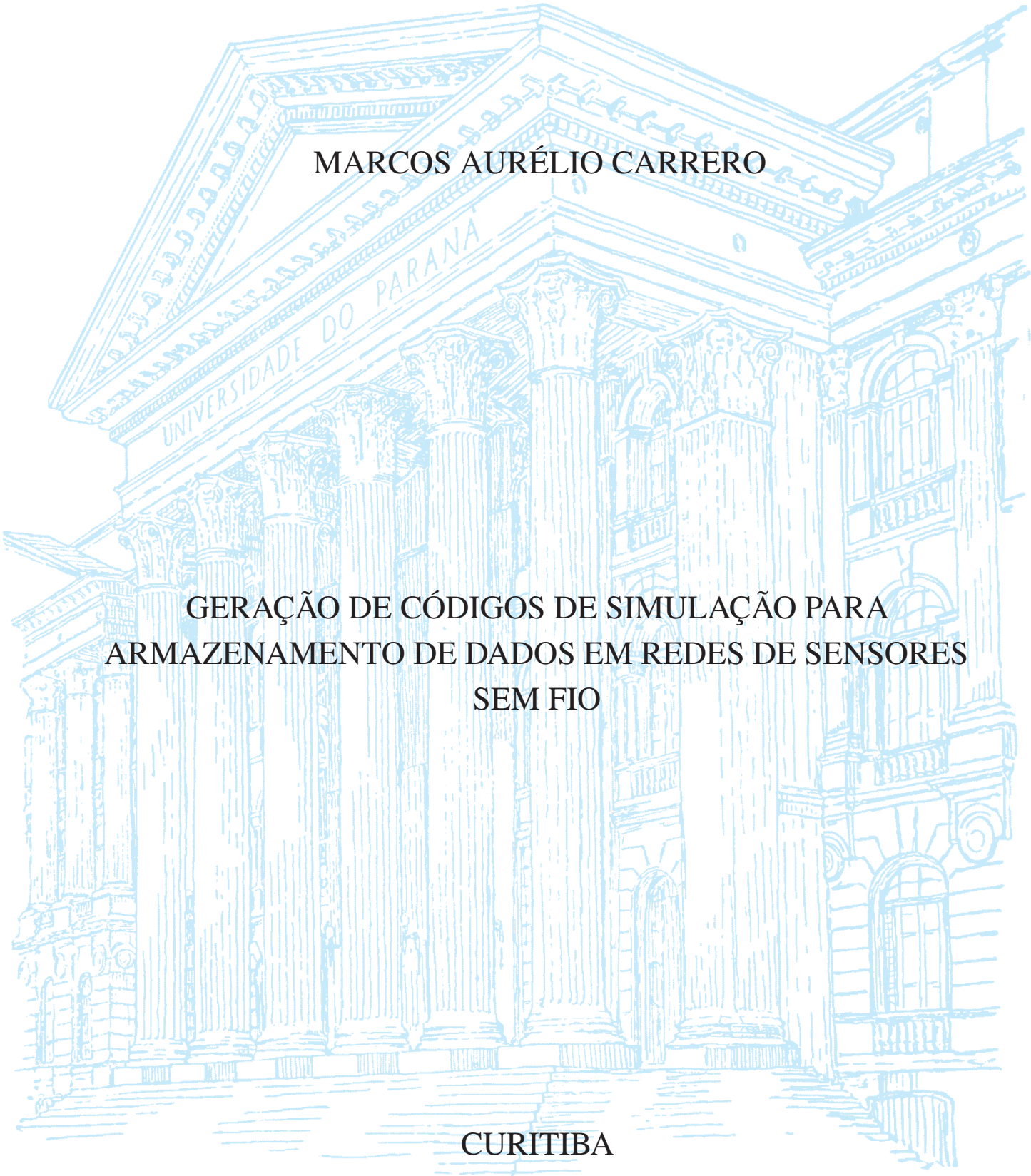
UNIVERSIDADE FEDERAL DO PARANÁ

MARCOS AURÉLIO CARRERO



GERAÇÃO DE CÓDIGOS DE SIMULAÇÃO PARA ARMAZENAMENTO DE DADOS EM REDES DE SENSORES SEM FIO

CURITIBA
2018

MARCOS AURÉLIO CARRERO

GERAÇÃO DE CÓDIGOS DE SIMULAÇÃO PARA ARMAZENAMENTO DE DADOS EM REDES DE SENSORES SEM FIO

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação, no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Profa. Dra. Carmem Satie Hara
Coorientador: Prof. Dr. Aldri Luiz dos Santos

CURITIBA
2018

FICHA CATALOGRÁFICA ELABORADA PELO SISTEMA DE BIBLIOTECAS/UFPR
BIBLIOTECA DE CIÊNCIA E TECNOLOGIA

C314g Carrero, Marcos Aurélio
Geração de códigos de simulação para armazenamento de dados em redes de sensores sem fio / Marcos Aurélio Carrero. – Curitiba, 2018.

Tese (Doutorado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2018.

Orientadora: Profa. Dra. Carmem Satie Hara.
Coorientador: Aldri Luiz dos Santos.

1. Ciência da Computação. 2. Gerenciamento de dados em sensores. 3. Reutilização de código. 4. Rede de sensores sem fio. I. Universidade Federal do Paraná. II. Hara, Carmem Satie. III. Santos, Aldri Luiz dos. IV. Título.

CDD: 004.67

Bibliotecária: Romilda Santos - CRB-9/1214

Para Dorival, Nélcia, Fábio, Fabíola, Viviani, Vivian e ao Binho (in memoriam)
Com amor e carinho...

AGRADECIMENTOS

À orientadora Profa. Carmem Hara, pela confiança e pela motivação em aprofundar os estudos na área de gestão de dados em redes de sensores. A sua dedicação, foco e paixão na condução da tese ajudou-me a superar barreiras que pareciam intransponíveis. Agradeço pelos desafios propostos com o intuito de elaborar uma tese com relevância científica e pelo envolvimento do meu orientador de mestrado Prof. Martin Musicante que participou do desenvolvimento da linguagem SLEDS. Por fim, meus sinceros agradecimentos pelo empenho e incentivo que possibilitaram minha participação em conferências e simpósios da área de pesquisa.

Ao coorientador Prof. Aldri Santos, que pelo seu profundo conhecimento em Redes de Sensores, soube explorar, aprimorar e transformar minha intuição em soluções inovadoras e criativas. Agradeço pelas incontáveis conversas, sugestões e críticas construtivas que ajudaram no amadurecimento da pesquisa. Além disso, preocupou-se em buscar colaboradores que pudessem contribuir com o trabalho e por explorar meu potencial na escrita de artigos científicos, dispondo de seu tempo para auxiliar com a submissão de trabalhos, incluindo feriados e finais de semana.

Ao Dr. Rone Ilídio, pelo tempo e esforço despendidos na implementação da parte experimental do modelo de armazenamento em sensores no simulador de redes NS2. A sua contribuição possibilitou a validação e posterior aceitação do artigo para publicação, termos. Ao Prof. Martin Musicante, meu orientador de mestrado e colaborador nesta tese do modelo de componentes e na especificação da linguagem. Agradeço pela companhia e incentivo durante a apresentação do artigo referente ao modelo de componentes. À Katriny Zamproni que contribuiu com a especificação da máquina de estados e do estudo de caso que mostraram a efetividade do modelo. Os resultados apresentados foram importantes para a definição da linguagem SLEDS.

Aos amigos Dr. Eduardo e Marjorie, por incentivarem o meu ingresso no doutorado e pelo apoio durante a elaboração da tese. Obrigado Eduardo pela amizade de longa data, pelo suporte emocional e orientação profissional. Aos amigos dos laboratório LBD e NR2, Flávio, Raqueline, Wendel, Diego, Patrick, Walmir, Robinho, Otto e Reginaldo, por proporcionarem um ambiente de descontração que ajudou-me a manter a sanidade em momentos de pressão.

Ao Programa de Pós-Graduação em Informática da Universidade Federal do Paraná, pelo auxílio administrativo que possibilitou minha participação em eventos e pela infraestrutura computacional. Agradeço também aos funcionários do departamento pelo auxílio com processos burocráticos da instituição.

Por fim agradeço à minha família. Aos queridos pais, irmãos e sobrinhas pelo amor e carinho. Meus pais sempre mostraram apoio incondicional e me motivaram durante a graduação, mestrado e doutorado. À amada Viviani que com muito amor e companheirismo manteve-me encorajado nos momentos de angústia. Seu apoio emocional e financeiro foi fundamental para que eu pudesse me dedicar ao doutorado com mais tranquilidade. À filha Vivian que com sua ternura trouxe momentos de paz e alegria. Ao primo Binho (*in memorian*) que com suas limitações físicas mostrou-me um olhar otimista da vida, sempre empolgado para ouvir as novidades sobre tecnologia.

RESUMO

Esta tese apresenta um estudo sobre as dificuldades comuns encontradas pelos desenvolvedores de sistemas de armazenamento em ambientes de simulação de redes de sensores sem fio (RSSFs). Os atuais métodos de especificação e desenvolvimento não são adequados para lidar com os requisitos de dinamicidade das redes urbanas para apoiar o desenvolvimento de uma ampla variedade de aplicativos de sensoriamento de dados. Como resultado, é preciso propor novos métodos de especificação formal para aumentar a eficiência do desenvolvimento e da avaliação de sistemas. A validação dos serviços propostos geralmente envolve programá-los em um ambiente de simulação, dados os custos e as dificuldades de implantar redes tão grandes em ambientes reais. O objetivo desta tese é propor uma abordagem formal para auxiliar a especificação de códigos de simulação de sistemas de armazenamento de RSSFs urbanas em um modelo de componentes de software. A estratégia abordada nesta tese está na definição de uma linguagem de alto nível inspirada nas máquinas de estados. A linguagem permite especificar o fluxo de execução de coordenação e interação em um modelo de componentes. A primeira contribuição consiste em uma classificação para os modelos de armazenamento de dados em RSSFs e um estudo sobre estratégias para especificação formal de códigos de simulação. A segunda contribuição consiste na implementação do AQPM, um sistema de armazenamento urbano autônomo e escalável. A proposta da classificação e a experiência com o desenvolvimento do AQPM motivaram a terceira contribuição: a identificação de entidades e funcionalidades que são comuns a vários sistemas de armazenamento. Esta abordagem é um passo importante para determinar componentes reusáveis de software. Nesse sentido foi proposto o RCBM, um metamodelo que associa entidades a um conjunto de componentes que implementam as funcionalidades comuns aos modelos. O RCBM torna possível a reusabilidade de código, facilitando o desenvolvimento de novos modelos. Para avaliar o funcionamento do RCBM, foram realizados três estudos de caso, implementando os sistemas LCA (Baker e Ephremides, 1981), LEACH (Heinzelman et al., 2000) e MAX-MIN (Amis et al., 2000). Os resultados mostram que o percentual mínimo de reutilização de código usando o RCBM foi de 66 %. A quarta contribuição desta tese é um modelo de máquina de estados para especificar a coordenação do fluxo de execução de sistemas desenvolvidos com o modelo de componentes RCBM. Através de um estudo de caso e a sua codificação no simulador NS2, mostrou-se que há uma correspondência direta entre a máquina proposta e o programa desenvolvido. Resultados mostram que a correspondência entre o conceito e o código facilita o desenvolvimento do coordenador. Por fim, a quinta contribuição desta tese propõe a linguagem SLEDS, uma linguagem inspirada em máquina de estados para geração de código do fluxo de coordenação do modelo de componentes RCBM. É proposto um esquema de tradução dirigido pela sintaxe (de SLEDS para NS2). Experimentos mostram que SLEDS promove reutilização de código e desenvolvimento ágil para a especificação de sistemas em um passo em direção à um *framework* padrão de desenvolvimento.

Palavras-chave: Gerenciamento de dados em sensores. Simulação. Reuso de código. RSSFs.

ABSTRACT

This thesis aims to study common difficulties encountered by system developers in wireless sensor networks (WSNs) simulation environments. The current specification and development methods do not allow developers to deal with the dynamicity requirements of urban sensor networks for supporting a variety of applications that demand sensing data. As a result, it is hard to improve the efficiency of system development and evaluation. Validation of the proposed models usually involves programming them in a simulation environment, given the costs and difficulties of deploying such large networks in real settings. This thesis has the purpose to provide a formal approach to assist the specification of storage systems simulation codes in a component-based development framework. The strategy addressed in this thesis is the definition of a high-level language that closely resembles a state machine. The language allows specifying the coordination execution flow and interaction in a component-based model. The first contribution consists of a data storage taxonomy model for WSNs and a study of the strategies employed to the formal specification of simulation code. The second contribution consists of the implementation of AQPM, an autonomous and scalable urban storage system. The classification proposal and the experience with the development of the AQPM motivated the third contribution: the identification of entities and functionalities that are common to several storage systems. This approach is an important step in determining reusable software components. In this sense, the RCBM was proposed, a metamodel that associates entities to a set of components that implement the functionalities common to the models. RCBM enables code reusability, facilitating the development of new models. We have conducted three case studies, implementing the LCA (Baker e Ephremides, 1981), LEACH (Heinzelman et al., 2000) and MAX-MIN(Amis et al., 2000) systems with RCBM. The results show that for these models RCBM provided at least 66% of code reuse. The fourth contribution was the proposal of a state machine model to describe data storage coordination systems on sensor networks. Through a case study and its coding in the simulator NS2, it was shown that there is a direct correspondence between the proposed machine and the developed program. Results show that the correspondence between the concept and the code facilitates the development of the coordinator. Finally, the fifth contribution of this thesis proposes the language SLEDS, a state machine-based language for RCBM coordination flow generation. This thesis presents a syntax-based translation of SLEDS to NS2. Achieved results show that SLEDS allows code reuse and agile development for system specification as a first step towards a WSN programming environment.

Keywords: Sensor data management. Simulation. Code reuse. WSNs.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

ADAGA - P*	*ADaptive AGregation Algorithm for sensor networks*
AQPM	*Autonomous Query Processing Model for Urban Networks*
CAG	*The Clustered AGgregation*
CBCWSN	*Component Based Clustering in Wireless Sensor Networks*
CBO	*Coupling Between Object classes*
CEXT	*Cluster Extend*
CFRM	*Cluster Formation Message*
CH	*Cluster Head*
CHADV	*Cluster Head Adverstisement*
CHC	*Cluster Head Candidate*
CHREQ	*Cluster Head Request*
CM	*Cluster Member*
CPS	*Cyber-Physical Systems*
DCASC	*Distributed Clustering-based Aggregation algorithm for Spatial Correlated sensor networks*
DCSSC	*Distributed Clustering Scheme based on Spatial Correlation*
DDFC	*Dynamic Data-aware Firefly Clustering*
EEFC	*Energy-Efficient Framework for Clustering-based Data Collection*
GC	Geografia do Cluster
GR	Geografia do Repositório
GHT	*Geographic Hash Table*
GW	*Gateway*
GWR	*Gateway Ready*
IBIS	*Itinerary Based dissemination for Irregular Shapes*
ICH	Líder Isolado
IoT	*Internet of Things*
LCA	*Linked Cluster Algorithm*
LEACH	*Low-Energy Adaptive Clustering Hierarchy*
LOC	*Lines of Code*
MBR	Menor Retângulo Delimitador
MDDWSN	*Model-Driven-Development-based stepwise software development process for Wireless Sensor Networks*
RCBM	*Reusable Component-Based Model for WSN Storage Simulation*
RSS	*Received Signal Strength*
RSSFs	Redes de Sensores Sem Fio
SIDS	*Spatial Index based on Data Similarity*
SLEDS	*State Machine-based Language for Event Driven Systems*
TIC	Tecnologia de Informação e Comunicação

SUMÁRIO

1 INTRODUÇÃO

A evolução das tecnologias de informação e comunicação (TIC) levou a um crescimento sem precedentes no volume de dados gerados. São exemplos de fontes de dados as redes sociais, os serviços de computação em nuvem, os dispositivos móveis e as redes de sensores sem fio (RSSFs) (Takaishi et al., 2014). O conceito de cidades inteligentes tem como objetivo prover serviços ubíquos para oferecer qualidade de vida e conforto para todos. Ela é deslumbrada em diversas áreas de aplicação como alertas de congestionamentos de tráfego Araújo et al. (2014), aplicações médicas Cremonezi et al. (2017) e casas inteligentes Filho et al. (2015). Uma questão primordial aos serviços inteligentes consiste no desafio da coleta e disseminação de grande quantidade de dados nestes ambientes físicos e dinâmicos. As redes de sensores sem fio têm oferecido essa infraestrutura para o desenvolvimento de diversos serviços para as redes urbanas em diferentes contextos de aplicação. Em virtude do ambiente complexo das cidades, os sensores são densamente espalhados por grandes áreas tais como rodovias, avenidas, parques e edifícios, monitorando e coletando dados de diferentes atributos do ambiente (Thepvilojanapong et al., 2010), contribuindo como importante fonte geradora de dados.

Nas Redes de Sensores sem Fio (RSSFs), os sensores interagem com diversos tipos de ambientes, de maneira cooperativa e autônoma, transmitindo informações coletadas do ambiente através de conexões sem fio (Rawat et al., 2014). As principais dificuldades estruturais de uma rede de sensores são a baixa capacidade de armazenamento, processamento e comunicação de curto alcance. Como existe um grande volume de dados sensoriados que estão em constante variação, interferências e barreiras do próprio ambiente podem dificultar a comunicação (Yick et al., 2008). Portanto, é preciso levar em consideração estas restrições durante o desenvolvimento de aplicações para redes urbanas que demandam o armazenamento e o processamento de consultas dos dados da rede, com foco na gestão otimizada dos recursos.

A literatura sobre os modelos de armazenamento de dados sobre RSSFs apresenta várias soluções desenvolvidas para diferentes contextos de aplicação. Abordagens clássicas de RSSFs consideram a rede vinculada a uma entidade externa, chamada de Estação Base (EB) (Rumín et al., 2010). A estação base é um dispositivo que oferece mais recursos de processamento, armazenamento e energia do que os nós sensores (Yick et al., 2008). A troca de informações entre os nós sensores e a EB ocorre através da comunicação multi-salto. Assim, a EB é muito utilizada para centralizar certas atividades, como coletar e processar dados de sensoriamento da rede (Chong et al., 2011; Rawat et al., 2014), como ponto de entrada para disseminar consultas (Tubaishat et al., 2004) e no monitoramento e controle da rede (Yang et al., 2010). No entanto, este enfoque tem como desvantagem a baixa escalabilidade da rede e alto custo de comunicação para transferência dos dados.

As abordagens de armazenamento decentralizadas, por outro lado, propõem contornar os problemas citados, de forma a manter a RSSF autônoma e garantindo escalabilidade. Essas soluções, que em geral possuem vários elementos em comum, procuram utilizar algoritmos de agrupamento da rede e outras estruturas de dados associadas. Alguns modelos definem estratégias para armazenar e consultar os dados de maneira eficiente, concentrando leituras de

um conjunto de sensores em *repositórios* de dados. Abordagens usadas em projetos de RSSFs de grande escala usam técnicas para manter a escalabilidade do sistema, como a formação de agrupamentos de sensores e a eleição de um ou mais sensores como líderes do grupo (Amaxilatis et al., 2011a). Nos agrupamentos, os sensores são reunidos de acordo com algum critério que possuam em comum. Os líderes são responsáveis por armazenar as informações de seus membros e por responder a requisições de consultas. Dessa forma, o desenvolvimento de sistemas de armazenamento para redes urbanas requer o desenvolvimento de estratégias que possam lidar com os desafios relacionados à dinamicidade de geração dos dados das cidades inteligentes.

A simulação é uma ferramenta importante para validação de sistemas de armazenamento para RSSFs urbanas antes da sua implantação. Os ambientes de simulação são mais flexíveis que as redes reais, pois oferecem o melhor compromisso entre custo e tempo gasto nas avaliações de soluções desenvolvidas em diferentes cenários de redes de grande escala (Khan et al., 2011; Horneber e Hergenröder, 2014). Embora existam várias soluções de sistemas de armazenamento propostas na literatura, poucas implementações estão disponíveis para a comunidade. Além disso, a maior parte desses algoritmos está diretamente relacionada a um sistema específico de RSSF, ou então está integrada a outra aplicação, dificultando o reuso por futuros desenvolvedores (Amaxilatis et al., 2011b).

Os modelos de componentes de software são uma boa abordagem para enfrentar esses problemas porque reduzem a complexidade do desenvolvimento e melhoram a produtividade. Modelos de software com componentes possibilitam que o desenvolvedor reutilize trechos previamente codificados, facilitando o trabalho de desenvolvimento. Por meio da identificação de padrões comuns de funcionalidades, é possível determinar componentes, que são definidos como trechos de código que se comunicam com outros componentes e oferecem um serviço determinado, utilizando uma interface bem definida. Além disso, componentes devem ser substitutíveis, independentes de contexto, encapsulados e devem constituir uma unidade independente para implantação e versionamento (Niekamp, 2005). Com um conjunto de componentes bem definido, é possível estabelecer uma biblioteca de componentes para o desenvolvimento de novos sistemas de armazenamento em ambientes de simulação, um passo importante para a construção de um *framework* que facilite a geração de códigos de simulação.

Existem diversas ferramentas de simulação e modelagem desenvolvidas para facilitar o estudo desse tipo de rede e a validação dos modelos sem que seja necessária uma implantação muitas vezes inviável no mundo real. Entretanto, a falta do suporte de uma ferramenta para o desenvolvedor especificar formalmente o comportamento do sistema em um modelo de desenvolvimento que possibilite abstrair as especificidades da rede, dificultam o projeto de sistemas. Portanto, faz-se necessário desenvolver novas ferramentas que auxiliem o desenvolver na análise e geração de códigos de simulação específicos para RSSFs urbanas.

Devido à natureza reativa das RSSFs, as máquinas de estados são frequentemente utilizadas para especificar códigos de simulação de sistemas de armazenamento de redes urbanas. Entretanto, as máquinas de estado de fluxo de controle geralmente não possuem especificações formais, mas são criadas *on-the-fly* pelo programador (Dunkels et al., 2006). Para solucionar este problema, esta tese apresenta uma linguagem baseada em máquina de estados para coordenar o fluxo de execução em um modelo de componentes de software. Na abordagem orientada a componentes, a implementação da coordenação do fluxo de execução e intermediação entre as funcionalidades oferecidas pelos componentes é uma importante tarefa para o correto funcionamento do sistema proposto. Como enfatizam Poizat et al. (2004), as linguagens de coordenação são métodos formais adequados para especificar eventos ou interações entre as entidades coordenadas.

1.1 PROBLEMA

Os desenvolvedores de aplicativos para as cidades inteligentes necessitam especificar, implementar e comparar uma variedade de modelos de armazenamento para verificar quais abordagens satisfazem aos requisitos do cenário de aplicação. A dinamicidade dos dados, a escalabilidade e a eficiência do sistema de consultas são fatores que influenciam no desempenho do acesso às estruturas de armazenamento. Para implementar e avaliar o desempenho destes modelos, as ferramentas de simulação são bastante utilizadas pois oferecem um ambiente flexível para testar modelos de armazenamento em diferentes contextos antes de sua implantação em redes reais. Experiências anteriores de desenvolvimento de aplicações de redes de sensores urbanas evidenciaram que a falta de um ambiente de programação dedicado dificulta a reutilização de códigos de simulação para construir novos modelos de armazenamento. O principal problema abordado nesta tese consiste em: *Oferecer um ambiente para elevar a produtividade do desenvolvimento de sistemas em ambientes urbanas, que auxilie a especificação, implementação e reuso de código para a construção rápida de sistemas de armazenamento em sensores*.

Para reduzir a complexidade do desenvolvimento, é comum o uso de métodos formais para especificar código destes sistemas. Os modelos de máquinas de estados e de componentes de software são métodos usados para esta finalidade. Embora existam diversas técnicas para aumentar a produtividade e acelerar o desenvolvimento de sistemas de armazenamento em sensores, elas não são flexíveis para promover a especificação do fluxo de execução de coordenação e interação entre os componentes, sendo essencial para apoiar o desenvolvimento de sistemas em ambientes de sensoriamento urbano. Logo, este problema tem motivado a investigação por novos métodos nas áreas de linguagens de coordenação, modelos de máquina de estados e de componentes de software para apoiar a geração de códigos de simulação para armazenamento de dados em sensores. Considerando o problema levantado, as seguintes questões precisam ser exploradas:

- *Quais modelos de armazenamento são eficazes para lidar com a dinamicidade de geração de dados em redes urbanas, de maneira escalável e autônoma?*
  Os modelos existentes consideram apenas a leitura de um atributo do ambiente. Portanto, estes modelos não são flexíveis para monitorar mais de um atributo do ambiente, funcionalidade que é disponibilizada em sensores reais. Além disso, muitos modelos consideram uma entidade externa para coordenar a rede, limitando a escalabilidade e autonomia destes modelos.

- *Como criar um conjunto de componentes reusáveis que possibilite criar novos modelos de armazenamento de dados em sensores?*
  A técnica de separação de conceitos está presente nos modernos processos de desenvolvimento de software. Ela é usada para definir componentes, associando-se entidades e funcionalidades que são comuns a vários sistemas. Os modelos existentes que utilizam esta técnica definem um padrão de interação entre os componentes que é difícil de estender a outros modelos de armazenamento, limitando o reuso de código.

- *Qual modelo de especificação formal possibilita especificar o fluxo de execução e interação dos componentes, abstraindo a complexidade do desenvolvimento em ambientes de simulação orientados a eventos?*
  As máquinas de estados têm sido usadas para especificar código de sistemas orientados a eventos. No entanto, os modelos de máquina de estados propostos não mostram uma correspondência direta entre a máquina e o código desenvolvido, para facilitar a sua implementação em ambientes de simulação.

- *Como combinar o modelo de máquina de estados e de componentes em um processo de desenvolvimento de sistemas de armazenamento em sensores?*
  Em uma perspectiva de plataforma de desenvolvimento, as linguagens específicas de domínio (DSLs) são comumente usadas no contexto das RSSFs. Entretanto, o foco destas linguagens não está no desenvolvimento de aplicativos para armazenamento em sensores.

1.2 OBJETIVOS

Esta tese tem por objetivo acelerar o desenvolvimento e a validação de sistemas de armazenamento de dados em sensores através de uma abordagem para especificação e reuso de códigos de simulação. Logo, a base para a especificação de código de simulação está na definição de uma linguagem específica de domínio baseada em máquina de estados para coordenar o fluxo de execução em um modelo de componentes de software. A flexibilidade da linguagem permite atender à grande demanda por sistemas inteligentes das redes de sensores urbanas.

Para atingir o objetivo proposto, os seguintes objetivos específicos foram definidos:

- Investigar os modelos e armazenamento de dados em RSSFs e propor uma taxonomia.
- Projetar um novo modelo de armazenamento com foco na dinamicidade da geração de dados de sensoriamento urbano.
- Desenvolver um modelo de componentes reusáveis, a partir da identificação de entidades e funcionalidades que são comuns a vários modelos de armazenamento.
- Investigar a possibilidade de se utilizar uma especificação formal inspirada em máquina de estados para descrever a coordenação do fluxo de execução em um modelo de componentes reusáveis.
- Elaborar a especificação de uma linguagem de geração de código de simulação para o coordenador do modelo de componentes.

1.3 CONTRIBUIÇÕES

O desenvolvimento desta tese resultou em contribuições científicas na área de computação, com ênfase em linguagens de programação e especificação formal aplicadas em sistemas de armazenamento de dados em redes de sensores. A seguir as contribuições estão descritas de forma detalhada:

- Um estudo do estado da arte da literatura. O estudo apresenta uma classificação dos modelos de armazenamento de dados em RSSFs e técnicas de especificação formal para reduzir o esforço e promover reuso de códigos de simulação. As técnicas apresentadas no estudo permitem que a sociedade se beneficie do desenvolvimento de novos serviços inteligentes, reduzindo a complexidade do desenvolvimento e melhorando a produtividade dos desenvolvedores.
- A implementação de um modelo de processamento de consultas para RSSFs urbanas (Carrero et al., 2015b,a). O modelo foi inspirado em características normalmente encontradas em aplicações de sensoriamento urbano, permitindo que os desenvolvedores implementem sistemas que atendam aos requisitos de autonomia e escalabilidade.

- A proposição de um modelo de componentes de software (Carrero et al., 2017), que utiliza entidades, propriedades e funções comuns presentes nos diversos modelos de armazenamento. Isso possibilita aos desenvolvedores estabelecer uma biblioteca de componentes para o desenvolvimento de novos sistemas de armazenamento em ambientes de simulação.

- O desenvolvimento de um modelo inspirado em máquina de estados para descrever a coordenação do fluxo de execução dos componentes (Carrero et al., 2018a). Este modelo descreve uma especificação formal do sistema e colabora para reduzir o esforço e a complexidade da geração de códigos em simuladores orientados a eventos.

- A especificação de uma linguagem inspirada em máquina de estados para coordenar o fluxo de execução em um modelo de componentes de software (Carrero et al., 2018b). A linguagem é apropriada para o desenvolvimento de sistemas de armazenamento de dados em RSSFs. A especificação atual da linguagem gera código para o ambiente de simulação NS2.

- A investigação e avaliação através de estudos de caso que mostram a efetividade do modelo de componentes para a reusabilidade de código. Além disso, estudos de caso mostram que a linguagem implementa um modelo flexível que pode ser usado em uma ampla variedade de aplicativos de armazenamento em sensores.

1.4 ESTRUTURA DA TESE

Esta tese está organizada em sete capítulos. O Capítulo 2 introduz uma revisão sobre modelos de armazenamento em RSSFs e trabalhos relevantes sobre técnicas de desenvolvimento de software a fim de promover reusabilidade de código de simulação para modelos de armazenamento em RSSFs. O Capítulo 3 descreve um modelo escalável e autônomo para o armazenamento de dados e processamento de consultas espaciais e por valor em RSSFs urbanas. O Capítulo 4 descreve o RCBM, um modelo de armazenamento para RSSFs que utiliza entidades, propriedades e funções comuns presentes nos diversos modelos de armazenamento, associando entidades a um conjunto de componentes que implementam funcionalidades comuns. O Capítulo 5 descreve a modelagem da coordenação de componentes a partir da definição de máquinas de estados, integrando esta formalização junto ao coordenador do RCBM. O Capítulo 6 apresenta a especificação de uma linguagem orientada a eventos, inspirada nas máquinas de estados para a geração de código do coordenador do modelo de componentes RCBM. Por fim, o Capítulo 7 conclui a tese, apresentando as considerações finais, os trabalhos futuros e a lista de publicações relacionados à tese.

2 FUNDAMENTOS

Este capítulo discute apresenta um estudo sobre o uso de técnicas de desenvolvimento de software para reduzir o esforço de análise, codificação e testes de códigos de simulação para modelos de armazenamento de RSSFs. A Seção 2.1 apresenta uma classificação dos sistemas de armazenamento que será usada no decorrer tese. A Seção 2.2 descreve trabalhos que utilizam o conceito de modelagem baseada em componentes para reuso de códigos de simulação. Na Seção 2.2.1, é abordado o uso de componentes aplicados ao escopo de redes de sensores sem fio. A Seção 2.3 apresenta abordagens que usam máquina de estados para especificação formal do fluxo de execução dos componentes.

2.1 MODELOS DE ARMAZENAMENTO PARA RSSFS

A gestão do armazenamento de dados em Redes de Sensores Sem Fio (RSSFs) urbanas também possui impacto significativo no uso de recursos e no desempenho do processamento de consultas (Yu et al., 2010; Kulkarni et al., 2011). Há vários trabalhos anteriores relacionados com o armazenamento de dados em RSSFs (Shen et al., 2011; Ahmed e Gregory, 2012). Os modelos de armazenamento de dados para RSSFs podem ser classificados em três categorias. O primeiro grupo compreende o armazenamento local. Os sistemas que seguem essa abordagem mantêm os dados localmente no sensor que produziu o dado. O segundo grupo compreende o armazenamento externo. Nesse modelo de armazenamento, os dados são enviados dos sensores que produziram os dados para uma estação base (EB), em formato bruto ou utilizando métodos de agregação. O terceiro grupo compreende o armazenamento centrado em dados. Este modelo distribui os dados na rede com base em um conjunto predefinido de regras ou funções aplicadas em seus valores. Entretanto, esta classificação não era genérica o suficiente para categorizar os modelos de armazenamento em sensores de grande escala, definidos nesta tese como *repositórios* de dados distribuídos na rede. Assim, foi proposto uma nova classificação para os sistemas de armazenamento. Na taxonomia ilustrada pela Figura 2.1, os modelos de armazenamento podem ser classificados como armazenamento **externo** (centralizado) ou **na rede** (distribuído). No armazenamento externo, os dados de sensoriamento são enviados para uma base de dados externa, ou estação-base (EB). No armazenamento na rede, os dados coletados são armazenados localmente no próprio sensor ou agrupados em *repositórios* de dados distribuídos na rede.

Embora a melhor escolha dependa do contexto da aplicação, nota-se que o modelo de armazenamento agrupado em *repositórios* oferece uma abordagem interessante entre o custo da consulta e o custo do armazenamento dos dados (Xie et al., 2014). Nas RSSFs de grande escala, manter todo o processamento centralizado aumenta o custo de comunicação (Can e Demirbas, 2013; Chatterjea e Havinga, 2007), sendo menos escalável que abordagens descentralizadas (Coman et al., 2007). Além disso, a ausência de um servidor central nas arquiteturas distribuídas possibilita que as tarefas de monitoramento, de controle e de processamento de consultas possam ser executadas na própria rede, requisitos desejáveis para se obter um sistema autônomo. Logo, a

eficiência no processamento de consultas baseia-se no estabelecimento de *repositórios* na rede, reduzindo-se o número de encaminhamento de consultas, isto é, de saltos na rede, para se obter o resultado desejado.

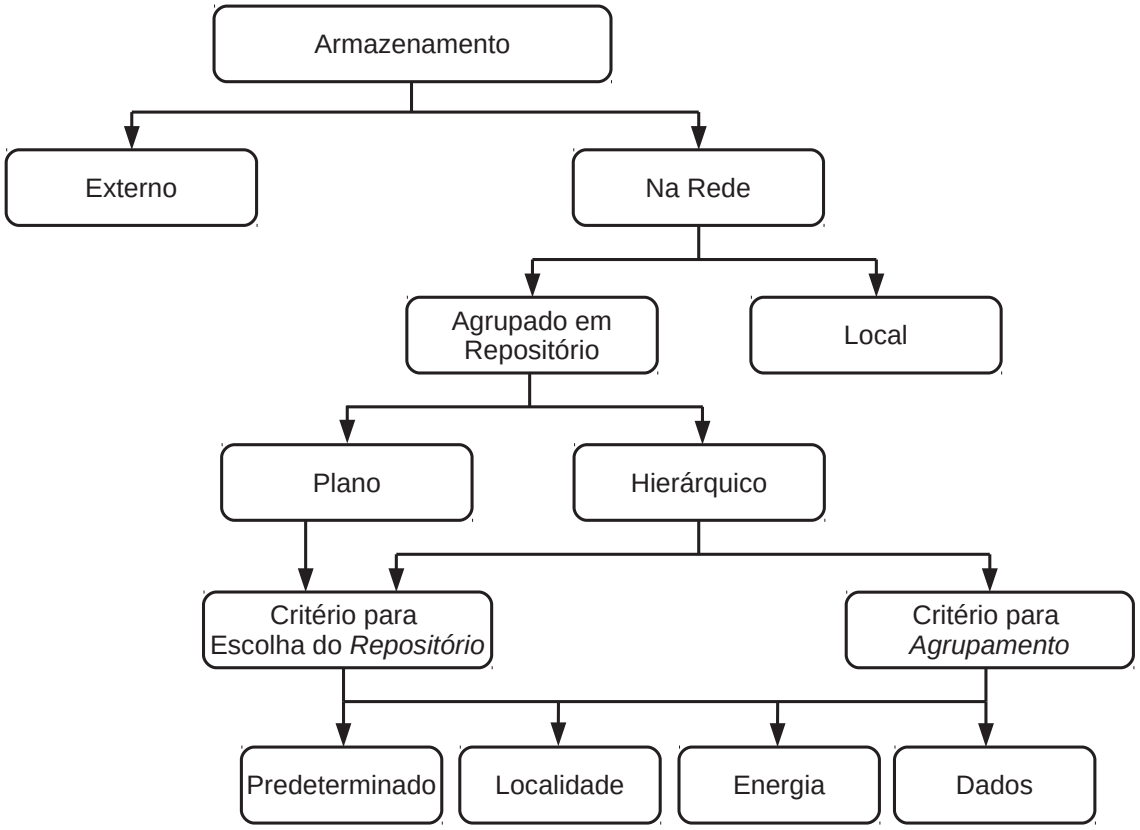


Figura 2.1: Classificação dos sistemas de armazenamento

2.1.1 Armazenamento Externo

No modelo de armazenamento externo, a estação base (EB) realiza o controle de configuração, de gerenciamento e de coleta de dados da rede (Chen et al., 2014). É possível observar na Figura 2.2 que os nós sensores, assim que realizam uma nova leitura, enviam os dados de sensoriamento para a EB para serem processados (Shenker et al., 2003; Rawat et al., 2014; Filipponi et al., 2008). O custo do processamento da consulta nesse modelo é baixo pois os dados estão disponíveis em um servidor central, possibilitando que os usuários realizem consultas diretamente na EB (Xu et al., 2015). Por outro lado, o custo de armazenamento e de comunicação é alto em virtude do grande tráfego de informações enviadas para a EB (Ahmed e Gregory, 2012). Logo, a EB pode se tornar um gargalo do sistema e um ponto único de falha do sistema (Kolcun et al., 2016), sendo uma solução ineficiente quando os dados da rede são atualizados frequentemente (Diallo et al., 2015).

Existem várias aplicações de RSSFs de pequena escala em cenários reais, descritas em Wang e Liu (2011), que utilizam o modelo de armazenamento externo. Como exemplo deste tipo de aplicação, podem ser citados: o monitoramento do ambiente de uma floresta (Tolle et al., 2005), o monitoramento do ambiente (Selavo et al., 2007; Barrenetxea et al., 2008), o monitoramento de vulcões (Werner-Allen et al., 2006; Song et al., 2009), o monitoramento da

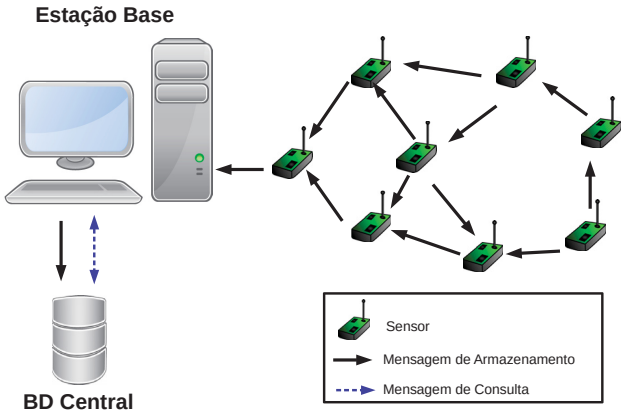
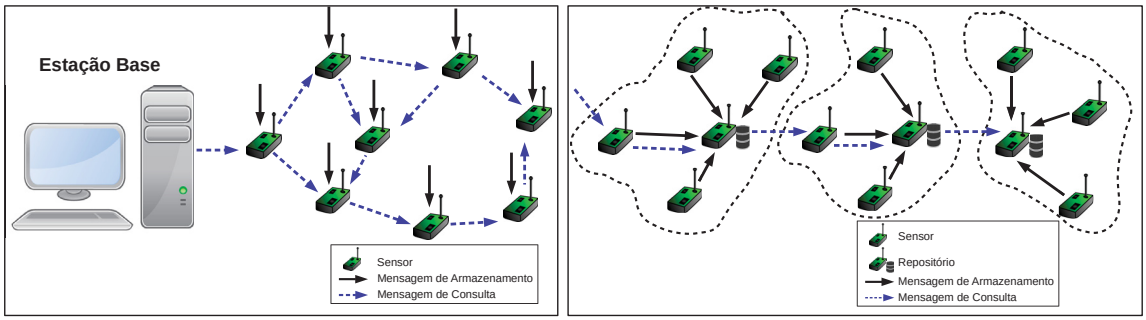


Figura 2.2: Modelo de armazenamento externo

água (Kim et al., 2008), na engenharia civil (Kim et al., 2007; Ceriotti et al., 2009) e na detecção de incêndios florestais (Hartung et al., 2006). No entanto, de acordo com Xu et al. (2015), o modelo de armazenamento externo é uma abordagem que possui várias restrições e deveria ser utilizada apenas em redes de pequena escala ou com baixa taxa de geração dos dados. Em alguns cenários, é impraticável manter a EB permanentemente conectada com a rede, enquanto que em outros cenários o armazenamento e processamento de todos os dados na EB mostra-se uma abordagem ineficiente (Albano e Chessa, 2015).

2.1.2 Armazenamento Na Rede

Os modelos de armazenamento na rede mostram-se uma alternativa mais eficiente de gerenciamento dos dados nas RSSFs, conforme ilustra a Figura 2.3. Nesta abordagem, os dados gerados pelo sensoriamento podem ser armazenados localmente no próprio sensor coletor, como visto na Figura 2.3(a) ou agrupados em *repositórios* distribuídos na rede (Subramanian et al., 2007), como ilustra a Figura 2.3(b). No armazenamento na rede, os dados de sensoriamento permanecem armazenados localmente nos sensores ou em *repositórios*, e somente serão extraídos e enviados para quem requisitou a coleta a partir da disseminação de consultas enviadas para a rede. Em alguns sistemas de armazenamento na rede, qualquer sensor pode receber uma requisição de consulta do usuário, aumentando a confiabilidade e mantendo completamente a autonomia da rede (Kolcun et al., 2016; Carrero et al., 2015a).



(a) Local (b) Agrupado em repositório

Figura 2.3: Modelos de armazenamento na rede

É possível observar na Figura 2.3(a), que a disseminação de consultas ocorre por inundação na rede no armazenamento local, pois todos os sensores devem verificar se o dado

armazenado satisfaz a condição da consulta. A disseminação de consultas por inundação favoreça alta confiabilidade e não gera sobrecarga referente ao envio de dados para a EB. Entretanto, o processamento de consultas pode ser extremamente caro porque exige a transmissão de uma solicitação de consulta para toda rede e o envio de respostas individuais de volta ao ponto de entrada da consulta, gerando um alto tráfego de mensagens duplicadas nas RSSFs de grande escala (Wu e Li, 2009; Wang e Liu, 2011). Logo, o custo do armazenamento é baixo, mas o custo de comunicação e do processamento de consultas é alto. Por outro lado, no armazenamento agrupado em *repositórios*, sensores específicos que servem como centro de dados armazenam informações de um conjunto de sensores e são os responsáveis pelo processamento das consultas. Note que na Figura 2.3(b), como apenas os *repositórios* são acessados para o processamento de consultas, o tempo de resposta da consulta e o uso de recursos da rede são minimizados.

Com relação ao armazenamento agrupado na rede, os dados podem estar distribuídos em uma estrutura de armazenamento plana ou hierárquica. A organização plana não apresenta uma relação hierárquica ou atribuição de diferentes papéis entre os sensores da rede. Dentre os modelos classificados como armazenamento plano, pode-se citar o SCOOP (Gil e Madden, 2007) e os modelos centrados em dados (DCS) (Ahmed e Gregory, 2012). No modelo SCOOP, os repositórios são escolhidos de maneira dinâmica, de acordo com a frequência da consulta e da produção de dados. Quando a frequência de consultas é maior que a de coletas, o repositório é posicionado mais próximo do sensor de requisição de consulta. Caso contrário, o *repositório* é posicionado mais próximo do sensor coletor. O modelo centrado em dados GHT (Ratnasamy et al., 2002) propõe uma estratégia de armazenamento que usa uma função *hash* para mapear atributos de sensoriamento, por exemplo temperatura, para coordenadas geográficas. Durante o processamento de consultas, o nó que recebe a requisição aplica a função *hash* no atributo da consulta, e o protocolo de roteamento geográfico faz o encaminhamento da requisição até o *repositório* que contém os dados armazenados referentes ao atributo. Além disso, utiliza um mecanismo de replicação de dados (Jr. e dos Santos, 2001; Mannes et al., 2012a,b) que permitem para garantir escalabilidade e alta disponibilidade (Lima et al., 2009) de acesso aos dados.

Na organização hierárquica, as estruturas de armazenamento são escolhidas de acordo com critérios predeterminados ou por relações lógicas dos sensores. Nos modelos de hierarquia onde o critério é predeterminado, como TSAR (Desnoyers et al., 2005), PRESTO (Li et al., 2009) e HDMST (Yang et al., 2010), o nível de *repositório*, denominado *cache*, é composto por sensores com mais recursos computacionais que os demais sensores da rede. Assim, nesta rede heterogênea um *repositório* é responsável por armazenar informações de uma dezena de sensores. Nos modelos lógicos, é comum estabelecer a relação de hierarquia a partir da formação de agrupamentos na rede, onde os sensores assumem diferentes papéis. Assim, nos modelos de armazenamento hierárquicos lógicos a rede é composta por agrupamentos de sensores, ou *clusters*, que são formados de acordo com critérios específicos determinados pelo algoritmo de formação. Dentro desses agrupamentos, pode existir a eleição de um líder ou *cluster-head* (CH), que em geral atua como unidade local de processamento de consultas relacionadas ao seu *cluster*. Aos sensores que fazem parte desses agrupamentos são atribuídos os papéis de membros, ou *cluster member* (CM). Uma descrição detalhada sobre os modelos hierárquicos lógicos encontram-se nos trabalhos de Abbasi e Younis (2007) e Afsar e Tayarani-N (2014).

Existem diversas RSSFs urbanas usadas em cenários reais que oferecem uma infraestrutura para prover serviços de interesse para os seus cidadãos. O estudo elaborado por Wen et al. (2013) busca compreender as variações espaço-temporais das concentrações de monóxido de carbono (CO) produzidas pelos fluxos de veículos, identificando potenciais riscos de exposição humana à poluição atmosférica. No projeto *CitySee* (Mao et al., 2012), o foco é o monitoramento do dióxido de carbono (CO_2), uma das principais causas relacionadas ao aquecimento global. O

trabalho UScan (Thepvilojanapong et al., 2010) propõe o uso de uma RSSF densa para coletar dados sobre a temperatura em diferentes regiões da cidade. Como observam os autores, em cenários complexos, como o ambiente urbano, se faz necessário trabalhar com uma granularidade fina no monitoramento do ambiente, pois vários fatores ambientais influenciam na variação da temperatura, tais como a presença de árvores, a largura de estradas e diferenças entre as regiões geográficas. No entanto, o foco destas soluções está em enviar dados brutos de sensoriamento para uma estação base para serem analisados posteriormente. Em cenários urbanos, esta solução pode ser inviável devido à grande quantidade de sensores e à dinamicidade da geração de dados na rede. Portanto, as técnicas de predição de dados e mecanismos que exploram a similaridade de dados descritas a seguir são mais adequadas para tratar o gerenciamento de dados em redes urbanas.

2.1.3 Sistemas de Armazenamento Agrupado

As abordagens usadas para reduzir o número de transmissões durante o processamento de consultas e, portanto, seu consumo de energia, consistem em eleger sensores para armazenar as leituras de um grupo de dispositivos e criar estruturas de indexação nos sensores denominados de *repositórios*. Exemplos de modelos que seguem esta estratégias são apresentados a seguir.

O modelo OSP

O artigo *Optimize storage placement in sensor networks* Sheng et al. (2010) mostra uma abordagem de armazenamento agrupado em *repositórios* de dados na rede. Este modelo considera que os *repositórios* são nós da rede que possuem mais recursos computacionais que os demais. Os *repositórios* concentram dados de sensores que estão próximos a ele, minimizando o custo de comunicação durante o processamento das consultas.

O objetivo principal do modelo é minimizar o consumo de energia gasto pelo armazenamento e pela consulta. Assim, o modelo determina o melhor posicionamento dos *repositórios* na rede, de tal forma a equilibrar o consumo de energia. Neste modelo, a estação base é o ponto de entrada para a disseminação de consultas. A consulta é enviada para a rede, processada pelos *repositórios*, e o resultado é enviado para a estação base.

O modelo DYSTO

O artigo *DYSTO - A Dynamic Storage Model for Wireless Sensor Networks* Gonçalves et al. (2012) é um modelo de gestão de armazenamento dos dados adaptável ao contexto do ambiente. O sistema DYSTO propõe a definição de um modelo adaptativo de armazenamento de dados na qual o sistema possa se adaptar a diferentes contextos de aplicação. O modelo utiliza um índice para o armazenamento, que mapeia um intervalo de valores a um *repositório* de dados, que pode ser um sensor ou a estação base. O índice é gerado pela Estação Base (EB) e atualizado periodicamente a partir do monitoramento do contexto da rede. Intuitivamente, DYSTO seleciona o local para armazenamento do dado mais perto de onde ele é frequentemente usado: perto da estação base quando a frequência de consultas é elevada, e próximo do sensor produtor quando a taxa de geração de dados é muito alta. Como exemplo, considere a RSSF ilustrada pela Figura 2.4, no qual cada sensor possui uma identificação e a leitura atual. Observe que existem três *repositórios* (s4 , s6 e s8), cada um armazenando dados dos sensores {s2 , s3 , s4 }, {s1 , s5 , s6 }, e {s7 , s8 }.

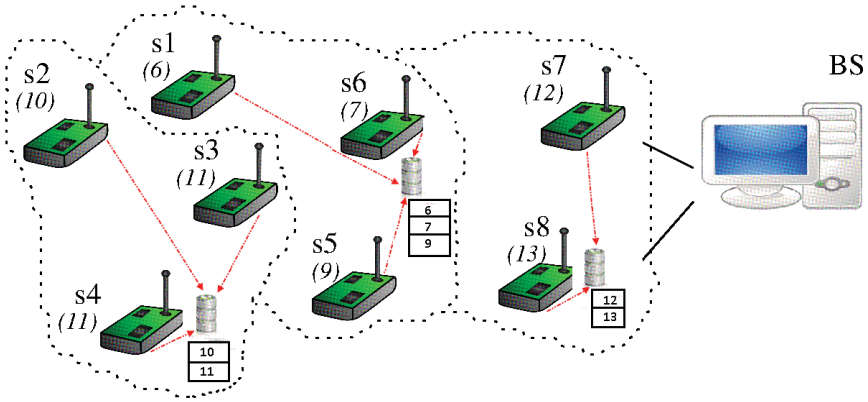


Figura 2.4: Modelo de Armazenamento do DYSTO

O sistema disponibiliza aos usuários enviar requisições de consultas através da EB. A consulta descreve um determinado intervalo de valores desejado, e o resultado é um conjunto de sensores que possuem leituras dentro deste intervalo. A EB leva em consideração informações do índice de armazenamento durante a disseminação da consulta. Assim, a consulta é enviada diretamente para os sensores repositórios responsáveis. Os repositórios realizam o processamento da consulta e enviam o resultado para a EB. Embora não esteja no escopo desta tese, uma extensão adaptativa do DYSTO (Gonçalves et al., 2014) considera o uso de políticas (Figueiredo et al., 2005; Macedo et al., 2009) para definir critérios de seleção dinâmica de *repositórios* durante o tempo útil da rede.

2.1.4 Modelos Baseados em Predição de Dados

A predição de dados consiste na construção de um modelo que descreve a evolução dos dados. O modelo pode prever os valores detectados pelos nós sensores dentro de certos limites de erro (Anastasi et al., 2009). Quando as leituras dos nós sensores podem ser previstas pelo modelo, o sensor não envia uma nova atualização para a estação base (modelo centralizado) ou para seu líder (modelo hierárquico), reduzindo os custos de comunicação durante a transmissão dos dados Chong et al. (2011). No entanto, o modelo deve ser recalculado assim que os dados reais se desviam significativamente dos dados estimados pelo modelo. A seguir são descritos alguns exemplos de abordagens que usam predição de dados.

ADAGA - P*

Motivados em reduzir o custo de comunicação em termos de consumo de energia, o trabalho *ADAGA − P** (Maia e Brayner, 2013) apresenta uma abordagem de gerenciamento de dados baseado em modelos de predição para processamento de consultas complexas. Os autores consideram que uma consulta Q é complexa se, e somente se, for necessário usar modelos de predição de dados para processá-la. Como exemplo de consulta complexa, considere a seguinte requisição de consulta na qual o usuário quer saber qual a probabilidade de chuva nos próximos dois dias na cidade de Fortaleza. Para processar a consulta, aplica-se um modelo de regressão para prever a temperatura, pressão e umidade para os próximos 2 dias. Em seguida, ele combina

esses modelos para construir um modelo capaz de estimar a probabilidade de chuva nos próximos 2 dias.

A estratégia de predição considera que a estação base é responsável por construir o modelo de regressão e que cada sensor da rede executa uma instância do modelo. O mecanismo trata as leituras consecutivas dos sensores ao longo do tempo como uma série temporal e usa a regressão linear para modelagem dos dados desta série. Além disso, o usuário pode especificar um limiar de erro durante o processamento da consulta. Como resultado, dados que podem ser previstos pelo modelo dentro de um erro limite não são transmitidos pelo sensor. No entanto, uma contribuição que pode ser realizada seria estender o modelo de predição para uma versão autônoma na qual a construção do modelo seria realizada pela própria rede, sem a necessidade de uma coordenação central.

EEFC

A fim de explorar o processamento distribuído das RSSFs, Jiang et al. (2011) propuseram uma solução que não depende de uma entidade externa para gerenciar os dados. A abordagem chamada de EEFC (***E**nergy-**E**fficient **F**ramework for **C**lustering-based Data Collection*), alia eleição de líderes para formação de agrupamentos na rede e um modelo de predição para atualização das leituras dentro dos agrupamentos. Os líderes dos agrupamentos armazenam as informações de seus membros e cada sensor monitora continuamente um atributo x associado a um valor de dados x_t em um determinado tempo t. É importante observar que, se o sensor não possuir um modelo de predição, ele irá enviar uma atualização do dado sensoriado para o seu líder sempre que ocorrer uma nova leitura. Porém, quando os sensores incorporam um modelo de predição, o sensor pode enviar de forma seletiva seus valores para o líder do grupo.

Um desafio encontrado em modelos que realizam agrupamentos na rede que consideram a similaridade de dados está em realizar a manutenção dinâmica e contínua dos grupos. Mecanismos que refazem os agrupamentos de toda a rede são muito custosos, portanto a estratégia do EEFC é realizar a manutenção do grupo localmente quando um nó sair do escopo do seu líder. O mecanismo do EEFC prevê a quebra ou fusão de agrupamentos de maneira decentralizada e que requer baixo custo de comunicação.

2.1.5 Modelos Baseados em Similaridade de Dados

Devido à alta densidade de nós presentes nas redes urbanas, é provável que as leituras dos sensores possuam correlação espaço-temporal (Villas et al., 2014; Vuran et al., 2004). Um sensor s é dito estar correlacionado com um conjunto de sensores S se as leituras de s podem ser inferidas ou calculadas a partir das leituras dos sensores de S, dentro de um limite de erro definido pelo aplicativo ou usuário (Gupta et al., 2008). A seguir são apresentados alguns trabalhos que consideram a similidade de dados presentes nas redes urbanas, que exploram esta característica para reduzir o número de transmissões e reduzir o consumo de energia na rede (Skordylis et al., 2006).

SIDS

O artigo *An efficient data acquisition model for urban sensor networks* descreve um modelo hierárquico de armazenamento de dados que leva em conta a similaridade de leituras (Furlaneto et al., 2012; Gielow et al., 2015). O sistema proposto, chamado SIDS (***S**patial **I**ndex based on **D**ata **S**imilarity*), estabelece um mecanismo de agrupamentos baseado

na similaridade espacial dos dados, um esquema para eleição de líderes e uma estrutura de indexação para evitar inundação da rede durante o processamento de consultas. O processo de agrupamento inicia-se aleatoriamente por um sensor s_i, do qual será selecionada a leitura base do agrupamento C_r. Dado um patamar de similaridade τ especificado pelo usuário, os sensores vizinhos de s_i ingressam no mesmo agrupamento se suas leituras estiverem no intervalo $[C_r \times (1 - \tau), C_r \times (1 + \tau)]$.

No SIDS, locais de concentração de informação, denominados de *repositórios* de dados são formados na rede, consistindo de pelo menos dois sensores líderes em região de borda. O mecanismo de indexação juntamente com o conceito de *repositórios* têm como objetivo processar de forma eficiente dois tipos de consultas: as espaciais e as de dados. As consultas espaciais obtêm a leitura de sensores dentro de uma região geográfica, enquanto as consultas de dados determinam os sensores que possuem leituras dentro de um intervalo de valores. A estrutura de indexação foi motivado pela presença de "ilhas de calor" existentes em cenários urbanos, ou seja, pequenas regiões dentro de áreas maiores que possuem características distintas em relação ao seu entorno. O índice é uma estrutura em árvore onde a hierarquia é definida através da correlação espacial dos agrupamentos. No entanto, o SIDS é um modelo centralizado que depende da estação base para criar os agrupamentos e a estrutura de indexação.

DDFC

Em uma abordagem baseada em modelos bio-inspirados, o DDFC (***D**ynamic **D**ata-aware **F**irefly-based **C**lustering*) estabelece de maneira autônoma a criação de agrupamentos em ambientes dinâmicos que apresentem similaridade de dados espaciais (Gielow et al., 2014). O protocolo agrupa nós com leituras semelhantes, inspirado pelos princípios biológicos dos vaga-lumes. O DDFC sincroniza agregações de leituras semelhantes em agrupamentos, possibilitando sua manutenção dinâmica e distribuída. Dessa forma, inspirado nos piscares dos vagalumes, os nós enviam periodicamente mensagens de alertas (*beacons*) para sincronizar a agregação de leituras dos nós de cada agrupamento. Essa abordagem possibilita um funcionamento adaptativo e reconfigurável dos agrupamentos.

O DDFC define uma função de similaridade para agrupamento dos sensores. Periodicamente, cada sensor s_i envia uma mensagem de *beacon* contendo sua leitura iR, sua origem src e a média das leituras de seus vizinhos aR. Considerando CTh como um limiar de similaridade definido pelo usuário, $getAvgReading()$ como uma função que calcula a média dos valores de sensoriamento do agrupamento corrente, $getReading()$ como uma função que retorna a leitura atual, um sensor s_j decide se unirá a s_i se as condições *(i)* $iR - getAvgReading() < CTh$ e *(ii)* $getReading() - aR < CTh$ forem satisfeitas. A função *(i)* verifica se a diferença entre a leitura iR recebida do nó vizinho s_i com a média de valores calculados por $getAvgReading()$ satisfaz o limiar CTh. A função *(ii)* verifica a diferença entre da leitura atual $getReading()$ com a média das leituras de seus vizinhos aR satisfaz o limiar CTh.

DCSSC

O DCSSC proposto no artigo (*Towards a Distributed Clustering Scheme based on Spatial Correlation in WSNs*) é uma abordagem que leva em consideração a formação de agrupamentos com base na similidade de um conjunto de atributos (Le et al., 2008). Nesse trabalho, os autores exploram a similaridade de dados em RSSFs de larga escala, aumentando a eficiência e prolongando a vida útil dos sensores da rede. Cada sensor comunica-se apenas com os sensores que estão à distância de um salto e todos têm os dados relativos ao nível de

energia de seus vizinhos. Esta informação é enviada por meio de mensagens do tipo *HELLO* trocadas periodicamente. O modelo de rede consiste em uma estação-base e N sensores. São atribuídos estados a todos os dispositivos, que determinam seu papel na rede. No início, os sensores estão no estado *INI* e ao final da fase de construção de agrupamentos eles estarão em um dos seguintes estados: *CH* (*cluster-head*), *GW* (*gateway*), *EXT* (*cluster-extend*) ou *MEM* (*member*). Os sensors com os estados *CH*, *GW* e *EXT* ficam nesse estado até o fim da fase de construção e são chamados de nodos de *backbone*. Existem também estados temporários, como o *INI*, *GWR* (*gateway-ready*) e *CHC* (*cluster-head candidate*).

As mensagens de formação de agrupamento incluem sempre a média de valores das leituras aferidas no tempo que precede o envio da mensagem. Baseado no tipo da mensagem, o receptor irá mudar seu estado, criar uma nova mensagem e propagar para seu vizinhos. A nova mensagem também deve conter o identificador (ID) do emissor original da mensagem recebida. A média de valores de leitura de um sensor s com n tipos de leitura é definida como o conjunto $s_1, s_2, ..., s_n$. A medida de dissimilaridade $d(s, v)$ entre dois sensores s e v é calculada da seguinte forma:

$$d(s, v) = \omega_1|s_1 - v_1| + ... + \omega_n|s_n - v_n| \tag{2.1}$$

Na Equação 2.1, o valor positivo da constante ω_i ($\sum_{i=1}^{n} \omega_i = 1$) indica o quanto o *i-ésimo* dado das leituras do sensor afeta o grau de dissimilaridade. Dois sensores s e v estão altamente correlacionados se a medida de dissimilaridade $d(s, v)$ definida pela equação é menor do que um limiar τ definido previamente pelo usuário como parâmetro, ou seja, $d(s, v) \leq \tau$.

O modelo DCSSC prioriza como representantes de agrupamentos os sensores com maior nível relativo de energia dentre os que apresentam leituras similares. Os nodos com os papéis de *CH*, *GW* e *EXT* (nodos de *backbone*) têm a função de coletar dados das leituras dos sensores, que enviam suas informações através do *backbone* a cada intervalo de tempo segundo o esquema de escalonamento *round-robin*. As leituras recebidas são armazenadas no *CH*, e podem ser comprimidas a fim de reduzir o espaço necessário de armazenamento. Os dados agregados são transmitidos do *CH* para o *GW* do respectivo *cluster*, que encaminha para um sensor vizinho que pertença a outro *cluster*. Dessa forma os dados trafegam na rede através de nodos intermediários, até chegar à estação-base, com menor custo de comunicação.

2.1.6 Discussão

A revisão dos trabalhos relacionados mostra uma ampla variedade de técnicas empregadas em diferentes domínio de aplicações, desenvolvidas de acordo com as características específicas de cada cenário. Usuários de computação científica, que necessitam analisar todos os dados coletados pela rede, usualmente fazem uso do modelo de armazenamento externo. No entanto, este enfoque tem como desvantagem a baixa escalabilidade da rede e alto custo de comunicação para transferência dos dados. Os modelos de armazenamento na rede, por outro lado, são mais eficientes no gerenciamento de dados. Assim, abrem-se novas possibilidades de desenvolvimento de diversos tipos de aplicações e serviços, como aplicações que realizam o processamento de consultas na própria rede e outras que possibilitam explorar a similaridade dos dados e a formação de agrupamentos para aumentar a escalabilidade do sistema. A proposta do AQPM, descrita no Capítulo 3, descreve um novo modelo de armazenamento de dados para redes urbanas que trata destas características.

A partir do estudo de vários modelos de armazenamento de dados e da experiência adquirida no desenvolvimento do AQPM, é possível associar algumas entidades e funcionalidades que são comuns a vários sistemas, de acordo com a Tabela 2.1.

Tabela 2.1: Conceitos e funcionalidades dos modelos de armazenamento

Conceito	**Descrição**	**Funcionalidades**
Estação-base (EB)	Dispositivo conectado à rede que possui grande poder computacional	Armazenar dados, monitoramento e processamento de consultas
Sensor	Dispositivo com baixa capacidade de processamento, armazenamento e comunicação	Coleta e envio de dados para a EB, processamento de consultas
Cluster-head (CH)	Sensor líder de um grupo	Seleção do líder do grupo, rotação do líder, processamento de consultas
Cluster member (CM)	Sensor membro de um grupo	Coleta e envio de dados para o CH, associação ao CH
Repositório	Local de armazenamento de dados na rede	Seleção de repositório, processamento de consultas

Por meio dessa identificação de padrões comuns, é possível determinar componentes, que são definidos como trechos de código que se comunicam com outros componentes e oferecem um serviço determinado, utilizando uma interface bem definida. Além disso, componentes devem ser substitutíveis, independentes de contexto, encapsulados e devem constituir uma unidade independente para implantação e versionamento (Niekamp, 2005).

Na Seção 2.2, é discutida a utilização de componentes de software, com exemplos de trabalhos que utilizam o conceito de modelagem baseada em componentes. Na Seção 2.2.1, é abordado o uso de componentes aplicados ao escopo de redes de sensores sem fio.

2.2 COMPONENTES DE SOFTWARE

O desenvolvimento baseado em componentes é um modelo de programação bem estruturado para desenvolver sistemas de software a partir de componentes existentes (Vale et al., 2016; Szyperski, 2002). A utilização de componentes de software que buscam facilitar a implementação de sistemas que possuam elementos em comum é algo recorrente em Ciência da Computação. A programação baseada em componentes fornece uma abstração de programação de alto nível através de interações entre os módulos do sistema estabelecidas por interfaces bem definidas. Essa abstração oferece a capacidade de integração e reutilização de módulos do sistema, a fim de simplificar a configuração e manutenção de software.

Devido à natureza das redes e dos sistemas distribuídos, há um grande número de tentativas bem-sucedidas de usar componentes no desenvolvimento de aplicativos distribuídos. Um trabalho de referência nesta área é apresentado por Niekamp (2005), que descreve em detalhes uma *arquitetura de componente de software*, mostrando exemplos de sua aplicação. O uso do modelo baseado em componentes geralmente está associado à existência de uma *biblioteca* que possui *templates*, contendo possivelmente código incompleto. Os *templates* implementam parcialmente determinados códigos associados a elementos ou funcionalidades. O conteúdo da biblioteca pode ser transformado em componentes, a partir da adição de código pelo usuário, como declarações de dados e comandos. Portanto, o *template* define os protótipos das funções, descrevendo seus parâmetros de entrada e os valores de saída, ou seja, a interface do componente.

As interfaces constituem as especificações técnicas dos componentes, promovendo o reuso de componentes em diferentes aplicações (Matthys et al., 2009; Canal et al., 2008). Um componente descreve os serviços que são oferecidos (suas funcionalidades) e os serviços por ele requiridos (que dependem de serviços de outros componentes). Uma interface que disponibiliza funcionalidades expõe quais são as operações realizadas pelo componente. Por outro lado, uma interface que especifica as funcionalidades requeridas, descrevem quais são as operações que o componente necessita de outros componentes. A interoperabilidade do componente depende do nível de abstração com que uma interface é descrita. A forma mais comum é a definição sintática de interfaces, também conhecida como assinatura da interface, que descreve o nome e tipos das funcionalidades. No entanto, a sintaxe das interfaces não descreve qual o resultado esperado pela execução das funcionalidades. Para tanto, a especificação semântica das interfaces descreve informações semânticas úteis sobre as execuções da funcionalidades. Como exemplo, informações sobre os valores que são aceitos como argumentos das funcionalidades e um conjunto de pré-condições e pós-condições, que definem as condições que devem ser atendidas antes e após a execução de uma funcionalidade, respectivamente.

Um exemplo de *framework* baseado em componentes é o Exhibit (Huynh et al., 2007), uma ferramenta para publicação de dados estruturados desenvolvida com a proposta de facilitar o trabalho de desenvolvimento de páginas *Web*, permitindo a criação de páginas interativas tendo apenas conhecimento básico em HTML. Uma página *Web* que utiliza o Exhibit tem duas interfaces. Uma delas é a que fica visível a todos os usuários que a acessam, e se parece com uma página *Web* comum. A segunda é a interface do autor. Nela, é preciso fazer a criação dos dados a serem utilizados pela página, e também a apresentação do conteúdo. Os dados são definidos em arquivos e são compostos por objetos que contêm pares chave-valor. Para a apresentação, é necessário criar uma página HTML simples, que inicializa as funcionalidades do Exhibit. Os códigos são pré-definidos e existem vários trechos de código já implementados que servem de base para o desenvolvimento da aplicação. É possível inserir trechos de *Cascading Style Sheets* (CSS), linguagem que descreve o estilo de um documento HTML, para sobrescrever o *default* da aplicação. O modelo de dados do Exhibit é composto por itens, que são classificados em tipos e possuem um conjunto de propriedades. A partir dessa definição, é possível estabelecer um paralelo dessa ferramenta para facilitar o desenvolvimento de aplicações *Web* que apresentam conteúdo de uma ou mais fontes de forma única (também chamadas de *Web mashups*) com o modelo de componentes, que utiliza a ideia de padronização de objetos com funcionalidades comuns.

2.2.1 Componentes para RSSFs

Estudos anteriores já identificaram a similaridade dos modelos de armazenamento para redes de sensores sem fio e propuseram soluções baseadas em componentes. Como observam Amaxilatis et al. (2011b), a maior parte desses modelos está diretamente relacionada a um sistema específico de RSSF, ou então está integrada a outra aplicação, dificultando o reuso por futuros desenvolvedores. Percebe-se então que desenvolver modelos existentes na literatura ou projetar novas soluções exigem grandes esforços de análise, projeto, desenvolvimento e codificação.

O trabalho feito por Amaxilatis et al. (2011b) faz um levantamento de alguns algoritmos existentes, analisando especificamente o aspecto de formação de *clusters* utilizada em cada um deles. Por meio desse estudo, os autores definiram três componentes: *Cluster-head decision* (responsável por definir o algoritmo de eleição de líder de acordo com os critérios utilizados pelo modelo), *Join decision* (relacionado à metodologia com a qual os nodos definem a qual *cluster* irão se associar) e *Iterator* (organiza quais nodos já se associaram a algum *cluster* e coleta

informações sobre agrupamentos já formados). O trabalho apresenta uma abordagem limitada, uma vez que forma componentes baseando-se apenas em algoritmos que utilizem formação de agrupamentos, além de determinarem um fluxo de execução pré-definido, dificultando a adaptação de modelos que não sigam o padrão previsto.

Outro trabalho relacionado foi proposto por Cheong et al. (2003), que define o *TinyGALS*, um modelo de programação para sistemas embarcados em rede. O programa é um sistema composto por módulos, os quais por sua vez são constituídos por componentes. Um componente tem variáveis internas, externas e métodos que lidam com essas variáveis, semelhante a um objeto de uma linguagem de programação orientada a objetos. Além disso, um componente possui uma interface e sua implementação. Por ser implementado no sistema *TinyOS*, o modelo mencionado acaba sendo limitado a redes de sensores em pequena escala.

O OASiS (Kushwaha et al., 2007) é outra proposta para RSSFs, que define um *framework* para redes de sensores sem fio orientadas a serviço. Nesse tipo de rede, cada atividade dos sensores é definida como um serviço, de forma similar aos serviços *Web*. Serviços são modulares, autônomos e têm interfaces pré-definidas, que permitem que eles sejam invocados na rede. Estes serviços são constituídos de componentes que fazem a comunicação e gerenciam os sensores e suas operações. Os serviços compostos resultam num grafo, que determina o fluxo de execução da aplicação. Os nodos fazem amostras periódicas do ambiente, criando objetos que traduzem logicamente um evento físico (por exemplo, para detectar a presença de fogo um sensor deve ter uma leitura superior a 100°C). Se em algum momento o protocolo de manutenção identificar mudança no status do objeto (de falso para verdadeiro ou vice-versa), é feita a transição para um novo modo, e um novo grafo de serviços é definido de acordo com a sequência de ações a serem tomadas. O OASiS também possui a limitação de tamanho de rede, devido à utilização do sistema *TinyOS*. Além disso, ele enfrenta desafios em aspectos relacionados à dinamicidade da rede.

O artigo *SensorBean: A Component Platform for Sensor-based Services* descreve um modelo de componentes orientado a serviços para o desenvolvimento de aplicações de RSSFs(Marin e Desertot, 2005). A arquitetura deste modelo está dividida em três níveis: sensores, gateways e servidores. Cada componente descreve os serviços que são oferecidos (suas funcionalidades) e os serviços por ele requiridos (que dependem de serviços de outros componentes). No nível de sensores, componentes coletores de dados enviam suas leituras para um ou mais componentes de armazenamento de dados, localizados no nível de *gateways*. Os gateways são dispositivos de maior poder computacional, e são os consumidores dos serviços de coleta de dados dos sensores da rede. Os *gateways*, por sua vez, enviam os dados da rede para os servidores de maneira reativa ou pró-ativa. A comunicação entre eles segue os protocolos padrões da computação orientada a serviços (SOAP e HTTP). O armazenamento de dados é realizado fora da rede (nos *gateways* e nos servidores), não mostrando evidências de como realizar o armazenamento na rede. Além disso, o modelo não descreve mecanismos para possibilitar a formação de agrupamentos entre os sensores e nem serviços de processamento de consultas na rede.

2.2.2 Discussão

O sistema de armazenamento AQPM, descrito no Capítulo 3, foi desenvolvido para atender a um domínio específico de aplicação, porém sem o suporte de um *framework* padrão de desenvolvimento e sem seguir uma metodologia de reuso de código. A dificuldade na implementação das simulações dos modelos foi observada por Malavolta e Muccini (2014), na qual destaca-se que os sistemas são implementados a partir do zero, para atender aos requisitos

específicos do sistema ou estão integradas em aplicações complexas, dificultando a separação de conceitos e limitando a reutilização de código (Malavolta e Muccini, 2014). Portanto, é desejável utilizar ferramentas de modelagem de sistemas que possibilite abstrair os detalhes de baixo nível, reduzindo a complexidade do desenvolvimento e melhorando a produtividade dos desenvolvedores de sistemas de simulação de RSSFs.

Para validação desses sistemas, além do estudo de novas abordagens relacionadas a redes de sensores sem fio, foram desenvolvidas diversas ferramentas de simulação e modelagem, que facilitam o estudo desse tipo de rede e a validação dos modelos sem que seja necessária uma implantação muitas vezes inviável no mundo real. Ao contrário de redes tradicionais, muitos aspectos de RSSFs ainda não foram padronizados, como tipos de arquitetura, protocolos e hardware, uma vez que existe uma ampla gama de aplicações com diversas finalidades, que requerem funções específicas (Minakov et al., 2016). Sendo assim, a falta de um *framework* de desenvolvimento genérico dificulta a implementação das simulações e praticamente impossibilita a reutilização de código.

A proposta do RCBM (*Reusable component-based model for WSN storage simulation*), que será abordada no Capítulo 4, aplica conceitos de componentes de software durante o desenvolvimento de sistemas, como as técnicas de separação de conceitos e padronização de objetos com funcionalidades comuns, possibilitando reutilizar códigos de simulação para aumentar a produtividade do desenvolvedor (Carrero et al., 2017). O RCBM provê uma alternativa aos trabalhos mencionados, pois é aplicável a redes de sensores em larga escala, por meio de simulações com a ferramenta *Network Simulator 2* (NS2) (Sundani et al., 2011). Além disso, ele possui um fluxo de execução flexível, que permite que vários tipos de modelo sejam explorados por meio do *framework*.

2.3 PROGRAMAÇÃO ORIENTADA A EVENTOS

O RCBM promove a reutilização de código de modelos de armazenamento para RSSF a partir da identificação de componentes comuns em diversos modelos propostos na literatura (Carrero et al., 2017). Nos estudos de caso são apresentados componentes para o dispositivo sensor, cluster head (representante de um agrupamento de sensores) e cluster member (membro de um agrupamento). Embora tenha sido reportado que a proposta promove reutilização de código de até 86%, a coordenação entre os componentes, que define o fluxo e intermediação entre as funcionalidades oferecidas pelos componentes, continua sendo responsabilidade do desenvolvedor. Dada a natureza das redes de computadores, a maioria dos simuladores de rede existentes, dentre eles o NS2, são baseados em eventos. Ao examinar programas desenvolvidos em NS2 para RSSF por diferentes programadores, identificou-se uma dificuldade comum: o controle do fluxo de atividades que não são disparadas por eventos, mas por uma condição lógica ou por um temporizador.

2.3.1 Máquinas de Estados Orientada a Eventos

As máquinas de estados são uma abordagem usada para tratar a complexidade do desenvolvimento orientado a eventos (Desai et al., 2013). A máquina é composta por um conjunto de estados, transições e a comunicação entre as máquinas ocorre através de eventos assíncronos. Como exemplos de eventos podem ser citados o envio de uma mensagem, um sinal de interrupção do relógio e a chamada de função de outro componente.

O trabalho proposto por Krämer et al. (2013) descreve uma biblioteca de desenvolvimento de aplicativos para RSSFs orientada a máquina de estados. Um programa pode ser decomposto

em módulos, cada um executando sua própria máquina de estados. A coordenação de vários módulos é feita pela máquina de estados do módulo principal. Embora a biblioteca promova modularidade e reusabilidade de código, a análise de desempenho da biblioteca não demonstra que sua arquitetura seja capaz de implementar sistemas escaláveis e autônomos, requisitos desejados para aplicações urbanas.

Um modelo de máquina de estados que busca garantir confiabilidade e desempenho de aplicações para RSSFs é descrito em Cecílio e Furtado (2012). No modelo proposto, existem duas classes de estados: estados que tratam de requisitos funcionais e estados que tratam de requisitos não-funcionais. Por exemplo, coletar e encaminhar dados é um requisito funcional e reencaminhar dados quando a perda de pacotes for maior que 1% é um requisito não-funcional. No entanto, o foco do modelo RCBM apresentado no Capítulo 4 está em direção oposta ao dos autores citados: enquanto o RCBM leva em consideração os requisitos funcionais encontrados em RSSFs urbanas, a proposta deles lida com os requisitos não-funcionais de ambientes industriais.

As estratégias usadas por Tokenit (Taherkordi et al., 2015) integram um ambiente de modelagem e implementação orientado à máquina de estados. O *framework* de desenvolvimento leva em conta as restrições de recursos encontradas em sistemas embarcados. O modelo define uma máquina de estados como um conjunto de atividades, que são operações executadas por um estado. As mudanças de estado ocorrem por eventos disparados por *timers* ou eventos assíncronos. A detecção de um movimento, por exemplo, é um evento assíncrono. A máquina de estados é descrita no formato XML e o compilador do *framework* gera automaticamente código para a plataforma de execução do sistema operacional Contiki (Dunkels et al., 2004). Contudo, a avaliação de desempenho não aborda aspectos sobre a escalabilidade do sistema, característica desejada em aplicações de sensoriamento urbano.

2.4 RESUMO

Experiências anteriores de desenvolvimento de código de simulação, como em Furlaneto et al. (2012) e Carrero et al. (2015a), mostram que a ausência de noções claras sobre transições que não são controladas por eventos, como as transições lógica e por temporização, dificultam o desenvolvimento de programas. O objetivo da máquina de estados proposta nesta tese é oferecer suporte para o desenvolvedor especificar o comportamento do coordenador antes de iniciar a implementação do sistema no ambiente de simulação orientado a eventos desejado.

O Capítulo 5 apresenta um modelo de máquina de estados para fazer o detalhamento do fluxo de operações em um simulador baseado em eventos, criando 2 tipos de transição: baseado em evento e baseado em lógica. A máquina de estados foi utilizada em um estudo de caso para especificar o código do fluxo de execução de um sistema de armazenamento em redes urbanas, coordenando as interações entre os componentes do sistema. Os resultados mostram que há uma correspondência direta entre a especificação da máquina e o código de simulação.

O Capítulo 6 apresenta a especificação de uma linguagem para prototipação de aplicações de sensoriamento urbano. A linguagem foi inspirada nos modelos de máquinas de estados discutidas no Capítulo 5. Experimentos mostram a viabilidade de tradução de código para o simulador NS2.

3 UM MODELO AUTÔNOMO DE PROCESSAMENTO DE CONSULTAS PARA REDES DE SENSORES URBANAS

Este capítulo descreve o AQPM, um modelo escalável e autônomo para o armazenamento de dados e processamento de consultas espaciais e por valor em redes de sensores sem fio. A escalabilidade do modelo resulta da estratégia de formação de agrupamentos de sensores que possuem similaridade espaço-temporal de leituras, bem como do conceito de *repositórios*. Os *repositórios* concentram dados de monitoramento de um conjunto de agrupamentos e são os responsáveis pelo processamento das consultas. Como apenas os *repositórios* são acessados para o processamento de consultas, o tempo de resposta da consulta e o uso de recursos da rede são minimizados. Diferente das soluções existentes, nas quais os agrupamentos por similaridade são determinados por uma entidade externa à rede e de forma centralizada, o AQPM é um modelo autônomo. Portanto, a própria RSSF realiza a formação de agrupamentos, a escolha de seus representantes bem como da localidade dos *repositórios*. O modelo proposto foi implementado no simulador NS2 e resultados experimentais mostram que o seu processamento de consulta é mais eficaz do que as abordagens clássicas e mantém aproximadamente a precisão dos resultados devido a abordagem adotada de agrupamento por similaridade.

3.1 INTRODUÇÃO E MOTIVAÇÃO

Em ambientes urbanos, os sensores podem ser densamente distribuídos a fim de coletar diferentes atributos ambientais tais como temperatura, pressão, umidade, luminosidade e poluição. Além disso, os dados coletados podem ser armazenados nos próprios sensores, proporcionando o desenvolvimento de aplicações que executam consultas na rede, sem depender de um servidor central. Em particular, em RSSFs de grande escala, manter todo o processamento centralizado aumenta o custo de comunicação (Can e Demirbas, 2013), sendo menos escalável que abordagens descentralizadas (Coman et al., 2007).

As consultas processadas por uma RSSF podem ser classificadas em consultas espaciais e por valor. As consultas espaciais têm como objetivo obter os valores coletados por sensores dentro de uma área geográfica de interesse. As consultas por valor, por outro lado, determinam quais sensores coletaram dados dentro de um intervalo de valores. A disseminação de consultas é uma tarefa difícil devido ao alto custo de comunicação e de restrições dos recursos da rede. No entanto, explorando algumas características encontradas nas RSSFs é possível reduzir a sobrecarga na comunicação (Le et al., 2008; Cheng et al., 2011). Em algumas aplicações, a correlação espaço-temporal presente nas leituras de sensores próximos possibilita reduzir o

número de transmissões na rede. No caso da correlação espacial, leituras de sensores próximos tendem a ser muito similares nos seus valores. Em relação à segunda característica, leituras consecutivas tendem a ser muito próximas no tempo (Vuran et al., 2004). Explorando-se a correlação espacial, que é o foco deste trabalho, é possível alcançar um requisito muito desejável que é a escalabilidade, organizando em grupos (*clusters*) sensores que apresentarem leituras similares em seus valores (Hung et al., 2012).

Além da característica espacial, a gestão do armazenamento de dados em RSSFs possui impacto significativo no uso de recursos e no desempenho do processamento de consultas (Yu et al., 2010). Conforme mencionado no Capítulo 2, os dados coletados podem ser armazenados localmente no próprio sensor, em uma base de dados externa, ou em *repositórios* de dados distribuídos na própria rede. Embora a melhor escolha dependa do contexto da aplicação, nota-se que o modelo de armazenamento de dados em *repositórios* oferece uma abordagem interessante entre o custo da consulta e o custo do armazenamento dos dados (Xie et al., 2014). Logo, a eficiência no processamento de consultas baseia-se no estabelecimento de *repositórios* na rede, reduzindo-se o número de encaminhamento de consultas, isto é, de saltos na rede, para se obter o resultado desejado.

Inspirados por estes desafios, trabalhos recentes que lidam com o agrupamento de sensores com base na similaridade de dados têm sido propostos. A maioria dos algoritmos de agrupamentos atuais, como SIDS (Furlaneto et al., 2012), consideram a tolerância de erro da leitura de apenas um atributo do ambiente. Por outro lado, poucas soluções consideram a correlação de similaridade de leituras de dados multidimensionais, como DCSSC (Le et al., 2008) e DCASC (Ma et al., 2011). Nestes trabalhos, sensores com alta correlação em suas leituras são agrupados e um sensor líder é eleito como representante do grupo. DCSSC e DCASC dependem apenas de CHs para responder solicitações de consulta. No entanto, resultados mostram que estes sistemas podem gerar uma grande quantidade de líderes, de modo que o processamento da consulta em uma rede de grande escala pode ser cara. Outra limitação do DCSSC é que as consultas são iniciadas a partir de um ponto de entrada externo. Por outro lado, o SIDS combina a formação de agrupamentos com base na similaridade espacial dos dados com a criação de *repositórios*, porém não é um modelo autônomo pois depende de uma entidade externa para coordenar o processo de agrupamento.

Para tratar as limitações citadas nos trabalhos anteriores, foi proposto um modelo autônomo de processamento de consultas para RSSFs urbanas, chamado AQPM (*Autonomous Query Processing Model for Urban Networks*). O AQPM é um modelo hierárquico e distribuído onde cada sensor armazena localmente os dados de monitoramento, e sensores com alta correlação espacial são organizados em grupos. No nível de agrupamento, um sensor é eleito para ser o representante do grupo e sua leitura sensoriada representa a leitura do seu agrupamento. No nível de *repositório*, sensores específicos que servem como centro de dados armazenam informações de um conjunto de agrupamentos distintos, reduzindo o custo total de comunicação no processamento de consultas. O AQPM tem por objetivo processar tanto consultas espaciais, para obter leituras em uma determinada localização geográfica, quanto consultas baseadas em valor, para recuperar a localização dos sensores com leituras dentro de um determinado intervalo de valores. O modelo é autônomo porque a própria rede atua na formação de agrupamentos, na escolha do representante do grupo e na definição de *repositórios*, sem depender de uma entidade externa central. Em nosso entendimento, este é o primeiro modelo **autônomo** que combina similaridade espacial de leituras com armazenamento distribuído em *repositórios* para reduzir a comunicação na rede no processamento de consultas no âmbito de RSSFs urbanas. Simulações mostram que o modelo reduz consideravelmente o tempo de processamento das consultas, reduzindo o número de transmissões na rede.

3.2 TRABALHOS RELACIONADOS

Diferentes abordagens têm sido propostas para tratar o armazenamento de dados, a construção de agrupamentos e o processamento de consultas em RSSFs. O gerenciamento dos dados na rede possibilita o desenvolvimento de aplicativos eficientes de processamento de consultas. Os agrupamentos são conjuntos de sensores, que são reunidos de acordo com algum critério que possuam em comum. Em geral, são eleitos representantes para cada agrupamento. Para obter as leituras dos sensores, ou enviar mensagens para cada dispositivo individualmente, é possível reduzir a comunicação à troca de mensagens com os representantes, diminuindo os custos com comunicação e consequentemente auxiliando na economia da bateria dos dispositivos e aumentando o desempenho do monitoramento.

Nas redes que apresentam correlação no dado sensoriado, os sensores podem ser agrupados com base na similaridade de suas leituras. Contudo, investigando a revisão literária sobre requisição de dados espaciais apresentada em Da Silva et al. (2014), verificou-se que nenhum trabalho utiliza agrupamentos por similaridade de dados na rede a fim de melhorar o desempenho do processamento de consultas.

O Scoop (Gil e Madden, 2007) e o GHT (Ratnasamy et al., 2002) são sistemas de armazenamento em sensores que seguem as abordagens baseadas em *repositórios* de dados. O Scoop usa um sistema de indexação que mapeia intervalos de valores para os dispositivos sensores selecionados como *repositórios*. No entanto, ao contrário do AQPM, o Scoop não é escalável e requer uma estação base para operar a rede, isto é, não é um sistema autônomo. O GHT, por outro lado, usa uma função de espalhamento (função hash) para mapear nomes de atributos para uma localização geográfica. Portanto, é provável que as consultas e atualizações de dados possam ser encaminhadas para sensores localizados longe dos dispositivos coletores, gerando um alto custo de comunicação no processamento dessas operações. Além disso, nenhum deles oferece suporte às consultas espaciais e baseadas em valor.

Estratégias usadas por CAG (Yoon e Shahabi, 2007), DCSSC (Le et al., 2008) e DCASC (Ma et al., 2011) descrevem mecanismos que lidam com a característica da similaridade de dados sensoriados. Em geral, nestes trabalhos os sensores são agrupados e há seleção de um líder como representante do grupo, de modo que os sistemas sejam escaláveis. Contudo, nenhumas destas abordagens são totalmente autônomas e eficientes no processamento de consultas espaciais. O CAG precisa constantemente reconstruir os agrupamentos da rede para estabelecer novas consultas. O DCSSC, por outro lado, depende da estação base para gerenciar a criação dos agrupamentos. O DCASC apresenta um algoritmo para construção de agrupamentos de dados sensoriados, no qual este trabalho se inspirou, porém o DCASC não oferece suporte para consultas espaciais.

Um modelo de armazenamento de dados que leva em conta a similaridade de leituras é descrita por SIDS (Furlaneto et al., 2012). Ele estabelece um mecanismo de agrupamentos baseados na similaridade espacial dos dados, um esquema para eleição de líderes e uma estrutura de indexação para evitar inundações. Além disso, locais de concentração de informação, denominados de *repositórios* de dados são formados na rede, consistindo de pelo menos dois sensores líderes em região de borda. No entanto, o SIDS é um modelo centralizado que depende da estação base para criar os agrupamentos.

Entre os algoritmos de consultas espaciais, o IBIS (da Silva et al., 2011) descreve uma proposta para RSSFs eficiente na redução do consumo de energia da rede. O IBIS é um mecanismo para processamento de requisições espaciais irregulares, baseado na criação de itinerários. Durante a requisição de uma consulta espacial, ele cria um itinerário para encaminhar e agregar os dados sensoriados da região de interesse. No entanto, o IBIS não leva em conta

a similaridade de dados e a formação de agrupamentos de sensores. Assim, um modelo de processamento de consultas espaciais que atenda os requisitos encontrados em RSSFs urbanas se faz necessário. Tal modelo deve ser capaz de combinar estratégias de agrupamentos por similaridade de dados e de formação de *repositórios*, adaptando-as para dar suporte a uma rede com alta densidade dos sensores, provendo escalabilidade e mantendo completamente a autonomia da rede.

3.3 O MODELO AQPM

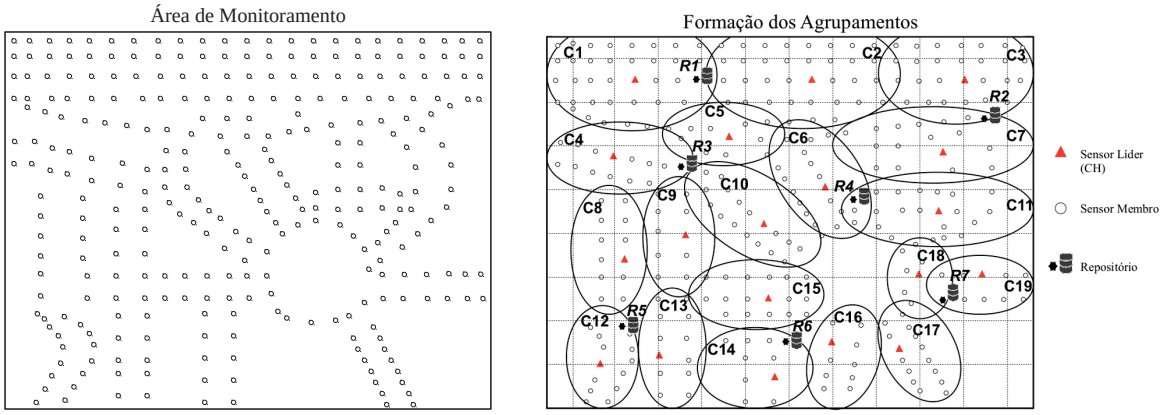
Esta seção descreve um modelo hierárquico e distribuído para realizar o processamento de consultas espaciais eficientes em redes que apresentam similaridade espacial nas leituras dos sensores. Este modelo, chamado AQPM (*Autonomous Query Processing Model for Urban Networks*), organiza em grupos sensores próximos que possuam similaridade em suas leituras. Uma vez definidos os agrupamentos, inicia-se o processo de seleção de sensores denominados de *repositórios*, responsáveis por armazenar informações de um conjunto de agrupamentos distintos. A seguir são detalhados o modelo de rede, os algoritmos de correlação espacial, de formação agrupamentos e de escolha de *repositórios*, bem como o modo de processamento das consultas espaciais.

3.3.1 Modelo de Rede

Uma RSSF é representada como um grafo $G = (V, L)$, onde $V = \{s_1, \ldots, s_n\}$ é um conjunto de sensores dispersos sobre a área monitorada M e L é o conjunto de ligações entre pares de sensores tal que $(s_i, s_j) \in L$ se s_i e s_j estão dentro do raio de comunicação um do outro. Diz-se que a distância entre s_i e s_j é de um *salto* e que s_i e s_j são *vizinhos*. A comunicação entre dois sensores quaisquer requer a existência de um caminho de roteamento $R = \{(s_1, s_2), (s_2, s_3), \ldots (s_{n-1}, s_n)\}$ tal que s_1 é o sensor que origina a mensagem e s_n é o seu destino final. Portanto, a comunicação em RSSFs depende da cooperação do repasse de mensagens na rede, e a escolha da rota R é tarefa do protocolo de roteamento. Neste trabalho, é assumido que os sensores são estáticos e, portanto, possuem um posicionamento geográfico fixo. Assume-se também que cada sensor realiza o sensoriamento de mais de um atributo do ambiente. As leituras de um sensor s são representadas por uma tupla $X = (x_1, x_2, \ldots, x_n)$, na qual cada x_i corresponde a um tipo de atributo de dado sensoriado como temperatura, umidade, luminosidade e poluição atmosférica.

Como exemplo, considere uma RSSF urbana monitorando diferentes valores de atributos ambientais, tais como temperatura, umidade, luminosidade e poluição do ar (CO, CO_2). A Figura 3.1(a) mostra os sensores densamente espalhados sobre algumas regiões da cidade, como parques, jardins, ruas e avenidas. Em cenários complexos, como o cenário urbano, se faz necessário trabalhar com uma granularidade fina no monitoramento do ambiente (Thepvilojanapong et al., 2010). O estudo mostrou que vários fatores ambientais influenciam na variação da temperatura, tais como a presença de árvores, a largura de estradas e diferenças entre as regiões geográficas. Além disso, devido à topografia irregular da cidade, verificam-se diferentes índices de concentração de poluentes mesmo em lados opostos da rua (Resch et al., 2009). Assim, justifica-se o uso de uma RSSF densa como forma adequada de monitorar tal ambiente complexo (Muller et al., 2013) .

Neste trabalho, sensores próximos são agrupados com base na similaridade espacial dos dados, como ilustra a Figura 3.1(b). Cada agrupamento possui um sensor líder, que é responsável pelo armazenamento de informações relevantes sobre os demais sensores do grupo.



(a) (b)

Figura 3.1: Visão geral do cenário e do modelo AQPM

Os agrupamentos são definidos em áreas geográficas contíguas. Os dados amostrados dentro de cada agrupamento apresentam alta correlação entre os seus membros. Portanto, com base no entendimento acima, apenas os dados coletados pelo líder são relevantes durante o processamento de consultas, não necessitando das informações coletadas pelos demais membros. Assim, a sobrecarga no processamento e a comunicação dentro do agrupamento diminuem, resultando em menos uso de recursos.

A seleção do líder foi inspirada pelo método de correlação espacial apresentado em Ma et al. (2011). Lembre-se que nós sensores monitoram uma série de informações sobre o ambiente, onde $X = (x_1, x_2, \ldots, x_n)$ denota as leituras do sensor s_i, $Y = (y_1, y_2, \ldots, y_n)$ denota as leituras do sensor s_j e $N(i)$ denota o conjunto de vizinhos que estão a um *salto* de distância do sensor s_i. Portanto, o líder selecionado será aquele que possuir alta correlação entre suas leituras e as leituras de seus sensores vizinhos. O processo da seleção do líder se faz em várias etapas. Na primeira equação, a distância euclidiana entre as leituras do sensor s_i em relação ao sensor s_j é calculada como

$$d_{ij} = \sqrt{|x_1 - y_1|^2 + |x_2 - y_2|^2 + \ldots + |x_n - y_n|^2} \tag{3.1}$$

Então, a média das distâncias $\overline{d_i}$, entre s_i e seus vizinhos, é dada por

$$\overline{d_i} = \frac{1}{|N(i)|} \sum_{j \in N(i)} d_{ij} \tag{3.2}$$

Além disso, o desvio médio absoluto de d_i é dado por

$$D(d_i) = \frac{1}{|N(i)|} \sum_{j \in N(i)} (d_{ij} - \overline{d_i})^2 \tag{3.3}$$

Portanto, o peso da correlação espacial $w(s_i)$ $(0 \leq w(s_i) \leq 1)$ de s_i é dado por

$$w(s_i) = \frac{\left[\sum_{j \in N(i)} |d_{ij} - \overline{d_i}|\right]^2}{|N(i)|^2 D(d_{ij})} = \frac{\left[\sum_{j \in N(i)} |d_{ij} - \overline{d_i}|\right]^2}{|N(i)| \sum_{j \in N(i)} (d_{ij} - \overline{d_i})^2} \tag{3.4}$$

De acordo com a equação acima, cada sensor s_i calcula um peso $w(s_i)$ que indica o quão correlacionados estão suas leituras com relação às leituras de seus vizinhos. Grandes valores de $w(s_i)$ indicam que as leituras de s_i e as leituras de seus vizinhos tendem a ser altamente correlacionadas. De fato, um sensor s_i será selecionado como líder com base em um limite τ definido pelo usuário. Assim, a seleção do líder ocorre se $w(s_i) \geq \tau$. Intuitivamente, em uma vizinhança, o nó com maior peso será eleito como líder de um grupo de sensores que possuem similaridade numérica de leituras.

No modelo AQPM existem três níveis de componentes: o sensor coletor s, o agrupamento h ao qual s pertence e o *repositório* r no qual os dados do agrupamento h são armazenados. No nível inferior, cada sensor s_i deve armazenar sua leitura atual, o peso da correlação $w(s_i)$, as leituras de seus vizinhos $readingsOfNeighbors(s_i)$, uma lista de seus vizinhos $N(i)$ e o líder do agrupamento ao qual ele pertence $CH(s_i)$. Assume-se que o mecanismo de disseminação de consulta depende de um protocolo de roteamento geográfico, de modo que cada sensor tem que armazenar sua própria informação geográfica $position(s_i)$. No nível superior, formado pelos agrupamentos e seus líderes, um sensor líder h armazena sua leitura atual, a posição geográfica dos membros do seu agrupamento $GC(h)$ (Geografia do Cluster) e o *repositório* no qual seus dados são armazenados $R(h)$. Observe que a partir do conjunto GC é possível determinar o menor retângulo delimitador que contém todos os sensores de um agrupamento $MBR(h)$. No nível mais alto, definido pelos *repositórios*, um sensor r estabelecido como *repositório* armazena as leituras de um conjunto de líderes, informações sobre as posições geográficas dos seus sensores GR (Geografia do Repositório) e uma lista dos *repositórios* mais próximos $knownRepo(r)$. É importante observar que um mesmo sensor pode assumir diversos papéis (*role*) simultaneamente, ou seja, além de um membro de um agrupamento, ele pode ser um líder (CH), um líder isolado (ICH) e/ou um *repositório* (R). Um líder é aquele que possui membros associados a ele. Por outro lado, um líder isolado não possui membros associados. O Algoritmo 1 mostra de forma detalhada o cálculo do peso da correlação espacial e da formação de agrupamentos no AQPM.

Inicialmente, cada sensor envia a mensagem "SEND_READINGS" por inundação na rede, informando sua leitura atual *(l.9)*. Cada sensor s_i que recebe a mensagem armazena as leituras de seus vizinhos *(l.16-17)*. Após certo período de tempo Γ, cada sensor s_i executa a função "CALCULATE_WEIGHT" *(l.10-11)* para calcular o peso de sua correlação espacial $w(s_i)$, informando qual o limite (*threshold*) definido pelo usuário. A seleção do líder (CH) ocorre quando o peso $w(s_i)$, calculado pela função $getWeight(s_i)$, for maior do que o *threshold* informado. Em seguida, o CH eleito anuncia esta decisão para os demais sensores *(l.20-23)*. Ao receberem mensagens de anúncio, os sensores armazenam os possíveis candidatos a CH ($candidateCHs$) *(l.34)*. Se $w(s_i)$ for menor que o *threshold* informado, o sensor vira membro e espera por anúncios dos líderes *(l.24-25)*. Em seguida, o sensor membro escolhe como CH, dentre os sensores armazenados como possíveis candidatos ($candidateCHs$), aquele que tiver a leitura mais parecida com a sua, e então envia uma mensagem de anúncio para o CH eleito como seu líder *(l.26-27)*. O CH ao receber o ACK de associação, armazena a posição dos sensores membros $GC(s_i)$ *(l.29-14)*, possibilitando ao líder calcular qual a MBR do agrupamento. Por

Algoritmo 1 Peso da Correlação Espacial e Agrupamento

```
1:  procedure GROUPING
2:      CH(s_i) ← s_i
3:      role[CH](s_i) ← FALSE
4:      role[R](s_i) ← FALSE
5:      readingsOfNeighbors(s_i) ← {}
6:      candidateCHs(s_i) ← {}
7:      GC(s_i) ← {}
8:
9:      broadcast('SEND_READINGS', currentReadings())
10:     WAIT(Γ time units)   // Γ < Δ
11:     CALCULATE_WEIGHT(Threshold)
12:     WAIT(Δ time units)
13:     if (GC(s_i) = {}) then
14:         role[ICH](s_i) ← TRUE   // Cluster-head alone
15:
16: procedure RECEIVING('SEND_READINGS', NeighborReadings) by s_i
17:     readingsOfNeighbors(s_i) ← readingsOfNeighbors(s_i) ∪ {neighborReadings}
18:
19: procedure CALCULATE_WEIGHT(Thresold) by s_i
20:     w(s_i) ← getWeight(readingsOfNeighbors(s_i))
21:     if (w(s_i) ≥ threshold) then
22:         role[CH](s_i) ← TRUE
23:         broadcast('CH_ANNOUNCEMENT', s_i)
24:     else
25:         WAIT(Γ time units) for CH_ANNOUNCEMENT
26:         CH(s_i) ← getCHWithSimilarReadings(candidateCHs(s_i))
27:         send ACK('CH_ANNOUNCEMENT', CH(s_i), position(s_i))
28:
29: procedure RECEIVING('ACK_CH_ANNOUNCEMENT', h, pos) by s_i
30:     if role[CH](s_i) = TRUE then
31:         GC(s_i) ← GC(s_i) ∪ {pos}
32:
33: procedure RECEIVING('CH_ANNOUNCEMENT', h) by s_i
34:     candidateCHs(s_i) ← candidateCHs(s_i) ∪ {h}
```

fim, o CH que não recebeu nenhuma mensagem de associação de um membro, tem seu papel alterado para líder isolado (ICH) *(l.12-14)*.

Como exemplo, considere a rede de sensores ilustrada na Figura 3.2, onde cada sensor está representado com seu identificador e seu conjunto de leituras. Na fase de eleição do líder, cada sensor s_i calcula o peso da correlação espacial $w(s_i)$, onde $0 <= w(s_i) <= 1$. Quanto mais próximo de 1, maior a correlação entre s_i e seus vizinhos. Dado que o limiar para eleição de líderes para o exemplo foi 0.85, então os sensores anuncia-se como líder se $w(s_i) >= 0.85$. Logo, s_4 e s_5 elegem-se como líderes. Em seguida, cada membro associa-se ao líder mais próximo que possui a maior correlação de suas leituras. Note que $s2$ associou-se ao líder s_4 e não ao líder s_5.

Dado que os sensores formam agrupamentos com base em sua localização espacial e similaridade em suas leituras, resta definir a estrutura do nível de *repositórios*. O *repositório*

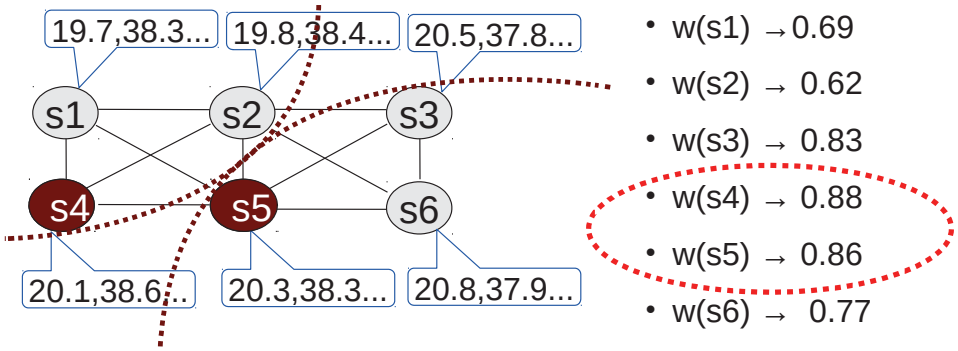


Figura 3.2: Peso da correlação espacial e agrupamento

tem por finalidade minimizar o número de *saltos* durante o processamento de consultas através da agregação de informações de líderes que estão próximos. Portanto, eles são os responsáveis por armazenar as leituras de um grupo de líderes tal que somente os dados dos *repositórios* serão utilizados durante o processamento de consultas espaciais. Intuitivamente, visto que os líderes representam dados de sensoriamento de cada membro do agrupamento, um *repositório* pode atuar como um centro de dados para responder a consultas referentes a quaisquer um dos agrupamentos que o compõem. O Algoritmo 2 apresenta a estratégia de escolha do *repositório*.

Inicialmente, a função *NeighboursInDistinctClusters* (s_i) *(l.3)* calcula o número de vizinhos de s_i que pertencem a agrupamentos distintos. O objetivo é procurar sensores que se encontram em regiões de borda para escolher como *repositório* aqueles que podem concentrar o maior número de agrupamentos. Assim, a seleção do sensor *repositório* ocorre se o número de agrupamentos em sua vizinhança for maior que um determinado patamar (*threshold*) definido pelo usuário *(l.4)*. Em seguida, o sensor escolhido como *repositório* anuncia esta decisão para os demais sensores *(l.5)*. Os sensores que recebem os anúncios dos *repositórios* armazenam cada *repositório* anunciado em um conjunto *knownRepo (l.16)*. Além disso, as mensagens de anúncio somente serão retransmitidas se a distância de s_i para o sensor r que enviou o anúncio for menor que a menor distância de algum *repositório* já conhecido *(l.17-18)*. Após aguardar certo período Δ de tempo, o *repositório* se anuncia para os seus *repositórios* membros (*knownRepo*) *(l.6-8)*. O *repositório* membro que recebe a mensagem atualiza seu conjunto de *repositórios* conhecidos *(l.27-28)*. Mantendo um conjunto de *repositórios* conhecidos, um *repositório* consegue calcular qual a área de cobertura (MBR) de seus vizinhos. Por outro lado, um *repositório* simplesmente repassa uma mensagem que não foi destinada a ele *(l.29-30)*. Após certo período Γ de tempo, um *CH* associa como *repositório* aquele que tiver a menor distância dentre os *repositórios* conhecidos *(l.10-12)*. Em seguida, o *CH* envia uma mensagem de associação para o *repositório* informando sua MBR ($GC(s_i)$) *(l.13)*. O *repositório* de destino da mensagem atualiza sua MBR ($GR(s_i)$) com a informação da MBR enviada pelo *CH (l.21-22)*. Caso contrário, o sensor retransmite a mensagem para o *repositório* de destino *(l.23-24)*.

Como exemplo, considere o cenário ilustrado pela Figura 3.3, no qual o *repositório* $R3$ está em uma região de borda compreendida pelos agrupamentos $C4$, $C5$, $C9$ e $C10$, enquanto o *repositório* $R4$ está em uma vizinhança composta pelos agrupamentos $C6$, $C7$ e $C11$. Intuitivamente, uma vez que os *repositórios* armazenam informações sobre mais de um agrupamento, espera-se que o número de *saltos* realizados durante o processamento da consulta será menor.

Algoritmo 2 Escolha do Repositório

```
1:  procedure REPOSITORY_ELECTION(Threshold)
2:      knownRepo(s_i) ← {}
3:      if (NeighboursInDistinctClusters(s_i) > threshold) then
4:          role[R](s_i) ← TRUE
5:          broadcast('REPOSITORY_ANNOUCEMENT', s_i)
6:          WAIT(Δ time units)
7:          for all r in knownRepo(s_i) do
8:              send('MBR_ANNOUNCEMENT' ,s_i, r)
9:      else
10:         WAIT(Γ time units)
11:         if role[CH](s_i) = TRUE then
12:             R(s_i) ← getMinDistance(s_i, knownRepo(s_i))
13:             send('CH_JOIN_REPOSITORY', R(s_i), GC(s_i))
14:
15: procedure Receiving('REPOSITORY_ANNOUCEMENT', r) by s_i
16:     knownRepo(s_i) ← knownRepo(s_i) ∪ {r}
17:     if knownRepo(s_i) = {} OR dist(s_i, r) < dist(s_i, getMinDis(s_i, knownRepo(s_i))) then
18:         broadcast('REPOSITORY_ANNOUCEMENT', r)
19:
20: procedure Receiving('CH_JOIN_REPOSITORY', r, gc) by s_i
21:     if R(s_i) = r then
22:         GR(s_i) ← GR(s_i) ∪ gc
23:     else
24:         forward('CH_JOIN_REPOSITORY', r, gc)
25:
26: procedure Receiving('MBR_ANNOUNCEMENT', s_j, r) by s_i
27:     if R(s_i) = r then
28:         knownRepo(s_i) ← knownRepo(s_i) ∪ {s_j}
29:     else
30:         forward('MBR_ANNOUCEMENT', s_j, r)
```

3.3.2 Processamento de Consultas

O processamento da consulta do AQPM funciona da seguinte forma: qualquer sensor da rede pode iniciar uma consulta espacial ou por valor, desta forma o modelo não se limita a um único ponto de entrada de consulta. Um único parâmetro de consulta *range* é dado como entrada. Para consultas por valor, *range* é um intervalo de valores e o resultado consiste nos sensores na área monitorada com valores dentro deste intervalo. Para consultas espaciais, o intervalo (*range*) refere-se a uma região retangular e o resultado consiste nas leituras coletadas por sensores nesta região. Assim, as consultas de processamento consistem em encontrar dentro dos *repositórios* o conjunto de sensores e suas informações associadas que correspondem aos critérios de consulta e, em seguida, retornam as informações para o ponto de entrada. O protocolo de roteamento geográfico adotado é o GPSR (Karp e Kung, 2000) para a fase de encaminhamento e resposta da consulta. O algoritmo 3 detalha a estratégia de processamento da consulta.

O algoritmo *processamento da consulta* explora o modelo hierárquico do AQPM. Inicialmente, cada consulta associa um identificador único *qryId*. O algoritmo coleta os

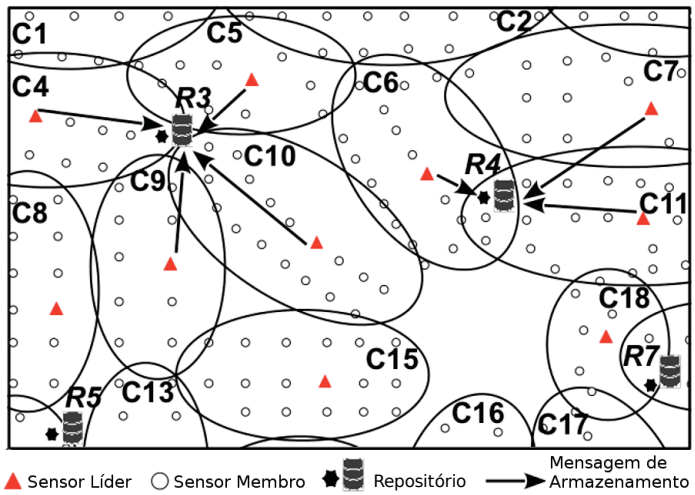


Figura 3.3: Posicionamento do repositório

resultados enviados de diferentes repositórios em uma variável *qryRes[qryId] (l.1-3)*. Um sensor designado como ponto de entrada recebe uma consulta e encaminha as consultas espaciais para a região de interesse e as consultas baseadas por valor para seu *CH (l.5-7)*. As consultas espaciais seguem a hierarquia do AQPM somente após o primeiro sensor na região de interesse ser alcançado *(l.16-24)*. Dentro da região, a consulta é encaminhada para um CH e, em seguida, para o respectivo *repositório (l.19-24)*. Para realizar a coleta de leituras, o procedimento *RECEIVING ('COLLECT')* é executado por todos os repositórios que estão no intervalo de consulta. Primeiro, os dados dos clusters armazenados nos *repositórios* que sobrepõem a região de consulta são obtidos pela função *GetDataThatOverlapsRange*, e enviados de volta ao ponto de entrada em uma mensagem *RESULT. (l.28-29)*. Então, uma mensagem *COLLECT* é enviada para *repositórios* vizinhos, que também sobrepõe a região de consulta *(l.30-32)*. Ao contrário das consultas espaciais, as consultas por valor são encaminhadas para todos os repositórios a fim de se obter o resultado da consulta *(l.39)*. A estratégia de encaminhamento segue o modelo hierárquico do AQPM *(l.40-43)*.

A Figura 3.4 ilustra uma consulta espacial, no qual os *repositórios* $R3$ e $R4$ estão em uma região de borda compreendida pelos agrupamentos $\{C_4, C_5, C_9, C_{10}\}$, e $\{C_6, C_7, C_{11}\}$, respectivamente. A região de interesse consiste no retângulo Q que envolve o repositório $R4$, que se sobrepõe aos *clusters* C_6, C_7, C_{10} e C_{11}. Suponha como ponto de entrada da consulta um sensor s_e do agrupamento $C8$. A consulta inicia-se com o envio da mensagem do ponto de entrada em direção à região de interesse Q. Suponha que o primeiro sensor em Q que recebe a consulta é um sensor s_q do *cluster* C_{10}. Após o recebimento da mensagem, s_q é encaminhada para o seu CH e, em seguida, para o repositório R_3. Os valores do *cluster* C_{10} são coletados de R_3 e enviados de volta ao ponto de entrada s_e. Entre os repositórios vizinhos de R_3, apenas o repositório R_4 se sobrepõem à região Q. Assim, a mensagem *COLLECT* é enviada para R_4, a partir da qual os valores dos *clusters* restantes que se sobrepõem Q são coletados e enviados de volta para s_e.

Como exemplo de uma consulta por valor, suponha a rede de sensores ilustrada pela Figura 3.5, no qual o repositório R1 é responsável por armazenar as leituras dos *clusters* $\{C_1, C_2, C_4\}$ e R2 armazena as leituras de $\{C_5, C_3\}$. Suponha como ponto de entrada da consulta um sensor s_e do agrupamento $C1$, que executa uma requisição para obter o conjunto de *clusters* que possuem leituras entre 19.5 e 20.5. A consulta inicia-se com o envio da mensagem do ponto de entrada em direção ao CH de C_1 e, em seguida, encaminhada ao repositório R_1. Os dados dos *clusters* armazenados nos *repositórios* que estão no intervalo $[19.5, 20.5]$ são obtidos pela função

Algoritmo 3 Processamento da Consulta

```
1:  procedure SEARCH(range, type) by s_i
2:      qryId ← new(id)
3:      qryRes[qryId] ← {}
4:      if type = SPATIAL then
5:          send ('SPATIAL', qryId, s_i, range, {}) in range direction
6:      else
7:          send ('VALUE', qryId, s_i, range, {}) to CH(s_i)
8:      WAIT(Ω time units) for response
9:      output qryRes[qryId]
10: procedure RECEIVING('RESULT', qryId, entryP, res) by s_i
11:     if s_i = entryP then
12:         qryRes[qryId] ← qryRes[qryId] ∪ res
13:     else
14:         forward ('RESULT', qryId, entryP, res) to entryP
15: procedure RECEIVING('SPATIAL', qryId, entryP, range, res) by s_i
16:     if position(s_i) not in range then
17:         forward ('SPATIAL', qryId, entryP, range, res, reached) in range direction
18:     else
19:         if s_i is a repository then
20:             execute ('COLLECT', qryId, entryP, range, res)
21:         else if s_i is a cluster-head then
22:             forward ('SPATIAL', qryId, entryP, range, res) to R(s_i)
23:         else
24:             forward ('SPATIAL', qryId, entryP, range, res) to CH(s_i)
25: procedure RECEIVING('COLLECT', qryId, entryP, range, res) by Rep
26:     if Rep has already processed qryId then
27:         drop message;  return;
28:     res ← GetDataThatOverlapsRange()
29:     send ('RESULT', qryId, entryP, res) to entryP
30:     for all repository r in knownRepo(Rep) do
31:         if MBR(r) overlaps range then
32:             send ('COLLECT', qryId, entryP, range, res) to r
33: procedure RECEIVING('VALUE', qryId, entryP, range, res) by s_i
34:     if s_i has already processed qryId then
35:         drop message;  return;
36:     if s_i is a repository then
37:         res ← GetDataThatIntersectRange()
38:         send ('RESULT', qryId, entryP, res) to entryP
39:         send ('VALUE', qryId, entryP, range, res) to knownRepo(s_i)
40:     else if s_i is a cluster-head then
41:         forward ('VALUE', qryId, entryP, range, res) to R(s_i)
42:     else
43:         forward ('VALUE', qryId, entryP, range, res) to CH(s_i)
```

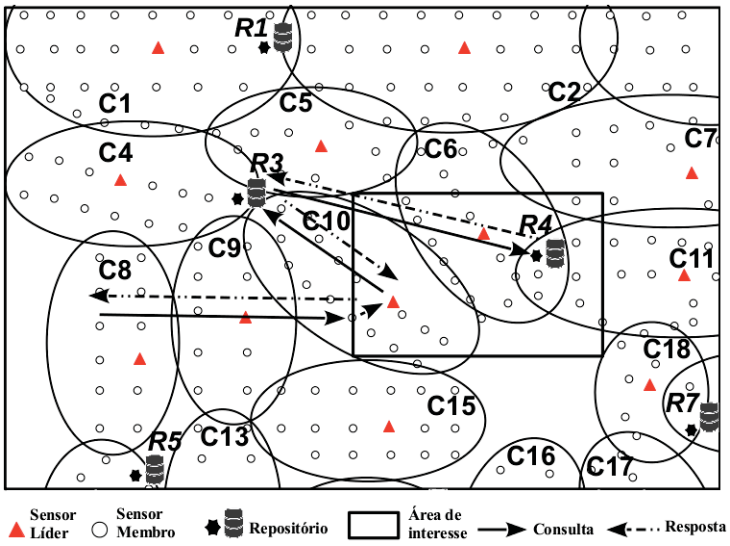


Figura 3.4: Consulta espacial

GetDataThatIntersectRange(). Assim, a mensagem *RESULT* é enviada para s_e com os valores $\{C_1, C_2, C_4\}$. Em seguida, $R1$ encaminha a mensagem para R_2, que processa a requisição e envia como resposta para s_e a mensagem *RESULT* contendo o *cluster* C_5.

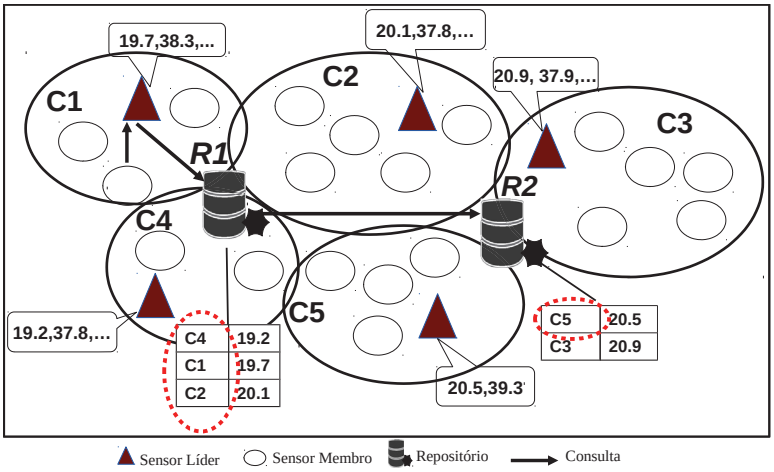


Figura 3.5: Consulta por valor

3.4 AVALIAÇÃO DO AQPM

As avaliações do AQPM e do IBIS ocorreram por meio de simulações no simulador de redes NS versão 2.35. No cenário empregado considerou-se uma região retangular de 1400m por 1000m, nós idênticos e estáticos, distância entre os nós em torno de 90m, com links simétricos, com raio de comunicação de 100m e protocolo MAC 802.11. Cada nó possui um GPS que informa sua posição sobre a região monitorada. No entanto podem-se assumir outras formas de localização, como a triangulação. As avaliações consideram também três cenários distintos para analisar a escalibilidade do AQPM, compostos por 140, 560 e 1260 sensores espalhados sobre a área monitorada. No início das operações da rede, os nós executam o algoritmo para definição dos agrupamentos e dos nós *repositórios* de dados. Em seguida, o nó 28 inicia o processamento

de uma consulta, a qual busca as médias das cinco leituras de todos os nós contidos dentro de um retângulo definido pelos vértices opostos $v_1(300, 50)$ e $v_2(950, 500)$, como ilustra a Figura 3.6. Os resultados mostrados nos gráficos são referentes aos dados coletados a partir do processamento desta consulta. Cada ponto plotado corresponde à média de 35 simulações, com intervalo de confiança de 95%. A Tabela 3.1 resume os principais parâmetros utilizados na simulação.

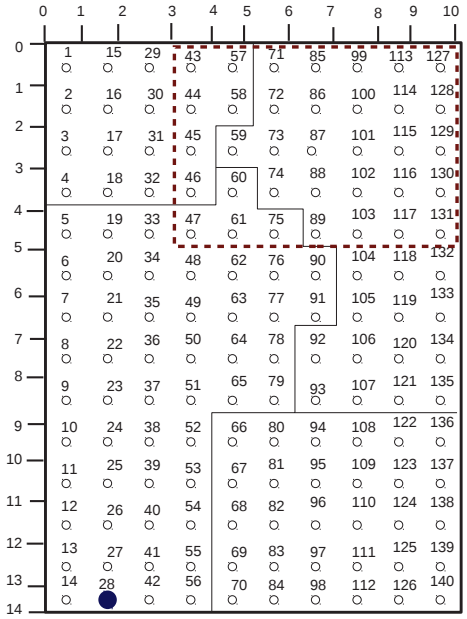


Figura 3.6: Cenário da simulação

Tabela 3.1: Parâmetros da simulação

Parâmetro	Cenário
Quantidade de sensores	140, 560, 1260
Fonte de dados	dados sintéticos
Área do ambiente	1400m por 1000m
Raio de comunicação do rádio	100m
$Threshold$ para agrupamento	$0.8 \leq \tau \leq 0.99$
$Threshold$ para *repositório*	$\tau >= 4$
Potência de transmissão	0.051W
Potência na recepção	0.048W

A avaliação dos experimentos realizou-se em duas etapas. O objetivo da primeira avaliação foi analisar o comportamento do modelo quanto à definição do número de *repositórios*, de líderes e de líderes isolados. Para cada simulação foram geradas novas leituras para os sensores distribuídos na área de monitoramento. Na segunda fase dos experimentos, o objetivo foi comparar o AQPM com o IBIS (da Silva et al., 2011), analisando-se três métricas: **consumo de energia (Joules)**, **tempo gasto para processamento da requisição (segundos)** e **porcentual de erro nas leituras obtidas**. A energia consumida pela rede para transmitir um pacote (E_{PT}) é igual a energia para transmissão de um pacote (E_{TX}), mais a soma da energia consumida por cada um dos n vizinhos do emissor para recebê-lo ($\sum_{x=1}^{n} E_{RE}$). Esse consumo pode ser modelado pela equação: $E_{(i \rightarrow j)PT} = E_{(i)TR} + \sum_{x=1}^{n} E_{(x)RE}$.

A geração dos dados de simulação levou em consideração a similaridade espacial dos nós na região monitorada. Assim, os nós sensores foram divididos em quatro regiões, como ilustrado pela Figura 3.6. Cada nó realizou a coleta de cinco leituras distintas, como temperatura e umidade, considerando-se que todas as leituras estão no intervalo entre 0 e 10. Inicialmente, foram criados quatro conjuntos de dados, sendo cada conjunto composto de 5 valores, gerados de forma aleatória. Cada conjunto foi associado a uma região e seus valores correspondem às sementes a partir das quais as leituras de cada nó sensor da região foram geradas. As leituras dos nós de uma mesma região são valores com variação aleatória de no máximo um porcentual V a partir do valor semente. Assim, a diferença na leitura dos nós de uma mesma região é no máximo V%.

3.4.1 Formação de Agrupamentos e Repositórios

Nestes experimentos, o objetivo foi determinar a porcentagem de *repositórios*, líderes e líderes isolados gerados pelo AQPM, em relação ao número de nós da rede. No gráfico da Figura 3.7, o eixo x representa o patamar de similaridade usado para definir os líderes (*threshold* do Algoritmo 1), o qual variou de 0.8 a 0.99, V foi fixado em 10% e o patamar para escolha de *repositórios* (*threshold* do Algoritmo 2) em todas as simulações foi 4.

Verifica-se que a porcentagem de *repositórios* praticamente não se altera. A definição de um *repositório* leva em consideração o número de líderes que um dado sensor é capaz de escutar. Por isso, a variação da similaridade praticamente não influencia a porcentagem de *repositórios* na rede. Em geral, a porcentagem máxima de nós selecionados como *repositórios* ficou abaixo de 10% em todos os cenários avaliados. Este resultado mostra que a estratégia do modelo de concentrar leituras em *repositórios* ao invés de mantê-los apenas nos líderes dos agrupamentos foi acertada. Como o processamento das consultas requer apenas acessos aos *repositórios*, o número de sensores contactados é pequeno, mesmo que as consultas espaciais cubram regiões geográficas extensas. O gráfico também mostra que, com o aumento do patamar de similaridade para definição de líderes, o número de líderes isolados cresce e de líderes com nós subordinados diminui. Isso ocorre em consequência da diminuição do número de nós com similaridade acima do patamar, o que gera mais líderes isolados. Além disso, o número de *repositórios* necessários para armazenar os dados de agrupamentos vizinhos também tende a ser menor quando há mais agrupamentos com nós subordinados do que agrupamentos com líderes isolados. Isso ocorre porque nesse caso o número total de agrupamentos formados tende a ser menor. Em cenários reais, nos quais há maior similaridade entre sensores vizinhos, acredita-se que a quantidade de líderes isolados seja ainda menor.

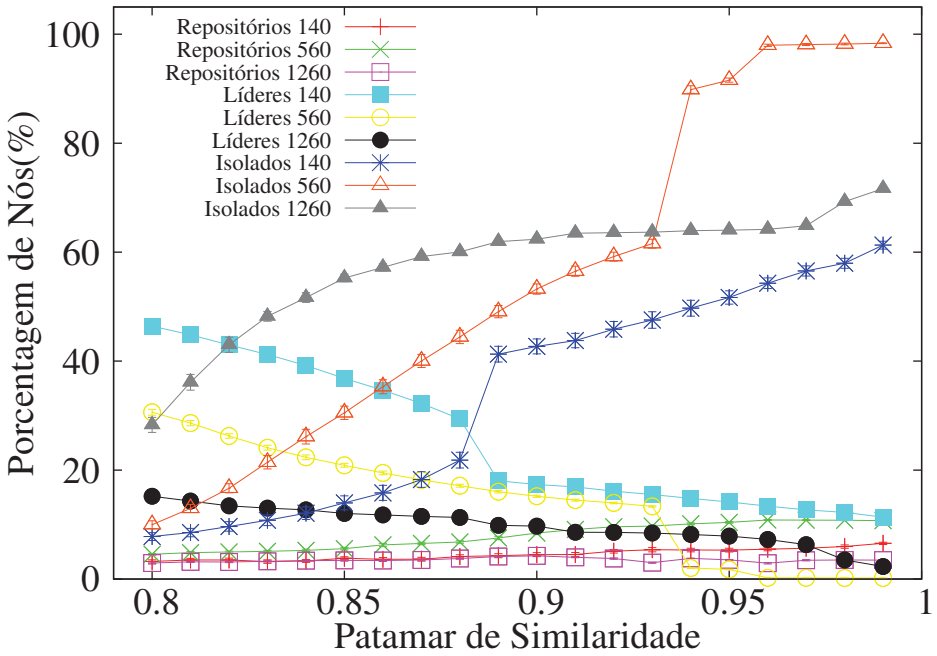


Figura 3.7: Similaridade na geração de agrupamentos

No gráfico da Figura 3.8, o eixo X representa a porcentagem de variação máxima V dos dados gerados como entrada para a simulação. Verifica-se que esse parâmetro não altera consideravelmente a formação de agrupamentos e a definição dos *repositórios*. Nestes experimentos, considerou-se um patamar de 0.88 de similaridade para os líderes.

Ainda nesse contexto, é importante analisar a quantidade de memória necessária para o funcionamento do AQPM. Para isso, analisaremos somente o pior caso, a memória necessária para um *repositório*. Tal nó precisa armazenar sua posição geográfica (2 bytes), sua MBR (4 bytes), suas leituras ($L * TL$ bytes, onde L é o número de leituras e TL o espaço ocupado por uma leitura), a MBR e as leituras de cada um dos NCH líderes associados a ele e a posição geográfica e a MBR de cada um dos NR repositório conhecidos. Logo, o custo de armazenamento de um repositório pode ser calculado pela seguinte equação: $(6 + (L * TL)) + NCH * (4 + L * TL) + (2 + 4) * NR$. No pior caso em nossos experimentos, ao considerar que um repositório sabe informações de todos os outros repositórios da rede e pela análise do gráfico ilustrado na Figura 3.7, temos $NR = 53$. Em análises dos logs dos experimentos, verificou-se que e o maior número de líderes associados a um repositório é $NCH = 54$. Logo, considerando $TL = 1$ byte, o custo máximo de armazenamento de um repositório foi de 834 bytes, menor que os 10 KB de memória RAM da maioria dos nós sensores descritos na literatura.

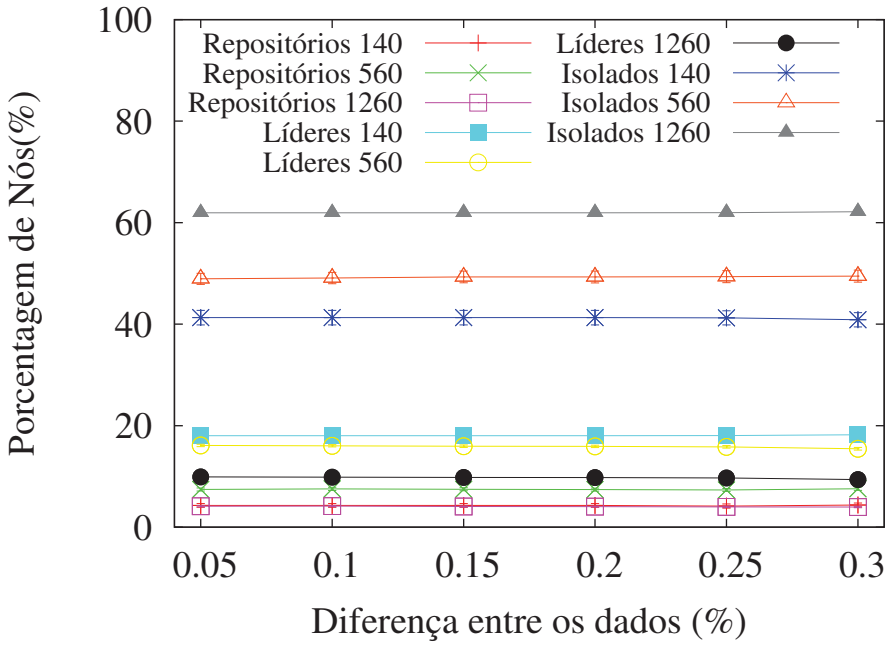


Figura 3.8: Impacto do grau de similaridade das leituras

3.4.2 Processamento de Consultas AQPM e IBIS

Nesta seção, o processamento de consultas do AQPM é comparado com o IBIS, o qual cria um itinerário dentro da região de coleta de forma que todos os sensores contidos nesta região são vizinhos do itinerário ou fazem parte dele. É importante salientar que o processamento de requisições do IBIS se altera somente com alterações na topologia da rede. Como nos experimentos aqui apresentados, a topologia da rede mantém-se inalterada, o comportamento do IBIS foi constante em todos os cenários analisados. Porém, dentre os trabalhos até então encontrados na literatura, tal algoritmo é o que apresenta o melhor desempenho no cenário utilizado na avaliação.

O gráfico da Figura 3.9 mostra que o AQPM consome menos energia que o IBIS, porque as requisições não precisam ser disseminadas para todos os nós na região de consulta, basta elas alcançarem os *repositórios* que contém dados dessa região para se obter a resposta. Em relação ao tempo de resposta, apresentado pelo gráfico 3.10, para 140 nós o AQPM é mais eficiente porque o

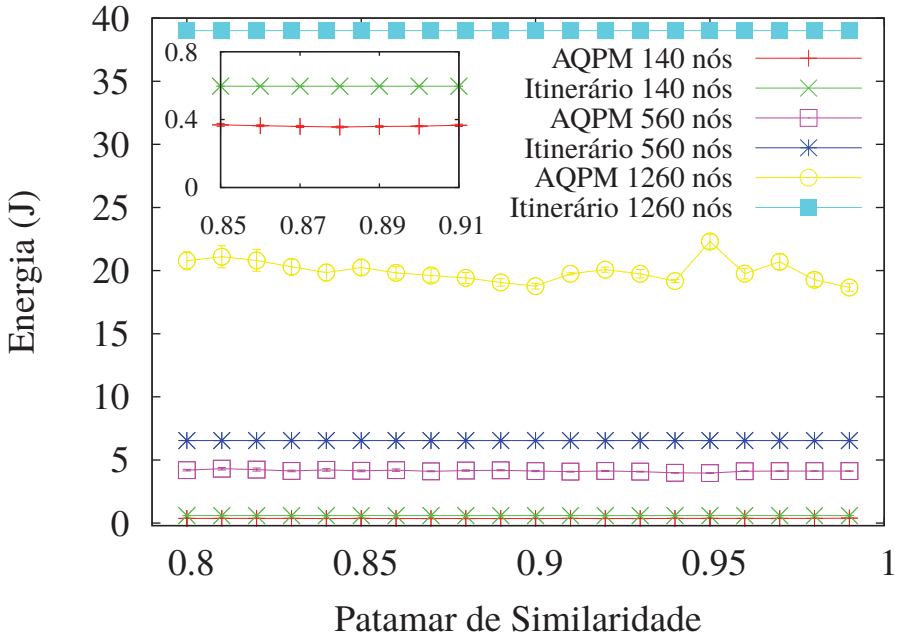


Figura 3.9: Consumo de energia

IBIS cria vários *delays* durante a disseminação da consulta pelo itinerário. No cenário de 560 nós, o desempenho do AQPM mostra-se inferior que o IBIS somente quando a similaridade ultrapassa 0.95, pois a partir deste limiar o AQPM apresenta o maior número absoluto de *repositórios* de dados dentre os cenários analisados. No entanto, no cenário com 1260 nós, o AQPM apresenta pior desempenho que o Itinerário. Isto ocorre pois o Itinerário tenta encaminhar a consulta sempre para o vizinho que estiver mais próximo da região de interesse. Dado que no cenário de 1260 cada nó apresenta vizinhos mais próximos dessa região, o encaminhamento e a transmissão da resposta até o nó que iniciou a requisição são realizados de maneira mais rápida.

De acordo com o gráfico 3.11, o AQPM apresenta uma margem de erro enquanto que o IBIS retorna o resultado exato, uma vez que as leituras são obtidas diretamente dos dispositivos sensores da região de interesse. Os valores apresentados mostram a porcentagem de diferença entre a média das leituras de todos os sensores contidos na região de coleta e a média obtida com o processamento da requisição. Verifica-se que o erro do AQPM varia em torno de 6% no cenário composto por 140 e 560 sensores, e relativamente um pouco maior para o cenário composto por 1260 sensores. Esse erro vem do fato que as leituras feitas por um líder representam a leitura do seu agrupamento. É importante observar que o *trade-off* entre o gasto de energia e a precisão dos resultados é conhecido na literatura. Curiosamente, Boulis et al. (2003) encontraram resultados semelhantes, isto é, um redução de cerca de cinco vezes no consumo de energia para um erro de 5%. No entanto, é preciso levar em consideração que a média do erro é calculada a partir das leituras de 5 atributos de sensoriamento, o que geralmente não ocorre nos modelos tradicionais onde a média é calculada a partir de um único atributo do ambiente.

Além das métricas apresentadas, torna-se importante analisar o **custo energético do processo de definição de líderes e *repositórios***. O gasto energético do AQPM ficou em torno de 0.35 Joules. Por outro lado, o consumo do IBIS foi menor, cerca de 0.054 Joules, pois necessita apenas de uma inundação na rede. Entretanto, essa diferença é compensada com o processamento de mais requisições. Como o AQPM consome menos energia para processar, o consumo da rede durante todo seu período de operação tende a ser menor. Esses dados não são mostrados nos gráficos porque não ocorreram grandes variações dos valores acima mencionados.

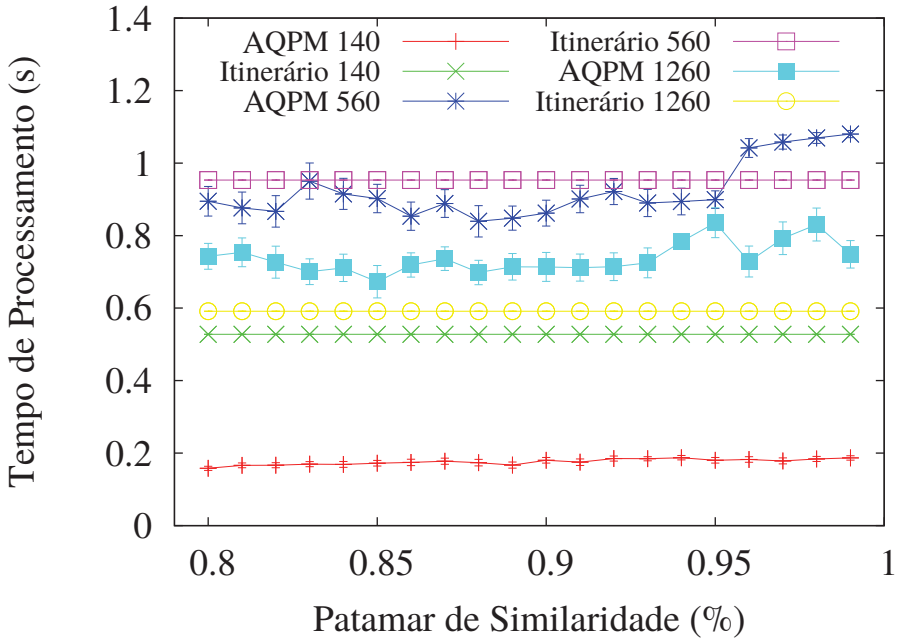


Figura 3.10: Tempo de resposta

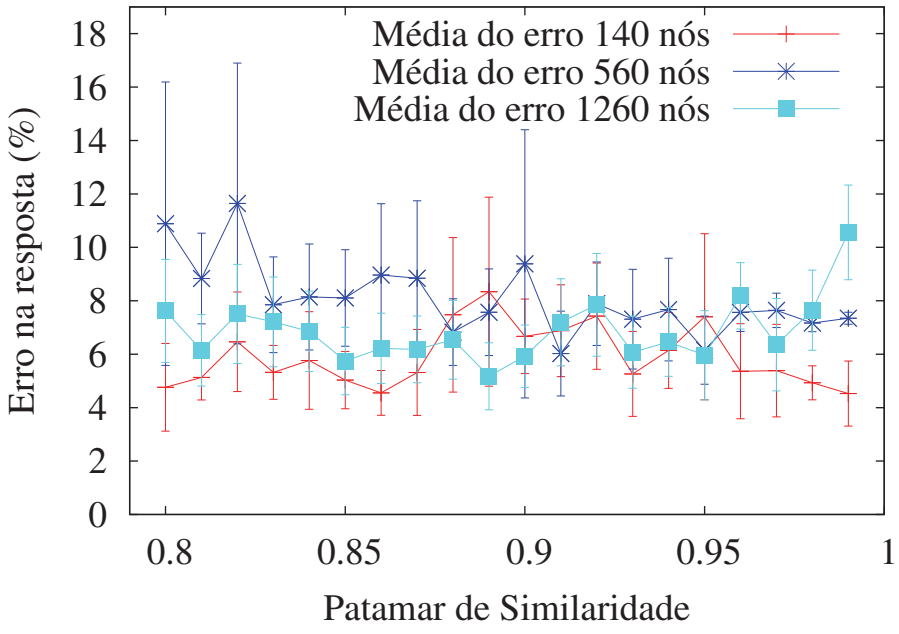


Figura 3.11: Percentagem de erro do resultado

Estes resultados como um todo mostram que o AQPM é um modelo que atende os requisitos de **escalabilidade** e **autonomia** para RSSFs urbanas. Mas, é preciso analisar e estender o trabalho com formas alternativas para reduzir o erro relativo das consultas. Observa-se que o AQPM determina a similaridade entre sensores, considerando um *conjunto* de métricas, e não apenas um tipo de leitura.

3.5 RESUMO

Este capítulo apresentou um modelo **autônomo** e **escalável** para processamento de requisições espaciais e por valor em redes urbanas, chamado de AQPM. O modelo foi inspirado em características normalmente encontradas em aplicações de sensoriamento urbano, como a alta densidade de sensores na rede para monitorar regiões extensas e a correlação espacial presente no dado sensoriado. Neste modelo, os dados de sensoriamento são distribuídos em estruturas denominadas de *repositórios*. Simulações mostraram que a estratégia adotada pelo AQPM reduz significativamente o fluxo de mensagens durante o processamento de consultas. Isso mostra que as técnicas utilizadas pelo modelo de armazenamento distribuído do AQPM apresentam um bom *trade-off* entre os custos de consulta e de armazenamento. Sem a definição de agrupamentos ou *repositórios*, seria preciso inundar a rede com consultas para se obter o resultado desejado, o que é impraticável em redes de grande escala. Embora não detalhado nesta tese, os agrupamentos, CHs e *repositórios* são modificados de forma autônoma e similar aos algoritmos apresentados, sempre que a correlação espacial de um CH deixar de satisfazer o *threshold* pré-estabelecido. No entanto, o AQPM mostrou uma taxa de erro pequena que deve ser levada em conta de acordo com os critérios da aplicação executada na rede.

4 UM MODELO BASEADO EM COMPONENTES REUTILIZÁVEIS PARA SIMULAÇÃO DE RSSFs

A simulação é uma ferramenta importante para validação de sistemas antes da implantação. No entanto, a maioria das implementações em ambientes de simulação está diretamente relacionada a um sistema específico de RSSF, ou então está integrada a outra aplicação, dificultando o reuso por futuros desenvolvedores. Os modelos de componentes de software são uma boa abordagem para enfrentar esses problemas, porque reduzem a complexidade do desenvolvimento e melhoram a produtividade. Este Capítulo descreve o RCBM (***R**eusable **C**omponent-**B**ased **M**odel for WSN Storage Simulation*), um modelo baseado em componentes reutilizáveis para a simulação de modelos de armazenamento para RSSFs. O RCBM promove a reutilização de software a partir de componentes existentes para melhorar a eficiência do desenvolvimento e avaliação do sistema. O RCBM foi implementado no simulador NS2 e os resultados experimentais mostram que o RCBM é mais flexível do que os modelos baseados em componentes anteriores para RSSFs. Devido à sua abordagem de propósito geral, o RCBM pode ser aplicado para desenvolver código de simulação para uma ampla gama de modelos de armazenamento em RSSFs, reduzindo o esforço de desenvolvimento.

4.1 INTRODUÇÃO E MOTIVAÇÃO

O desenvolvimento de sistemas de armazenamento é uma tarefa difícil em virtude das restrições computacionais dos sensores e das especificidades do ambiente. Os *repositórios* distribuídos de dados e o agrupamento de sensores são abordagens usadas para tratar os problemas da escalabilidade e do armazenamento de dados em RSSFs. Embora esforços recentes tenham sido feitos para desenvolver sistemas eficientes de armazenamento de dados, a natureza específica das RSSFs e a falta de um *framework* padrão de desenvolvimento tornam o projeto desses aplicativos uma tarefa complexa (Minakov et al., 2016). Em geral, os sistemas são implementados a partir do zero, para atender aos requisitos específicos do sistema ou estão integradas em complexas aplicações, dificultando a separação de conceitos e limitando a reutilização de código (Malavolta e Muccini, 2014). Portanto, é desejável utilizar ferramentas de modelagem de sistemas que possibilitem abstrair os detalhes de baixo nível, reduzindo a complexidade do desenvolvimento e melhorando a produtividade dos desenvolvedores de sistemas de simulação de RSSFs.

As abordagens de modelagem de software oferecem um nível apropriado de abstração para o desenvolvimento de sistemas distribuídos. No contexto dos sistemas de armazenamento em RSSFs, os modelos de componentes de software permitem aos desenvolvedores construir sistemas a partir da reutilização de componentes existentes, com base em especificações funcionais, ou seja, usando-se técnicas para definir as *interfaces* e padrões de interação dos componentes (Crnkovic

et al., 2011). Alguns modelos de componentes, como OSGi (Alliance, 2007) e OpenCOM (Coulson et al., 2008), descrevem as *interfaces* como um conjunto de operações (funções). Outros modelos usam o conceito de *portas* como elementos de interface, usadas pelos componentes para receber e enviar dados. A modelagem de componentes tem sido usada em diferentes domínios de aplicação, na área de software automotivo (Jo et al., 2015), CPS (Masrur et al., 2016), simulação de RSSFs (Coulson et al., 2012) e IoT (Patel et al., 2011). Nos ambientes de simulação em RSSFs, os modelos baseados em componentes são uma abordagem promissora para gerenciar cenários de aplicativos em ambientes de grande escala.

Inspirados por esses desafios, alguns trabalhos foram propostos para melhorar a produtividade reutilização do código durante o desenvolvimento de sistemas. A maioria deles enfatiza a importância de identificar conceitos e funcionalidades comuns em diferentes áreas de aplicação, como na organização da rede e no processamento de dados (Salman e Al-Yasiri, 2016; Tei et al., 2015) e IoT (Patel e Cassou, 2015). Em nosso entendimento, o único modelo baseado em componentes específico para o armazenamento de dados em redes de sensores é o CBCWSN (Amaxilatis et al., 2011b).

Para tratar esta questão, foi proposto o RCBM (***R**eusable **C**omponent-**B**ased **M**odel for WSN Storage Simulation*), um modelo reutilizável de componentes para a simulação de sistemas de armazenamento em RSSFs. O foco do RCBM está em especificar as entidades de nível de armazenamento, definindo um conjunto de conceitos e funcionalidades comuns que representam várias instâncias de sistemas de armazenamento. O RCBM permite que os desenvolvedores reutilizem componentes existentes, reduzindo a complexidade para projetar e desenvolver novos sistemas. O modelo de componentes do RCBM e do CBCWSN diferem em certos aspectos. Enquanto o CBCWSN tem um fluxo rígido de execução, o RCBM adota um modelo de execução flexível baseado em um coordenador, responsável pelas trocas de mensagens. Este modelo está de acordo com o simulador de rede NS2, no qual o RCBM foi implementado. Para avaliar o funcionamento do RCBM, foram realizados três estudos de caso, implementando os sistemas LCA Baker e Ephremides (1981), LEACH Heinzelman et al. (2000) e MAX-MINAmis et al. (2000). Os resultados mostram que o percentual mínimo de reutilização de código usando o RCBM foi de 66 % para o MAX-MIN, enquanto que o CBCWSN reportou, para este caso, reutilização de 23 %.

4.2 TRABALHOS RELACIONADOS

A utilização de componentes de software que buscam facilitar a implementação de sistemas que possuam diversos elementos em comum é algo recorrente em Ciência da Computação. A ideia de construir sistemas a partir de componentes existentes e tecnologias de acesso a dados está presente nos modernos conceitos e paradigmas de programação (Gabbrielli e Martini, 2010), e até em outras áreas dentro da engenharia de software, como as linhas de produtos de software (Clements e Northrop, 2015), desenvolvimento de software orientado a aspectos (Filman, 2005; Ali et al., 2010) ou arquiteturas orientadas a serviços (Erl, 2005; Welke et al., 2011).

As ferramentas de simulação de rede são úteis para testar e comparar sistemas antes da implantação em cenários reais, fornecendo informações sobre o comportamento dos sistemas. Em ambientes de simulação de RSSFs, os modelos de componentes são uma abordagem promissora para gerenciar o aumento da demanda dos requisitos de escalabilidade e autonomia dos sistemas. A arquitetura modular do RCBM, inspirada nos princípios de encapsulamento e padrões de projetos para a criação de componentes reutilizáveis, pode ser implementada em qualquer simulador conhecido, como o NS2 (Sundani et al., 2011), NS3 (Riley e Henderson, 2010) e OMNeT++ (Varga e Hornig, 2008).

Trabalhos que aplicam separação de conceitos, como SenNet (Salman e Al-Yasiri, 2016), MDDWSN (Tei et al., 2015), CBCWSN (Amaxilatis et al., 2011b) e IoTSuite (Patel e Cassou, 2015), destacam aspectos específicos do sistema em construção. O MDDWSN e o SenNet propõem um processo de desenvolvimento de software para abordar conceitos relacionados ao processamento de dados e com conceitos relacionados à rede. Essas ferramentas, no entanto, não suportam o conceito de *repositórios* de dados distribuídos e são implementadas no sistema operacional TinyOS, o que limita sua aplicabilidade a redes de pequena escala. IoTSuite, por outro lado, especifica os conceitos comuns presentes nos cenários de IoT. É proposto uma entidade de armazenamento centralizada, que pode lidar grande fluxo de dados e comunicação, sendo menos escalável do que abordagens descentralizadas. O CBCWSN identificou elementos e funcionalidade comuns encontrados na maioria dos algoritmos de agrupamento para RSSFs. No entanto, ao contrário do RCBM, o CBCWSN adota um fluxo de execução predefinido. Como resultado, é mais difícil estender o CBCWSN a outros modelos de armazenamento que não sigam o seu padrão de interação.

4.3 O MODELO DE COMPONENTES DE ARMAZENAMENTO DE DADOS

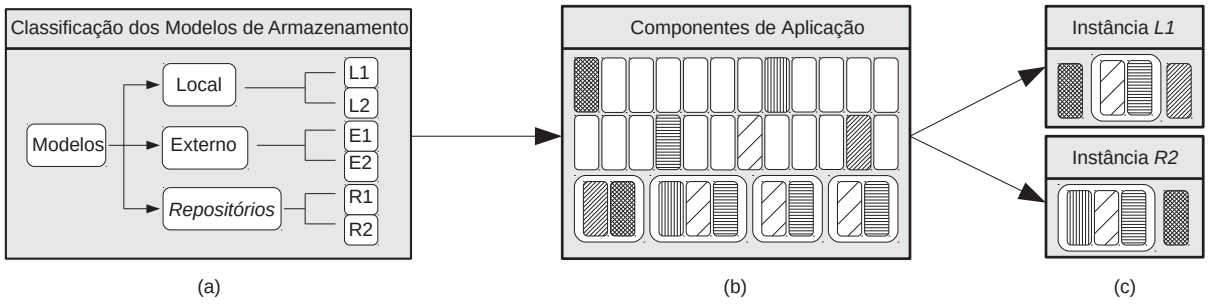


Figura 4.1: Processo de desenvolvimento baseado em componentes para modelos de armazenamento em RSSFs

Há uma grande quantidade de trabalhos que propõem modelos aplicados a sensores (Diallo et al., 2015). Embora estes modelos estejam relacionados a um sistema específico de RSSF, alguns conceitos (ou entidades) são comuns a uma grande maioria destes modelos. Nesta seção propomos o ***R**eusable **C**omponent-**B**ased **M**odel* (RCBM), um modelo de componentes para a simulação de sistemas de armazenamento em RSSFs. Ele consiste em um metamodelo de armazenamento para RSSFs que utiliza entidades, propriedades e funções para descrever o que seriam as instâncias desse metamodelo, ou seja, as propostas de modelo de armazenamento. O RCBM torna possível a reusabilidade de código, facilitando o desenvolvimento de novos modelos. Como resultado, novos modelos específicos de determinadas aplicações podem ser criados ao desenvolver algumas funções em vez de considerar o problema de armazenamento como um todo. Desta forma, o RCBM promove *reutilização de código* e suporta o *desenvolvimento ágil* de novos modelos.

Para determinar as entidades comuns entre os modelos existentes, primeiro identificou-se três classes de armazenamento, com base no local onde os dados de sensoriamento são armazenados.

As classes, com ilustra a Figura 4.1(a) são: *Local*, refere-se aos modelos que armazenam dados localmente na memória *flash* dos sensores; *Externo*, relaciona-se aos modelos que armazenam em que todos os dados coletados pelos dispositivos sensores são armazenamento em uma entidade externo; e *Repositórios*, que incluem modelos em que alguns sensores são responsáveis pelo armazenamento de dados de um conjunto de sensores. Em cada uma das classes, é possível identificar componentes comuns para criar um metamodelo RCBM. Em RSSFs densas, manter os dados agrupados em repositórios reduz o número de transmissões de dados para consultar dados (D´ Angelo et al., 2016), um requisito importante para alcançar a escalabilidade. Entre os modelos de armazenamento descritos em nossa taxonomia, a classe *repositórios* tem atraído muita atenção, já que é mais escalável do que as outras abordagens. Assim, esta tese centra-se no desenvolvimento de componentes para modelos da classe *repositórios*, que é o mais apropriado para redes de grande escala. É importante ressaltar que a abordagem baseada em componentes proposta pode ser estendida para considerar outras classes de armazenamento, desde que suas entidades sejam devidamente modeladas.

No RCBM, cada entidade está associada a um conjunto de componentes que implementam funcionalidades comuns. Para modelos na classe de *repositório*, essas entidades incluem: *(i)* dispositivo sensor; *(ii) clusters*, que são arranjos lógicos de sensores; e *(iii) cluster-heads* (CHs), que correspondem aos sensores responsáveis por armazenar informação do *cluster*. Essas entidades definem um modelo de armazenamento hierárquico, onde cada agrupamento designa um sensor como seu líder para armazenar as leituras dos membros do grupo. Esta abordagem compartilha algumas tarefas comuns, conforme descrito em Amaxilatis et al. (2011b). O primeiro está relacionado à estratégia de eleição do líder (CH), que pode ser uma escolha aleatória ou com base em um atributo do sensor. A segunda tarefa refere-se à associação ao grupo, ou seja, como os nós que não foram eleitos como líderes decidem qual *cluster-head* eles vão se juntar. Os critérios incluem decisões aleatórias, decisões baseadas em localização e baseadas em atributo.

Em um projeto baseado em componentes, os componentes são unidades reutilizáveis e o seu desenvolvimento é mantido separado da implementação do sistema (Crnkovic et al., 2011). Esta ideia está ilustrada na Figura 4.1, mostrando que os componentes existentes podem ser combinados e reestruturados para desenvolver novos modelos, representados na figura como instâncias modelo.

Além dos componentes de aplicação, que estão diretamente relacionados aos modelos de armazenamento em RSSFs, o RCBM define dois outros tipos de componentes: coordenador e biblioteca. O primeiro é responsável por coordenar o fluxo de execução e a interação entre os componentes da RSSF. O segundo fornece funcionalidades adicionais que são comuns a vários modelos. Assim, existem três tipos distintos de componentes no RCBM:

- Componentes de biblioteca: implementam funcionalidades úteis para os modelos, como funções de agregação e temporizadores;
- Componentes de aplicação: compõem o modelo de armazenamento e possuem um conjunto de funções que podem ser chamadas por outros componentes ou pelo coordenador, de acordo com o funcionamento do sistema proposto;
- Componente do coordenador: controla o fluxo de execução do sistema.

Cada componente é definido a partir de um *template* e de sua implementação. O *template* define os protótipos das funções, descrevendo seus parâmetros de entrada e os valores de saída, ou seja, a interface do componente.

Implementações específicas podem ser definidas para cada modelo. A separação da interface de sua implementação possibilita criar um conjunto de componentes conectável,

que pode promover a reutilização de várias maneiras. Ele permite que os componentes de biblioteca sejam usados no desenvolvimento de qualquer componente de aplicação. Além disso, o coordenador pode ser reutilizado para aplicar o mesmo fluxo de execução para diferentes implementações de componentes de aplicação. Portanto, os mesmos componentes de aplicação podem ser reutilizados em um fluxo de execução diferente por uma nova implementação do coordenador.

4.3.1 A Plataforma RCBM

A arquitetura do RCBM, composta pelas camadas de especificação, implementação e comunicação, está representada na Figura 4.2. A camada de especificação é composta pelos componentes de biblioteca, *template* do coordenador e *templates* dos componentes de aplicação. Esses elementos são utilizados da forma como foram especificados, sem a necessidade de alterações realizadas pelo usuário. A camada de implementação é a camada que de fato será construída pelo desenvolvedor, e contém o código que implementa os componentes de aplicação e o componente do coordenador, que estão definidos da camada de especificação. A camada de comunicação é a responsável por interligar as camadas de especificação e de implementação, fornecendo a infraestrutura de comunicação entre os componentes. No caso das redes de sensores sem fio, a camada de comunicação é a própria ferramenta de simulação. A separação entre especificação e implementação facilita o desenvolvimento, promovendo a composição de novos componentes com os desenvolvidos anteriormente.

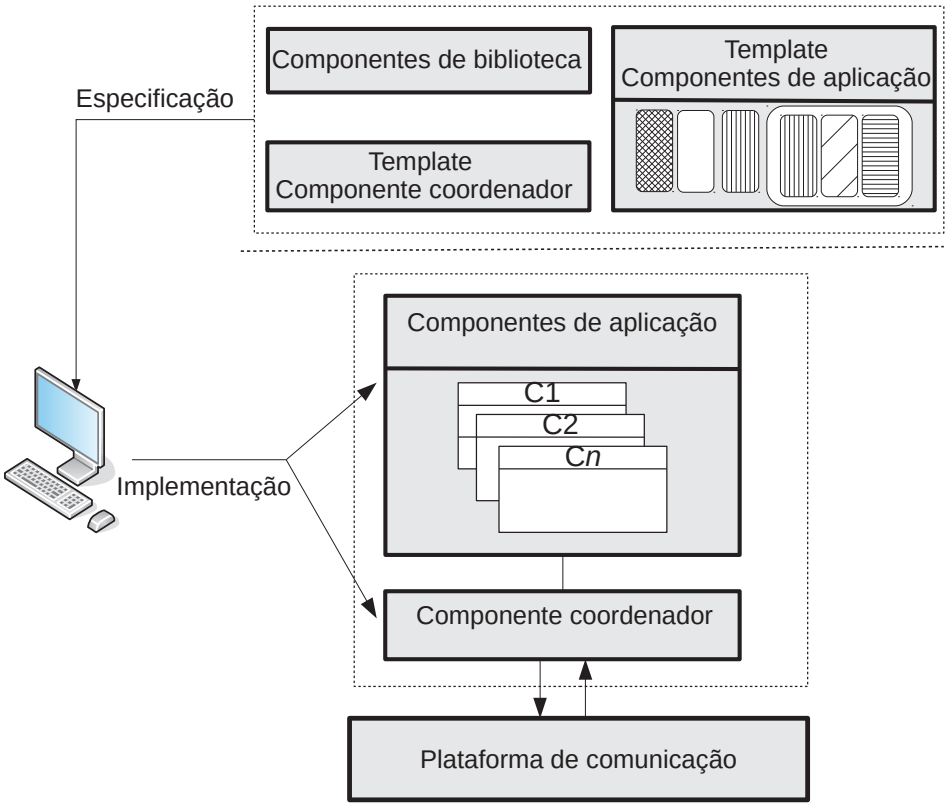


Figura 4.2: Arquitetura do RCBM

No processo de desenvolvimento, o programador começa com um conjunto de bibliotecas de funções e *templates* de componentes que podem ser usados para construir o modelo desejado. Durante a implementação, que compreende a combinação de componentes e o coordenador, os *templates* são transformados em componentes, pela adição de porções de código definidas pelo usuário (implementações de componentes). Os componentes da aplicação associam funcionalidades a entidades das RSSFs, como *clusters* e *CHs*, enquanto o coordenador controla o fluxo de execução entre eles. Na camada de comunicação, a plataforma fornece o suporte de tempo de execução para o ambiente de simulação. Além disso, o RCBM fornece um esqueleto de código que integra o coordenador com a camada de comunicação, abstraindo os detalhes de baixo nível desta camada.

Nas seções que seguem, são apresentados os detalhes dos componentes de biblioteca, *templates* e componentes de aplicação.

4.3.2 Componentes de Biblioteca

Os componentes da biblioteca não estão diretamente relacionados às entidades de aplicação, mas fornecem funções úteis que podem ser usadas no desenvolvimento de outros componentes. Nesta seção, são apresentadas as funções de agregação e o componente de temporização.

As funções de agregação são executadas em um conjunto de valores, para obter um único e representativo valor, como o valor máximo (MAX), o valor mínimo (MIN), a média (AVG), a soma (SUM) e a cardinalidade do conjunto (COUNT). As funções de agregação são úteis em várias operações, por exemplo, em modelos de armazenamento agrupado em *repositório*, os *cluster-heads* (CHs) são os únicos sensores contactados para processar consultas na rede. Em geral, os CHs não reportam as leituras individuais de todos os membros do *cluster*, mas fornecem um único valor representativo, que pode ser obtido por uma função de agregação. O objetivo desta abordagem é minimizar o volume de tráfego e o número de transmissões na rede.

O Código 4.1 ilustra a especificação do componente de funções de agregação que define as operações MIN, MAX, AVG e SUM.

```
1  class Library_Aggregation {
2  public:
3    template <typename K, typename V>
4      map<K, V> MIN(map<K, V>);
5
6    template <typename K, typename V>
7      map<K, V> MAX(map<K, V>);
8
9    template <typename K, typename V>
10     V AVG(map<K, V>);
11
12   template <typename K, typename V>
13     V SUM(map<K, V>);
14 }
```

Código 4.1: Template do componente da biblioteca de aggregagação

Observe que as declarações de função usam a sintaxe de *template* do C++, uma funcionalidade útil para lidar com diferentes tipos de dados (de entrada e saída). Em nossa abordagem, todas as funções agregadas operam em um conjunto de pares chave-valor “map<K, V>”. Por exemplo, suponha que um *cluster-head* receba a leitura 10.2 de um membro do

cluster s_2. Então, esta informação pode ser representada como o par chave-valor $(s_2, 10.2)$, onde s_2 é a chave K, ou seja, o identificador único do sensor e 10.2 é o valor associado V. Dado um conjunto de pares chave-valor que representa as leituras dos membros do *cluster*, $\{(s_2, 10.2), (s_3, 10.5), (s_4, 10.7)\}$, o *cluster-head* pode obter a média do *cluster* aplicando a função AVG, que retorna o valor 10.47. As funções MIN e MAX retornam um par de chave-valor em vez de um valor. O resultado da operação MIN é representado por $(s_2, 10.2)$, enquanto o resultado para a operação MAX é dado por $(s_4, 10.7)$.

```
1 class Timer : public TimerHandler {
2 public:
3   virtual void expire(Event *e);
4   template <class T> void initTimer(T);
5   template <class T> void restartTimer(T);
6 }
```

Código 4.2: Template do componente da biblioteca de timer

Outro componente útil para as aplicações em RSSFs é o temporizador. Ele fornece funções para fins de sincronização, incluindo transmissões de mensagens e em tarefas de coleta de dados. O Código 4.2 ilustra a especificação do *template* do componente Timer.

A biblioteca de temporização fornece funções para inicializar o temporizador "initTimer(T)" e para reiniciar o temporizador "restartTimer(T)". A função "expire(Event * e)" é automaticamente invocada quando o temporizador expirar. Como consequência, o usuário não precisa verificar manualmente a expiração do temporizador.

4.3.3 Template Componentes de Aplicação

Os componentes de aplicação, diferente dos componentes de biblioteca, estão diretamente relacionados às funcionalidades das entidades identificadas nas RSSFs. Além disso, as funções de componentes de biblioteca fornecem operações úteis que os programadores podem usar sem conhecer os detalhes internos. Por outro lado, diferentes modelos de armazenamento em RSSFs podem envolver etapas específicas e, portanto, o programador tem que modificar ou implementar alguns dos componentes de aplicação, associados às entidades do modelo. Como o foco está em modelos da classe de *repositório*, essas entidades são: o dispositivo de sensor, *clusters* e *cluster-heads* (CHs).

A principal característica apresentada em sistemas autônomos e escaláveis baseia-se em seguir uma solução distribuída em vez de uma abordagem centralizada. Nas arquiteturas distribuídas, os sensores possuem apenas uma visão local sobre a rede. Por exemplo, um sensor s_i mantêm informações sobre seus vizinhos $N(i)$, tais como identificador, coordenadas geográficas, nível de energia, anúncios de eleição de CHs e anúncios para associação ao CH. O Código 4.3 ilustra a especificação do componente sensor, que fornece algumas das funcionalidades apresentadas anteriormente.

```
1 class Sensor {
2 public:
3   template <typename K, typename V>
4   void setNeighborReadings(map<K, V> );
5
6   template <typename K, typename V>
7   map<K, V> getNeighborReadings ();
8
```

```
9    template <typename K, typename V>
10   map<K, V> getCandidateCHs ();
11 }
```

Código 4.3: Template do componente sensor

```
1  class CH_Election {
2  public:
3    template <typename K, typename V>
4    void selectCH(map<K, V> );
```

Código 4.4: Template do componente CH_Election

Para a entidade de *cluster*, foram identificadas duas fases principais nos algoritmos de agrupamento: eleição de *cluster-head* (CH) e formação de *cluster*. Assim, define-se dois componentes de aplicação: CH_Election e Cluster_Formation, que estão ilustrados no Código 4.4 e 4.5, respectivamente. As estratégias dos algoritmos de eleição de CHs podem ser classificadas em três categorias: predefinida, aleatória e baseada em atributos Afsar e Tayarani-N (2014). O modelo fornecido pelo componente CH_Election é genérico o suficiente para implementar os principais critérios de seleção CHs.

O componente CH_Election especifica a função "selectCH(map<K, V>)" que os usuários devem implementar de acordo com o critério para a seleção de CH do modelo. Observe que a função opera em conjunto de pares chave-valor (map<K, V>). Como exemplo, considere o modelo probabilístico onde um sensor anuncia-se CH de acordo com uma probabilidade p. Intuitivamente, cada sensor s_i executa $selectCH(s_i, p_i)$, onde $K = s_i$ e $V = p_i$.

O *template* fornece informações úteis aos programadores, juntamente com a especificação dos argumentos. O programador tem que estender o modelo de componente CH_Election para implementar a tarefa de seleção do CH requerida pelo modelo de armazenamento.

```
1  class Cluster_Formation {
2  public:
3    template <typename K, typename V>
4    void join(map<K, V> );
```

Código 4.5: Template do Componente Cluster_Formation

Uma vez que os CHs são eleitos, eles anunciam a decisão para a rede. Um nó decide juntar-se a um *cluster* com base em um critério de agrupamento. O componente Cluster_Formation contém a função "join(map <K, V>)", que assume como entrada um conjunto de pares chave-valor (map<K, V>) de anúncios de CHs. Suponha que o critério de formação de agrupamento estabelece que cada sensor irá seguir um CH com menor identificador (ID) que estiver mais próximo a ele. Por exemplo, suponha que a função " join ", executada pelo sensor s_i, receba como entrada um conjunto de anúncios de CHs, dado por $\{(s_j, 3), (s_k, 5)\}$, onde s_j, s_k são os sensores com IDs 3 e 5, respectivamente. A implementação da função deve escolher s_j como CH de s_i, dado que ele possui a menor identificação dentre os candidatos.

A comunicação entre os componentes do sistema é definida pelo componente coordenador, que também é responsável pelo fluxo de execução do sistema. O Código 4.6 ilustra de maneira simplificada o *template* do componente coordenador. Lembre-se de que o foco da arquitetura do RCBM está em validar os modelos gerados em ambientes de simulação. No RCBM, cada programa tem uma função "startSimulation()", que é chamado no início da simulação. Durante a execução da simulação, a comunicação entre componentes remotos (e sensores) é realizada através de troca de mensagens assíncronas. Assim, as funções "sendPkt " e " recv" definidas no componente do coordenador fornecem essas funcionalidades.

```
1  class Coordinator {
2  public:
3    void startSimulation();
4    virtual void recv (Packet *, Handler *);
5    void sendPkt(MsgID, WSN_Components_Message *);
6  }
```

Código 4.6: Template do Componente Coordenador

4.4 CAMADA DE IMPLEMENTAÇÃO DO RCBM

Esta seção apresenta detalhes de implementação que mostram como o RCBM foi usado para desenvolver dois modelos de armazenamento da classe de *repositórios*: LCA e LEACH. Esses estudos de caso mostram que a reutilização de componentes e *templates* de biblioteca traz benefícios para a produtividade do desenvolvimento de sistemas.

4.4.1 Implementação do Modelo LCA

O LCA (***L**inked **C**luster **A**lgorithm*) (Baker e Ephremides, 1981) é um modelo de armazenamento hierárquico destinado para redes pequenas. O critério para o agrupamento é baseado no identificador único (ID) associado a cada nó sensor. Durante a fase de eleição do CH, um nó com menor identificação entre seus vizinhos e que não tenha recebido um anúncio de CH, é eleito o líder do *cluster*. Após a fase de eleição, os sensores remanescentes juntam-se ao CH mais próximo. O modelo de componente pode ser usado para implementar o LCA estendendo os *templates* "CH_Election" e "Cluster_Formation".

Componente CH_Election . O Código 4.7 mostra a implementação simplificada da operação "selectCH" do modelo LCA.

```
1  template <class K, class V>
2  void CH_Election :: selectCH(map<K, V> Neighbors) {
3   int minNeighbors = compLib->MIN(Neighbors);
4
5   if ( getSensorId() < minNeighbors ) {
6    role = CH;
7    WSN_Components_Message param();
8    param.setId(getSensorId());
9    param.setDestination(Broadcast);
10   sendPkt(CH_ANNOUNCE, &param);
11  }
12  else
13   role = CM;
14 }
```

Código 4.7: Implementação do componente CH_Election

O desenvolvedor deve fornecer a implementação da função "selectCH" de acordo com o critério de seleção de CHs do modelo. A função "selectCH" opera em um conjunto de pares de chave-valor "Neighbors", isto é, os vizinhos de s_i que não foram atribuídos a um CH *(l.2)*. Inicialmente, a função de biblioteca MIN(Neighbours) *(l.3)* calcula o menor ID entre os vizinhos de s_i. Um sensor é selecionado como CH quando seu ID, dado por "getSensorId()", é menor que

o “minNeighbor” calculado *(l.5)*. Em seguida, ele transmite seu papel, atribuído como CH, para a rede *(l.7-10)*. Caso contrário, o papel do sensor é definido como um membro do *cluster* (CM) *(l.13)*.

Componente Cluster_Formation. No próximo passo, os nós que não foram selecionados como CHs devem se juntar a um *cluster*, conforme ilustra o Código 4.8. Um nó escolhe como CH o sensor com o menor ID entre seus vizinhos que foram eleitos como CHs *(l.3)*. Então, ele envia uma mensagem ACK para o CH escolhido como seu líder *(l.5-9)*.

```
1  template <class K, class V>
2  void Cluster_Formation :: join(map<K, V> knownCHs) {
3      int minCH = MIN(knownCHs);
4
5      WSN_Components_Message param ();
6      param.setId(getSensorId ());
7      param.setDestination(minCH );
8
9      sendPkt(ACK_CH_ANNOUNCE, &param);
10 }
```

Código 4.8: Implementação do componente Cluster_Formation

Componente Coordenador. O coordenador é responsável por juntar os componentes, coordenando as interações entre eles. As tarefas principais envolvem controle do *buffer* de entrada/saída e coordenação do fluxo de execução. O Código 4.9 ilustra a implementação das principais funções.

```
1  void Coordinator :: startSimulation(double T) {
2      setCurrentRound(SELECT_CH);
3      initTimer(T);
4  }
5
6  void Coordinator :: recv(Packet* pkt, Handler *) {
7    switch(param.getMsgId ()) {
8      case (CH_ANNOUNCE):
9          manageCHAnnounce(&param); break;
10     case (ACK_CH_ANNOUNCE):
11         manageACKCHAnnounce(&param); break;
12 }
13
14 void Coordinator :: sendPkt(MsgID ID, MsgParam* P) {
15   switch (ID) {
16     case (CH_ANNOUNCE):
17     case (ACK_CH_ANNOUNCE):
18         send(P->getPkt(), 0); break;
19 }
20
21 void Timer :: expire(Event*) {
22     if (getCurrentRound () == SELECT_CH) {
23         param=getCHParams ();
24         compCHElection->selectCH(&param);
25         setCurrentRound(JOIN_CLUSTER);
26     } else if (getCurrentRound () == JOIN_CLUSTER) {
27         param=getJoinParams ();
```

```
28          compClusterFormation->join(&param);
29          setCurrentRound(FINISH);
30      }
31  }
```

Código 4.9: Implementação do componente Coordenador

No LCA, as transmissões de dados seguem o esquema de escalonamento *round-robin* (fila circular), iniciando pelo sensor com o menor ID até o sensor com maior ID. Assim, cada nó configura seu escalonamento e a rodada atual no início da simulação LCA *(l. 2-3)*. Quando o temporizador expira *(l. 21)*, cada nó executa as tarefas para a rodada atual. Durante a primeira rodada, cada nó executa a função "selectCH()", que implementa a estratégia de eleição do CH e aguarda algumas unidades de tempo para iniciar a próxima rodada *(l. 23-25)*. Quando o temporizador expira, cada nó decide juntar-se ao *cluster*, executando a função "join" do componente Cluster_Formation *(l. 27-29)*.

4.4.2 Implementação do Modelo LEACH

O LEACH (***L**ow-**E**nergy **A**daptive **C**lustering **H**ierarchy*) (Heinzelman et al., 2000) é um modelo probabilístico de formação de *clusters* de sensores que estão à distância de um salto. O LEACH assume que todos os nós estão dentro do raio de comunicação uns dos outros. Um sensor é eleito como CH de acordo com uma certa probabilidade p. Os sensores remanescentes que não foram eleitos como CHs juntam-se ao CH que requer o menor consumo de energia para comunicação.

Componente CH_Election. O Código 4.10 apresenta a implementação simplificada do Componente "CH_Election".

```
1  template <typename T>
2  void CH_Election :: selectCH(map<K, V> P) {
3    double r = ((double) rand() / (RAND_MAX));
4    prob = getProbability(P);
5    double threshold = prob / ((1.0 - prob) * fmod(1.0, (1 / prob)));
6
7    WSN_Components_Message param ();
8    param.setId(getSensorId());
9
10   if (r < threshold)
11     sendPkt(CH_ANNOUNCE, &param);
12 }
```

Código 4.10: Implementação do componente CH_Election

O componente CH implementa a função "selectCH", que recebe uma probabilidade P como parâmetro de entrada *(l. 2)*. Primeiro, cada nó gera um número aleatório $X \in [0, 1]$ *(l. 3)*. Considerando *prob (l.4)* como a porcentagem desejada de CHs na rede, o sensor calcula um limiar $threshold$ para seleção do CH *(l. 5)*. Um nó se elege CH quando o valor calculado r é menor que o limite $threshold$, anunciando a decisão para a rede *(l. 10-11)*.

Componente Cluster_Formation. No próximo passo, os nós que não foram selecionados como CHs devem se juntar a um *cluster*. Cada nó escolhe como CH o sensor que requer o menor custo de energia de comunicação, com base na força de sinal recebida durante os anúncios dos CHs. O Código 4.11 ilustra a implementação simplificada do componente Cluster_Formation.

```
1 void Cluster_Formation :: Join(map<K, V> knownRSS){
2     double maxRSS = MAX(knownRSS);
3
4     WSN_Components_Message param ();
5     param.setId(getSensorId());
6     param.setDestination(maxRSS);
7     if (role == CM)
8      sendPkt(ACK_CH_ANNOUNCE, &param);
9 }
```

Código 4.11: Implementação do componente Cluster_Formation

Durante a fase de formação do *cluster*, o sensor s_i executa a função "Join", tendo como entrada um par chave-valor "knownRSS", que representa um conjunto de anúncios de CHs recebidos por s_i. Primeiro, cada nó calcula a força máxima de sinal recebido "knownRSS". O maior RSS é o CH que exige o menor consumo de energia para se comunicar. Assim, s_i define como CH o sensor com o maior valor de "knownRSS" *(l. 2)*, e envia uma mensagem de ACK *(l. 4-8)*.

Os dois estudos de caso apresentados nesta seção mostram que as instâncias compartilham os mesmos modelos de componentes. Esta abordagem promove a reutilização e o programador desenvolve algumas linhas de código de acordo com as especificidades de cada modelo. Além disso, foi implementado um terceiro modelo, o MAX-MIN (Amis et al., 2000), que será avaliado junto com os outros dois no estudo experimental [1].

4.5 ESTUDO EXPERIMENTAL

Esta seção mostra a avaliação de desempenho empírica do RCBM. Esta abordagem foi aplicada para apoiar a implementação dos protocolos LCA, LEACH e MAX-MIN no simulador de redes NS2 versão 2.35. Os algoritmos de agrupamento LCA e MAX-MIN seguem um critério de agrupamento baseado em atributo. LEACH segue um modelo probabilístico. Foram realizamos dois experimentos. No primeiro experimento, analisou-se a quantidade de reutilização de código nos sistemas, comparando os resultados do RCBM com os resultados reportados pela avaliação do CBCWSN (Amaxilatis et al., 2011b).

O objetivo do segundo experimento foi validar a corretude da implementação do MAX-MIN, aplicando-a ao mesmo cenário de avaliação e parâmetros relatados em Amis et al. (2000). O MAX-MIN é um modelo de formação de *clusters* baseado em atributos que difere de LCA e LEACH, relatados na Seção 4.4, porque os membros do *cluster* podem estar a múltiplos saltos do *cluster-head*. Além disso, o algoritmo de agrupamento envolve duas etapas de comunicação entre os nós sensores antes da eleição dos CHs. Mais especificamente, o algoritmo segue quatro etapas lógicas: *(i)* propagação dos maiores identificadores (IDs) dos sensores, *(ii)* propagação dos maiores identificadores (IDs) dos sensores, *(iii)* eleição de CHs, e *(iv)* formação dos *clusters*. Embora o fluxo de execução do algoritmo MAX-MIN difere dos fluxos de LCA e LEACH, algumas das etapas desses algoritmos são semelhantes. A separação do coordenador dos componentes de aplicação propostos pelo RCBM permitiu explorar essa similaridade para a reutilização do código. A seguir são apresentados os resultados dos experimentos.

[1] A implementação está disponível em `https://github.com/macarrero/Q2SWinet`.

4.5.1 Reuso de Código

O objetivo deste experimento é determinar o impacto da reutilização do código na implementação geral do sistema. Foi avaliado o modelo de componentes RCBM para apoiar a implementação dos sistemas LCA, LEACH e MAX-MIN, comparando os resultados com os relatados pelo trabalho CBCWSN (Amaxilatis et al., 2011b). No cenário empregado considerou-se uma região retangular de 1400m por 1000m, nós idênticos e estáticos, distância entre os nós em torno de 90m, com links simétricos, com raio de comunicação de 100m e protocolo MAC 802.11. Cada nó possui um GPS que informa sua posição sobre a região monitorada. Os parâmetros usados na simulação estão descritos na Tabela 4.1.

Tabela 4.1: Parâmetros da simulação

Parâmetro	**Cenário Sintético**
Quantidade de sensores	140 sensores
Área do ambiente	1400mX1000m
Raio de comunicação do rádio	100 metros
Tempo de simulação	40 minutos

A Tabela 4.2 lista o número total de linhas de código, linhas de código reutilizadas e a porcentagem de código reusado (diferença) dos modelos RCBM e CBCWSN. Em ambos os casos, as linhas de código foram obtidas a partir de programas em formato legível por humanos. Embora os princípios de projeto do RCBM e CBCWSN resultem em alguns componentes comuns (como a eleição de CH e a formação de *cluster*), os componentes do modelo RCBM são mais flexíveis porque não assumem um fluxo ou critério de execução fixo. Em vez disso, operações de coordenação simples são de responsabilidade do desenvolvedor. Como resultado, um número maior de modelos e projetos pode se beneficiar da *reutilização* de código promovida pelo RCBM. Isso é mostrado nos resultados experimentais. O MAX-MIN adota uma sequência distinta de passos lógicos quando comparada com as abordagens LCA e LEACH, que, por sua vez, têm algumas semelhanças entre si. Os resultados mostram que o RCBM é mais eficaz em termos de reutilização de código do que o CBCWSN para a implementação do MAX-MIN, enquanto que apresenta um resultado satisfatório de *reutilização* para os sistemas LCA e LEACH.

Tabela 4.2: Proporção de código reutilizado em cada modelo de sistema

Modelo de Sistema	**Modelo de Componente**	**Linhas de Código**	**Código Reusado**	**Diferença (%)**
MAX-MIN	RCBM	725	480	66 %
	CBCWSN (Amaxilatis et al., 2011b)	1140	259	23 %
LCA	RCBM	556	481	86 %
	CBCWSN (Amaxilatis et al., 2011b)	671	655	98 %
LEACH	RCBM	488	366	75 %
	CBCWSN (Amaxilatis et al., 2011b)	681	661	97 %

A eficiência do reuso que foram obtidos resultam do fato de que RCBM fornece uma biblioteca de componentes e fluxo de controle de execução que são *flexíveis*. O RCBM fornece um esqueleto de código genérico, que o desenvolvedor deve preencher suas lacunas em locais específicos do código. Grandes porções de código são reutilizadas a partir de componentes de biblioteca e o usuário precisa codificar algumas linhas, principalmente nos esqueletos das funções “selectCH” e “joinCluster”.

O CBCWSN segue uma abordagem diferente. Ele não fornece um esqueleto de código genérico que possa ser preenchido pelo usuário. Em vez disso, o desenvolvedor começa com um conjunto predefinido de implementações de componentes que compartilham muitas semelhanças com os modelos LCA e LEACH. Assim, é mais fácil estender o CBCWSN para modelos semelhantes ao LCA e LEACH, porém mais difícil de estendê-lo a outros modelos de armazenamento. Conforme visto na Tabela 4.2, o CBCWSN reduz significativamente o esforço para implementar os modelos LCA e LEACH. No entanto, o conjunto predefinido de componentes do CBCWSN tem diferenças significativas em relação ao projeto do sistema MAX-MIN, o que faz com que seu desenvolvimento seja muito custoso.

4.5.2 Validação da Implementação

O objetivo deste experimento é validar o sistema desenvolvido com o modelo de componentes RCBM. Em outras palavras, a finalidade é verificar se o código gerado por esta abordagem apresenta anomalias ou se a implementação resultante tem o mesmo comportamento que os relatados anteriormente na literatura. Portanto, foram analisadas o número de CHs gerados com a implementação do modelo MAX-MIN e LCA, comparando-se os resultados com os relatados no trabalho original do MAX-MIN (Amis et al., 2000). A simulação considerou uma região monitorada de $200X200$ metros quadrados com variações na densidade de nós da rede. Foram consideradas quatro variações de densidade, considerando redes com 100, 200, 400 e 600 nós sensores. Para cada densidade, gerou-se cinco cenários estáticos diferentes. O raio de comunicação do rádio de cada sensor no ambiente de monitoramento foi ajustado para 20 metros , e a distância máxima entre um membro do *cluster* até um nó CH foi ajustado para 2. Os parâmetros de configuração de simulação são mostrados na Tabela 4.3.

Tabela 4.3: Parâmetros da simulação

Parâmetro	**Valores**
Quantidade de nós	100, 200, 400 and 600 sensores
Área de Monitoramento	$200mX200m$
Raio de Transmissão	20 metros
d-distância até o CH (saltos)	2

Para validar a implementação realizada, foi comparado o impacto da densidade de nós da rede em relação ao número de CHs reportado no artigo do MAXMIN (Amis et al., 2000), com o que foi observado durante o desenvolvimento deste estudo de caso. O resultado da avaliação está ilustrado na Figura 4.3. Observe que os resultados experimentais relatados em Amis et al. (2000) apresenta um gráfico, identificado como Figura 8, que compara os sistemas MAX-MIN, LCA2, LCA e DEGREE. Este estudo irá comparar as implementações do MAX-MIN e do LCA2. O LCA2 é uma versão modificada do LCA original descrito na Seção 4.4.1, pois considera que os membros do *cluster* estão à uma distância de dois saltos do líder. No LCA original, a distância entre os membros e o CH é de 1 salto. A implementação do LCA2 seguiu as etapas descritas em Amis et al. (2000), com reuso de código do LCA original. Percebe-se que, devido à reutilização de código do modelo RCBM, concluiu-se com sucesso a implementação, simulação e validação do LCA2 em 2 dias, com pequenas mudanças no código do LCA original relatado na Seção 4.4.1. Os resultados alcançados, considerando MAX-MIN e LCA2, foram consistentes com os relatados no trabalho de Amis et al. (2000). Os valores originais não são plotados no gráfico porque os dados reais não foram disponibilizados em Amis et al. (2000). No entanto, analisando

o gráfico e os valores dos sistemas avaliados, percebe-se que os resultados são muito próximos aos relatados no trabalho original.

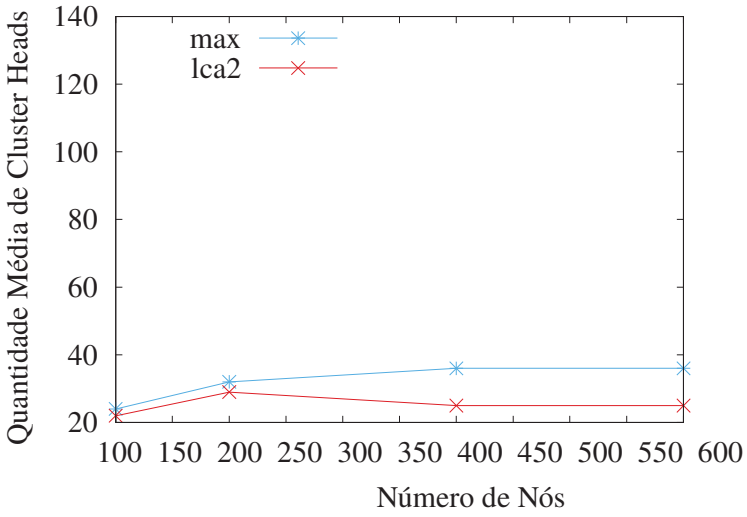


Figura 4.3: Impacto da densidade da rede no número de CHs

4.6 RESUMO

Este capítulo apresentou um modelo baseado em componentes reutilizáveis (RCBM) para apoiar e facilitar o desenvolvimento de sistemas armazenamento de dados para ferramentas de simulação. O RCBM aborda entidades de nível de armazenamento que compartilham conceitos e funcionalidades, que representam várias instâncias de sistemas de armazenamento em RSFFs. Essas funcionalidades compartilhadas formam os componentes do sistema. O RCBM considera três tipos de componentes: componentes de biblioteca, componentes de aplicação e o coordenador. Os componentes de biblioteca fornecem uma caixa de ferramentas, que pode ser usada para implementar componentes de aplicação, que são associados às entidades das RSSFs. O coordenador é responsável pelo fluxo de execução. Os componentes de biblioteca contém código pré-definidos. Os componentes de aplicação e o coordenador, por outro lado, possui um esqueleto que contém vários trechos de código já implementados que servem de base para o desenvolvimento da aplicação. O esqueleto leva em consideração algumas especificidades do simulador de rede. Em nossa implementação atual, usamos o simulador NS2.

Para validar o RCBM, foram desenvolvidos três estudos de casos, implementando os modelos LCA, LEACH e MAX-MIN. Os experimentos mostraram que a reutilização de código do modelo MAX-MIN é melhor no RCBM quando comparado com CBCWSN (Amaxilatis et al., 2011b), devido ao fato do controle de fluxo flexível do RCBM. Os resultados obtidos também mostram que o código gerado para os modelos MAX-MIN e LCA é compatível com a implementação MAX-MIN original (Amis et al., 2000), em relação ao número de agrupamentos formados. Esses experimentos mostraram que nossa proposta gera implementações que estão de acordo com as implementações originais dos modelos. O RCBM pode ser visto como um primeiro passo para um ambiente de programação para RSSFs, no qual o código dos componentes pode ser gerado a partir de primitivas de composição de alto nível.

5 UMA MÁQUINA DE ESTADOS PARA ESPECIFICAÇÃO DE CÓDIGOS DE SIMULAÇÃO PARA REDES DE SENSORES SEM FIO URBANAS

Os sistemas ubíquos de sensoriamento urbano enfrentam desafios relacionados à grande quantidade de sensores na rede e à dinamicidade de geração de dados. Individualmente, as funções de um sensor são acionadas como resposta a um evento, como a mudança do valor sensoriado ou o recebimento de uma mensagem. As máquinas de estados têm sido usadas para especificar estes sistemas. No entanto, a modelagem do processo colaborativo exige mudanças de estado dos sensores não apenas como resposta a um evento, mas também por uma condição lógica. Este capítulo propõe uma máquina de estados com dois tipos de transição: por evento e por lógica. O objetivo da modelagem com esta máquina é facilitar a implementação do código de simulação. Um estudo de caso que considera um modelo de armazenamento em sensores, desenvolvido com o apoio de um *framework* de componentes reusáveis, mostra uma correspondência entre a máquina proposta e o código, facilitando a sua implementação em ambientes de simulação.

5.1 INTRODUÇÃO E MOTIVAÇÃO

Os sistemas ubíquos de sensoriamento urbano são tipicamente sistemas orientados a eventos (Muñoz e Leone, 2017). A fim de oferecer serviços ubíquos ao cidadão e ao gestor público, as RSSFs podem ser usadas como infraestrutura para atender à grande demanda pelas aplicações de sensoriamento que exigirão o encaminhamento e o armazenamento de dados em diversos domínios. Dada a natureza reativa das RSSFs, a maioria dos simuladores existentes, dentre eles o NS2, OMNeT++ e TOSSIM, são baseados em eventos. É um modelo intuitivo de programação que associa eventos a ações, mas oferece um **baixo nível de abstração**. Assim, faz-se necessário o uso de ferramentas que reduzam a complexidade do desenvolvimento de sistemas ubíquos orientados a eventos, aumentado a produtividade e minimizando possíveis erros de projetos.

Para lidar com estes problemas, diversas técnicas têm sido desenvolvidas para reduzir o esforço e a complexidade do desenvolvimento de aplicações. O modelo de componentes de software orientado a serviços USEME propõe um ambiente de programação para implementar sistemas complexos a partir de serviços mais simples (Cañete et al., 2011). No entanto, o modelo proposto não é flexível para a criação de sistemas autônomos, como por exemplo, em aplicações que realizam a formação de agrupamentos na própria rede de sensores. Estratégias propostas por Kasten e Römer (2005); Glombitza et al. (2010); Cecílio e Furtado (2012) usam máquinas de estados para modelar sistemas orientados a eventos. No entanto, os estudos de caso mostram

modelos de máquinas de estados simples, o que não é a realidade encontrada em sistemas de armazenamento de larga escala.

As máquinas de estados (MEs) são uma abordagem usada para tratar a complexidade do desenvolvimento de programas orientados a eventos, possibilitando abstrair detalhes de implementação (Desai et al., 2013). Os estados são representações lógicas que descrevem o contexto atual de execução. As MEs são frequentemente usadas para especificar códigos para RSSFs, uma vez que os sensores, considerados individualmente, reagem a eventos como a obtenção de uma nova leitura ou o recebimento de uma mensagem. No entanto, sensores e outros dispositivos autônomos frequentemente realizam funções colaborativas, que exigem mudanças de estado dos dispositivos não apenas como resposta a um evento, mas também por uma condição lógica ou por um temporizador. Logo, a modelagem de sistemas com MEs torna-se mais complexa, dificultando sua implementação em simuladores orientados a eventos. Portanto, faz-se necessária uma abordagem de alto nível que associe de maneira clara modelos de máquinas de estados complexas, que tratam diferentes casos de mudanças de estados e que facilitem sua codificação.

Para reduzir o esforço e a complexidade do desenvolvimento de aplicações, Cañete et al. (2011) propõem um modelo de componentes de software orientado a serviços para desenvolver sistemas complexos a partir de serviços mais simples. No entanto, o modelo proposto não é flexível para a criação de sistemas autônomos, como por exemplo, em aplicações que realizam a formação de agrupamentos na própria rede de sensores. Estratégias propostas por Krämer et al. (2013) usam máquinas de estados para modelar sistemas orientados a eventos. No entanto, os estudos de caso mostram modelos de máquinas de estados simples, o que não é a realidade encontrada em cenários urbanos.

A máquina de estados proposta é usada para fazer a modelagem do detalhamento do fluxo de operações em um simulador baseado em eventos, criando dois tipos de transição: baseada em evento e baseada em lógica. Este detalhamento pode então ser utilizado para estruturar o código de simulação. Um estudo de caso descreve uma ME para o DCSSC (***D**istributed **C**lustering **S**cheme based on **S**patial **C**orrelation in WSNs*), um modelo apropriado para o sensoriamento urbano, que realiza agrupamentos explorando a similaridade de dados (Le et al., 2008). A ME foi implementada usando o *framework* RCBM, que oferece um conjunto de bibliotecas e componentes reusáveis para desenvolver sistemas de simulação para o NS2. O código do DCSSC desenvolvido com o apoio do *framework* RCBM mostrou uma correspondência direta entre os tipos de estados propostos na ME e o seu detalhamento no programa de simulação. Além disso, os resultados mostram uma reutilização de código de até 71.6%, o que demonstra o potencial da proposta para facilitar o desenvolvimento de novas aplicações para RSSFs e outras de natureza reativa e colaborativa, como os sistemas ubíquos e pervasivos.

5.2 TRABALHOS RELACIONADOS

Componentes de software são uma abordagem usada para possibilitar o reuso de software por futuros desenvolvedores. No entanto, os componentes oferecem uma visão *estática* do sistema. As máquinas de estados, por outro lado, tratam da parte *dinâmica* do sistema, descrevendo o comportamento e as interações entre os componentes do sistema. Em uma revisão literária sobre técnicas de desenvolvimento de sistemas para as redes de sensores (Malavolta e Muccini, 2014), verificou-se a inexistência de uma abordagem que trate da modelagem e implementação de uma maneira sistemática, requisitos necessários para atender à grande demanda por serviços ubíquos de grande escala.

O trabalho proposto por Krämer et al. (2013) descreve uma biblioteca de desenvolvimento de aplicativos para RSSFs orientada a máquina de estados. Um programa pode ser decomposto em módulos, cada um executando sua própria máquina de estados. A coordenação de vários módulos é feita pela máquina de estados do módulo principal. Embora a biblioteca promova modularidade e reusabilidade de código, a análise de desempenho da biblioteca não demonstra que sua arquitetura seja capaz de implementar sistemas escaláveis e autônomos, requisitos desejados para aplicações urbanas.

Um modelo de máquina de estados que busca garantir confiabilidade e desempenho de aplicações para RSSFs é descrito em Cecílio e Furtado (2012). No modelo proposto, existem duas classes de estados: estados que tratam de requisitos funcionais e estados que tratam de requisitos não-funcionais. Por exemplo, coletar e encaminhar dados é um requisito funcional e reencaminhar dados quando a perda de pacotes for maior que 1% é um requisito não-funcional. No entanto, o foco deste trabalho está em direção oposta ao dos autores citados: esta abordagem leva em consideração os requisitos funcionais encontrados em RSSFs urbanas enquanto a proposta deles lida com os requisitos não-funcionais de ambientes industriais.

As estratégias usadas por Tokenit Taherkordi et al. (2015) integram um ambiente de modelagem e implementação orientado à máquina de estados. O *framework* de desenvolvimento leva em conta as restrições de recursos encontradas em sistemas embarcados. O modelo define uma máquina de estados como um conjunto de atividades, que são operações executadas por um estado. As mudanças de estado ocorrem por eventos de *timers* ou eventos assíncronos. A detecção de um movimento, por exemplo, é um evento assíncrono. A máquina de estados é descrita no formato XML e o compilador do *framework* gera automaticamente código para a plataforma de execução do sistema operacional Contiki (Dunkels et al., 2004). Contudo, a avaliação de desempenho não aborda aspectos sobre a escalabilidade do sistema, característica desejada em aplicações de sensoriamento urbano.

5.3 UMA MÁQUINA DE ESTADOS ORIENTADA A EVENTOS

Em RSSFs, os sensores podem desempenhar diversas funções, desde a participação no roteamento de mensagens (Gielow e dos Santos, 2009) em uma comunicação multi-salto até a atuação como um repositório de dados coletados do ambiente. Estas funcionalidades são em geral acionadas através do recebimento de mensagens, que podem ser modeladas como transições de uma máquina de estados (ME) orientada a eventos. No entanto, existe uma dicotomia entre o entendimento intuitivo de uma transição em uma máquina de estados e o seu funcionamento em uma RSSF, uma vez que a máquina de estados deve representar a funcionalidade de cada sensor *individualmente*. Logo, não há necessariamente uma correspondência entre o estado do sensor que envia uma mensagem e o estado do sensor que a recebe. Ou seja, um sensor pode enviar uma mensagem a partir de um estado e_1, mas o sensor que a recebe pode estar em um estado e_2, que não é necessariamente o próximo estado do sensor origem da mensagem. Assim, a mudança de estado por evento pode não estar necessariamente relacionada ao estado do sensor emissor e o envio de mensagens não necessariamente determina uma mudança de estado. Em suma, não há correspondência direta entre transições na máquina e a comunicação entre sensores. No entanto, as mudanças de estado do sensor origem e destino são representadas na mesma máquina, que é executada em todos os sensores, como ilustrado pela Figura 5.1(b). Experiências anteriores de desenvolvimento de código de simulação para RSSF (Furlaneto et al., 2012; Gonçalves et al., 2012), mostraram que esta dicotomia dificulta o desenvolvimento de programas.

O objetivo da ME proposta neste capítulo é oferecer suporte para o desenvolvedor especificar o comportamento do sistema antes de iniciar a implementação, através da criação de dois tipos de transição (por evento e lógica), além de uma notação para o envio de mensagens. A ME combina elementos de máquinas de estados finitos, como estados e transições, com princípios de programação reativa dos simuladores de RSSFs, que associam eventos à um conjunto de ações. As ferramentas visuais de modelagem de máquina de estados que seguem o padrão da *UML StateChart*, podem ser usadas para criar as MEs desenvolvidas neste capítulo. Um exemplo desta máquina está ilustrada na Figura 5.1(a). Cada estado possui um nome, como `Ini` e `Wait_First_Sensor_ID`, e pode possuir uma anotação (entre colchetes) sobre o envio de mensagens ou inicialização de temporizadores. As transições podem ser por evento (representadas em azul) ou lógica (representadas em vermelho):

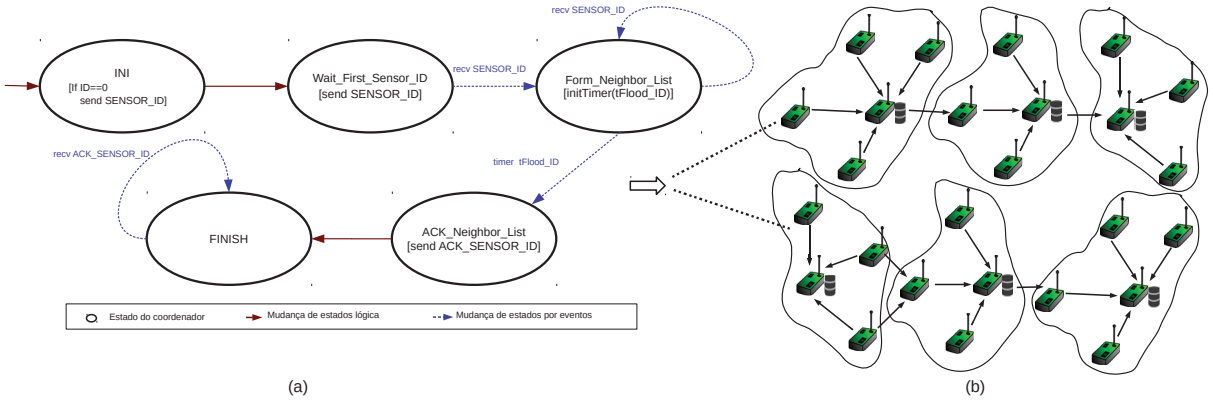


Figura 5.1: Um modelo de máquina de estados para descoberta de vizinhos

- Mudança de estado por evento: são aquelas que ocorrem quando um sensor recebe algum tipo de mensagem ou então após o tempo de expiração de um temporizador.
- Mudança de estado lógica: é realizada com base no resultado de uma computação.

O exemplo da Figura 5.1(a) descreve o algoritmo de descoberta de vizinhos por inundação na rede. No estado inicial `INI`, o sensor com identificador (`ID`) zero envia a mensagem `SENSOR_ID` para seus vizinhos. O sensor zero e os demais sensores realizam uma mudança lógica do estado `INI` para o estado `Wait_First_SENSOR_ID`. Os sensores que receberem a primeira mensagem `SENSOR_ID` armazenam o `ID` contido na mensagem e enviam seu identificador para seus vizinhos. Em seguida, realizam uma mudança por evento do estado `Wait_First_SENSOR_ID` para o estado `Form_Neighbor_List`. No estado `Form_Neighbor_List`, os sensores continuam armazenando o `ID` dos demais vizinhos durante um certo tempo t_{Flood_ID}. Quando o tempo expirar, os sensores fazem uma mudança para o estado `ACK_Neighbor_List` e enviam uma mensagem de `ACK` para os sensores conhecidos. Após o envio da mensagem, os sensores realizam mudança lógica para o estado `FINISH`, armazenando as mensagens de `ACK` recebidas. Ao final da inundação, todos os sensores conhecem seus vizinhos.

Este é um exemplo bastante simples, no qual algumas características de RSSFs não ficam tão evidentes. Elas advêm da natureza distribuída das RSSFs, na qual cada sensor executa a máquina de estados de forma independente. Dessa forma, a modelagem destes sistemas, que frequentemente realizam funções colaborativas assíncronas, na qual múltiplas máquinas podem estar em estados distintos, não é uma tarefa trivial. A próxima seção apresenta um estudo de caso que salienta estas características e mostra a efetividade da máquina proposta para a representação do modelo.

5.4 ESTUDO DE CASO

Um estudo de caso foi desenvolvido para demonstrar o uso da máquina de estados em outros contextos. Como o foco é em aplicações urbanas, foi desenvolvido um estudo de caso a partir do modelo de armazenamento em RSSFs chamado DCSSC (Le et al., 2008). O algoritmo proposto constrói e mantém os *clusters* de forma distribuída e dinâmica. A abordagem de agrupamento por similaridade de dados em redes de larga escala torna o DCSSC um modelo apropriado para o sensoriamento urbano.

5.4.1 O modelo DCSSC

No modelo DCSSC, cada sensor comunica-se apenas com os sensores que estão à distância de um salto e todos têm os dados relativos ao nível de energia de seus vizinhos. Esta informação é enviada por meio de mensagens do tipo *HELLO* trocadas periodicamente. O modelo de rede consiste em uma estação-base e N sensores. São atribuídos estados a todos os dispositivos, que determinam seu papel na rede. No início, os sensores estão no estado `INI` e ao final da fase de construção de agrupamentos eles estarão em um dos seguintes estados: `CH` (*cluster-head*), `GW` (*gateway*), `EXT` (*cluster-extend*) ou `MEM` (*member*). Os sensores com os estados `CH`, `GW` e `EXT` ficam nesse estado até o fim da fase de construção e são chamados de nodos de *backbone*. Existem também estados temporários, como o `INI`, `GWR` (*gateway-ready*) e CHC (*cluster-head candidate*).

As mensagens de formação de agrupamento incluem sempre a média de valores das leituras aferidas no tempo que precede o envio da mensagem. Baseado no tipo da mensagem, o receptor irá mudar seu estado, criar uma nova mensagem e propagar para seus vizinhos. A nova mensagem também deve conter o identificador (ID) do emissor original da mensagem recebida. A Figura 5.2 ilustra as fases da simulação que inicia com a construção de agrupamentos, determinada pelos passos que seguem:

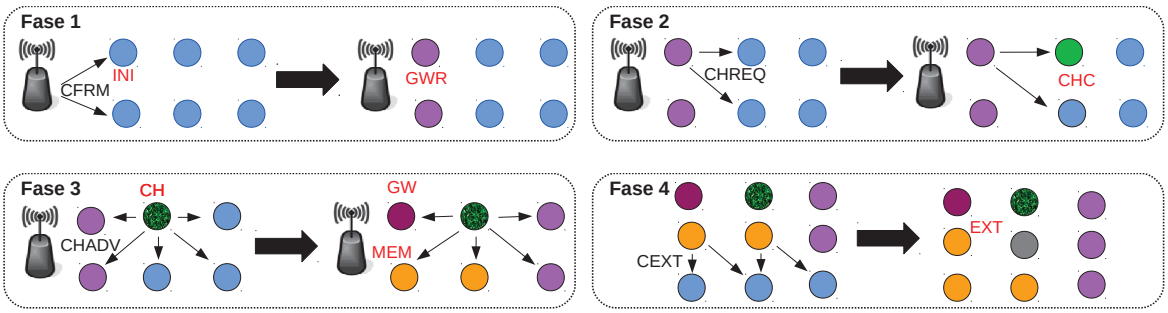


Figura 5.2: Fases do algoritmo DCSSC

Fase 1
A estação-base inicia a formação de agrupamentos fazendo *broadcast* de uma mensagem do tipo `CFRM` (*Cluster Formation Message*). Cada sensor que recebe essa mensagem e está no estado INI muda seu estado para `GWR`, criando dois *timers*, t_{req} e t_{wait}, escolhidos aleatoriamente. Quando o tempo definido por t_{req} expira, sensores no estado `GWR` fazem o envio de uma mensagem `CHREQ` (*Cluster Head Request*) para seus vizinhos a fim de encontrar um *cluster-head*.

Fase 2
Quando um sensor `INI` recebe a mensagem `CHREQ`, calcula sua similaridade com o sensor `GWR`

que enviou a mensagem. Os sensores que forem altamente correlacionados mudam seu estado para CHC (candidatos a *cluster-head*). Os nodos CHC criam um novo *timer* t_{adv}, baseado no seu nível relativo de energia. Um sensor CHC se declara como CH fazendo *broadcast* da mensagem CHADV (*Cluster Head Adverstisement*) quando seu timer t_{adv} expirar. Como t_{adv} é inversamente proporcional ao seu nível de energia, sensores com maiores níveis de energia têm maior chance de serem selecionados como líder.

Fase 3
Quando algum sensor em um estado temporário (INI, GWR, CHC) recebe uma mensagem CHADV, ele calcula a similaridade com o sensor que enviou a mensagem. Se eles forem altamente correlacionados, o receptor se torna membro (estado MEM) do *cluster* formado por aquele CH. Senão, ele vai para o estado GWR, seguindo os mesmos passos definidos anteriormente para um sensor no estado GWR. Todo sensor altamente correlacionado compara o ID da fonte original da mensagem recebida com o seu próprio ID. Se forem iguais, ou seja, o CHADV recebido foi uma resposta ao seu próprio CHREQ, o sensor no estado GWR altera seu estado para GW. Quando o *timer* t_{wait} expira, o sensor no estado GWR muda seu estado para CHC se não houver nenhuma mensagem CHADV proveniente dos seus vizinhos.

Fase 4
Os sensores nos estados GW e MEM propagam a mensagem de formação criando e fazendo *broadcast* de mensagens do tipo CEXT (*Cluster Extend*), que incluem a média das leituras do CH que originou a mensagem, e não do sensor atual. Após receber uma mensagem CEXT, todo sensor em estado temporário (INI, GWR, CHC) calcula a sua similaridade com relação ao CH. Se forem altamente correlacionados, ele integra o *cluster* formado e se torna um membro (estado MEM), fazendo com que o transmissor da mensagem CEXT vá para o estado EXT. Caso não seja altamente correlacionado, ele vai para o estado GWR e repete os passos atribuídos a ele.

O modelo DCSSC prioriza como representantes de agrupamentos os sensores com maior nível relativo de energia dentre os que apresentam leituras similares, prolongando a vida útil da rede. Os nodos com os papéis de CH, GW e EXT (nodos de *backbone*) têm a função de coletar dados das leituras dos sensores, que enviam suas informações através do *backbone* a cada intervalo de tempo segundo o esquema de escalonamento Round-Robin (fila circular). As leituras recebidas são armazenadas no CH, e podem ser comprimidas a fim de reduzir o espaço necessário de armazenamento. Os dados agregados são transmitidos do CH para o GW do respectivo *cluster*, que encaminha para um sensor vizinho que pertença a outro *cluster*. Dessa forma os dados trafegam na rede através de nodos intermediários, até chegar à estação-base, com menor custo de comunicação.

5.4.2 A Máquina de Estados do modelo DCSSC

A máquina de estados foi aplicada para descrever a especificação formal da coordenação do fluxo de execução do modelo DCSSC. O fluxo de execução do DCSSC segue uma máquina de estados como representado na Figura 5.3(b). Todos os nodos iniciam no estado INI, e modificam seu estado ao longo da execução. Neste exemplo, as características distintas da máquina proposta ficam mais evidentes. Por exemplo, todos os sensores iniciam no estado INI e a mensagem CFRM é transmitida por *broadcast* pela estação base. Aqueles que a recebem passam para o estado GWR, que esperam por um tempo aleatorizado t_{req} para passar para o estado TREQ e enviar uma mensagem CHREQ. Os sensores que recebem esta mensagem estão no estado INI e passam ao estado selectCH, no qual a similaridade de dados com o sensor emissor é calculado para

determinar se ele passa ao estado `CHC`. Caso as condições para mudança de estado para `CHC` não sejam satisfeitas, o sensor retorna ao estado `INI` para que possa receber mensagens de outros sensores.

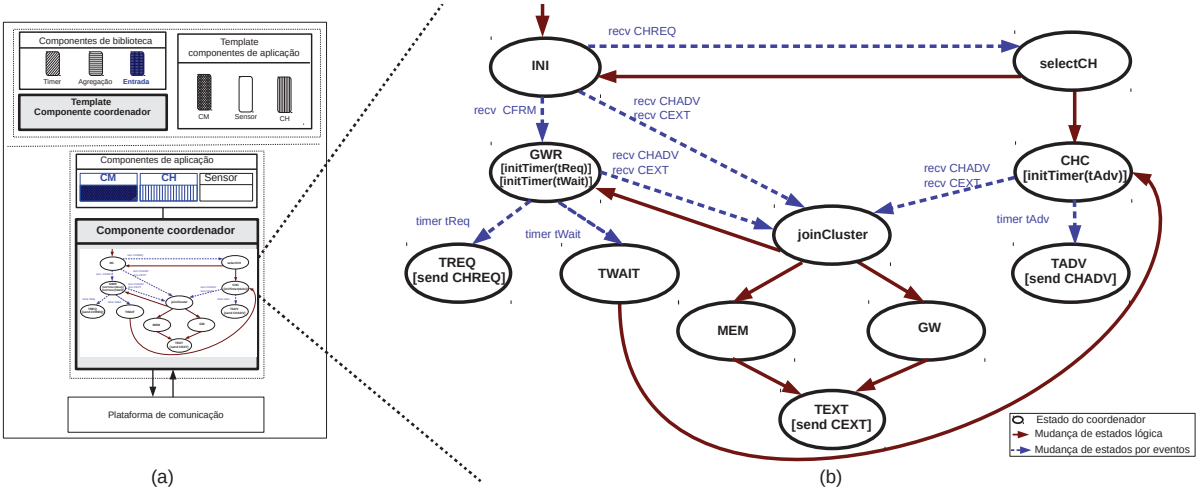


Figura 5.3: Visão do *framework* RCBM e da máquina de estados do DCSSC

A próxima seção descreve a implementação da máquina de estados do modelo DCSSC. A Figura 5.3(a) ilustra o RCBM, como visto em detalhes no Capítulo 4, descreve uma metodologia para o desenvolvimento de sistemas de armazenamento para RSSFs orientado a componentes. O modelo associa a cada entidade do modelo um conjunto de componentes que implementam as funcionalidades comuns aos modelos. Por exemplo, para a máquina de estados ilustrada pela Figura5.3(b), as entidades são os próprios sensores, os agrupamentos formados por *cluster members*(CM) e os líderes do grupo (CH).

5.5 IMPLEMENTAÇÃO

A Figura 5.3(a) representa uma instância da arquitetura do metamodelo RCBM, com base na implementação do modelo DCSSC. Os componentes em preto foram utilizados conforme fornecido, e os destacados na cor azul são aqueles que foram modificados e/ou criados para o desenvolvimento do DCSSC. A plataforma de comunicação escolhida foi a ferramenta de simulação *Network Simulator 2* (NS2). Percebe-se que a parte da especificação, que contém os *templates* e componentes de biblioteca, praticamente não foi alterada. A única modificação foi a inclusão de um novo componente de biblioteca, que inicializa as leituras dos sensores.

O *framework* RCBM já possui a definição de interfaces de funções para os componentes CH e CM, nos quais os estados `selectCH` e `joinCluster` da Figura 5.3(b), foram adaptados para o modelo DCSSC. No entanto, o desenvolvimento do código de coordenação entre os componentes é responsabilidade do usuário. A partir da máquina de estados ilustrada na Figura 5.3(b), o componente coordenador, que implementa os diversos estados e transições do modelo RCBM, foi desenvolvido através de duas funções principais no NS2: `recv` e `TimerHandle`. A função `recv` é responsável pelo recebimento de pacotes pelos sensores e trata essas mensagens. É a parte do código que faz as transições do estado do tipo "mudança por evento" e também faz a chamada para os componentes externos. O `recv` é composto por um comando *switch*, com vários *cases*, cada um equivalente a um estado que pode ser atingindo pelas mudanças de estado por evento. Cada mensagem possui um ID, e dentro dela existem alguns parâmetros como o ID do emissor, suas coordenadas e leituras, além do ID do emissor da última mensagem recebida

pelo sensor. A função `TimerHandle`, é executada quando existe uma transição ativada quando um *timer* expira e uma nova rodada (ou *round*) é iniciada. O `TimerHandle` cria pacotes e os envia conforme o tipo da rodada atual. O Código 5.1 ilustra parte da implementação da função `recv`[1] da implementação do DCSSC.

Ao receber um pacote, é feita uma comparação do tipo da mensagem recebida (linha 5). Caso seja um `CFRM` (linha 6) e o papel do sensor for `INI`, o dispositivo muda para o papel `GWR`. É inicializado o *timer* `Treq`, que ao expirar iniciará uma nova rodada TREQ (linhas 8 a 11) através da atribuição à variável `round`, que será tratada pela função `TimerHandle`. No modelo RCBM existe um componente de biblioteca que faz a implementação dos *timers*, o que facilita o tratamento de mudanças de estados lógicas cuja execução é disparada após determinado intervalo de tempo. A variável `lastMsg` armazena o tipo da última mensagem recebida (linha 7 e linha 16). Se a mensagem recebida foi um `CHREQ` (linha 15) e o papel do sensor for `INI`, o sensor calcula a similaridade com a leitura (linhas 19 a 22). Em seguida, a função `SelectCH` do componente CH é chamada (linhas 23 a 26). Trechos de código similares são executados para os outros estados.

```
1  void WSN_ComponentsAgent::recv(Packet* pkt, Handler *) {
2    SensorDataParams sp;
3    WSN_Components_Message param = pkt;
4
5    switch(param.getMsgId()) {
6      case(CFRM): {
7        lastMsg = CFRM;
8        if ( compSensor->role == INI ) {
9          compSensor->role = GWR;
10         TimerTreq();
11         round = TREQ;
12       }
13     }
14     break;
15     case(CHREQ): {
16       lastMsg = CHREQ;
17       if ( compSensor->role == INI ) {
18         compSensor->pID = param.getHdrCmn()->prev_hop_;
19         sp.reading = param.getHdrWsnComp()->
20                      msgParams.msgParam[0].reading;
21          double s1 = compSensor->getReadings();
22          double similaridade = fabs(s1-sp.reading);
23          map<string, double> sim;
24          sim.insert(pair<string, double>("K", similaridade));
25          compCH->SelectCH(sim);
26       }
27   }
28     break;
29      (...)
30   }
```

[1]O código completo está disponível em `https://github.com/macarrero/SBCUP2018`.

```
31 }
```

Código 5.1: Função `recv` do coordenador

A função `TimerHandle` é executada quando existe a transição do tipo "mudança de estado lógica" e é ativada quando um *timer* expira e uma nova rodada (ou *round*) é iniciada. O `TimerHandle` cria pacotes e os envia conforme o tipo da rodada atual. Um trecho do código dessa função é apresentado no Código 5.2.

```
1  void WSN_ComponentsAgent::TimerHandle(RoundCmd roundCmd) {
2      SensorDataParams sp;
3      MsgParam newp;
4      WSN_Components_Message param( getNewPkt() );
5      param.setId (compSensor->getSensorId());
6      sp.id = param.getId ();
7      sp.coordX = compSensor->getX();
8      sp.coordY = compSensor->getY();
9      sp.coordZ = compSensor->getZ();
10
11     switch (roundCmd) {
12         case TREQ: {
13             param.setMsgId(CHREQ);
14             sp.reading = compSensor->getReadings();
15             newp.msgParam[0] = sp;
16             SendPkt(CHREQ, &param);
17         }
18         break;
19         case TADV: {
20             param.setMsgId(CHADV);
21             sp.pID = compSensor->pID;
22             newp.msgParam[0] = sp;
23             SendPkt(CHADV, &param);
24         }
25         break;
26          (...)
27 }
```

Código 5.2: Função `TimerHandle` do coordenador

No início da função são inicializados os parâmetros necessários para o envio do pacote (linhas 2-9). Em seguida, é feito um *switch* da rodada atual (linha 11). Caso a rodada seja `TREQ` (linha 12) é criada uma mensagem do tipo `CHREQ`, com os parâmetros relacionados ao sensor (`ID`, coordenadas, leitura), e o pacote é enviado (linhas 13 a 16). O código para o caso `TADV` é similar, uma mensagem do tipo `CHADV` é criada com os mesmos parâmetros da mensagem `CHREQ` e é enviada (linhas 20 a 23). O tratamento dos outros tipos de rodada ocorre de forma análoga.

Analisando o código do NS2, verifica-se que há uma correspondência direta entre os tipos de transição propostos na máquina de estados e o seu detalhamento no componente coordenador do programa de simulação. Esta distinção conceitual de tipos de transição facilita o desenvolvimento de código, mantendo a reutilização promovida pelo *framework* RCBM, como mostram os experimentos detalhados na próxima seção. É importante ressaltar que o componente

coordenador deve ser implementado pelo programador e que sua função é fazer a junção dos componentes, coordenando as interações entre eles. O RCBM é mais flexível do que modelos baseados em componentes anteriores para RSSFs, porém possui como limitação a falta de uma especificação formal para a coordenação do fluxo de execução dos componentes. A máquina de estados proposta na Seção 5.3 supre esta deficiência, com uma proposta para especificar o fluxo de operações do coordenador, facilitando sua implementação.

5.5.1 Avaliação

Essa seção mostra a avaliação do sistema DCSSC, desenvolvido a partir de técnicas que integram reuso de componentes de software e máquina de estados que descrevem o comportamento dinâmico do sistema. Foram realizados dois experimentos. No primeiro foi usada a métrica de contagem de linhas de código (LOC) para analisar a porcentagem do código que foi possível reutilizar do modelo RCBM. No segundo, analisou-se a métrica *Coupling between object classes* (CBO), para avaliar o grau de acoplamento entre os componentes do sistema.

Reutilização de código

Para analisar a eficácia do modelo RCBM durante o desenvolvimento de código, foram contabilizadas as linhas de código necessárias para implementar o modelo. A Tabela 5.1 apresenta o número total das linhas de código, quantas delas foram criadas, quantas foram reutilizadas e a porcentagem equivalente à reutilização. Para implementar o DCSSC, foi adicionada uma nova biblioteca ao RCBM, que inicializa as leituras dos sensores. O número de linhas equivalentes à implementação do componente de entrada da biblioteca foi inserido na contagem `A` como linhas novas e na contagem `B` como linhas reutilizadas. Considerando que o componente não existia no modelo RCBM e foi criado para o desenvolvimento deste estudo, seu código pode ser classificado como novo. Entretanto, levando-se em consideração implementações futuras, esse componente pode ser considerado reutilizável, justificando as linhas de sua implementação inseridas como linhas reutilizadas.

Tabela 5.1: Proporção de linhas reutilizadas do RCBM

Contagem	**Total de linhas**	**Linhas reutilizadas**	**Linhas novas**	**Porcentagem de reuso**
A	1333	905	428	67,9%
B	1333	954	379	71,6%

Métrica CBO

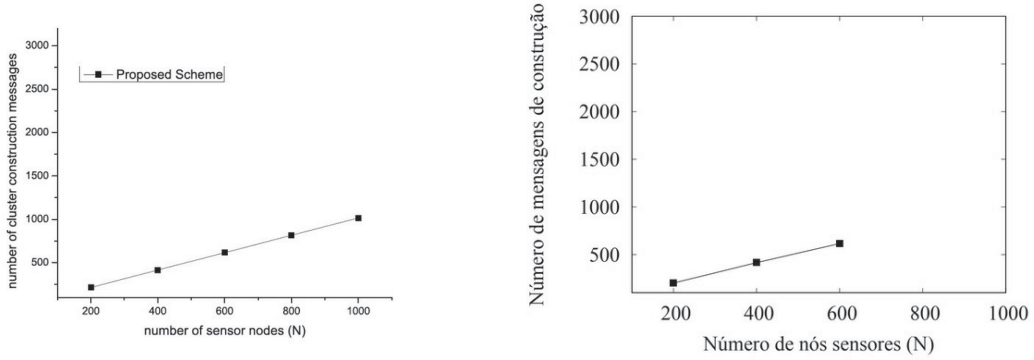
Na avaliação do coeficiente de dependência entre os componentes foi aplicada a métrica *Coupling between object classes* (`CBO`), cujo valor está entre 0 e 1. Quanto mais próximo de 1 for o `CBO` de um determinado componente, mais dependente ele é de outras classes. O resultado da avaliação está apresentado na Tabela 5.2. Percebe-se que o coordenador é altamente dependente de outros componentes, o que é esperado uma vez que ele é o responsável por fazer a junção de todos os componentes. Os componentes de biblioteca por sua vez não são dependentes de nenhum outro componente, e isso garante que sua utilização é completamente independente da implementação do usuário. Os outros componentes utilizados apresentam um baixo grau de acoplamento, e isso favorece a depuração do código e a correção de erros nos módulos, além de facilitar a inclusão de novos componentes ao RCBM.

Tabela 5.2: Análise da métrica CBO para cada componente do modelo DCSSC

Componente	Coordenador	Componentes de biblioteca	Sensor	CM	CH
CBO	1	0	0,33	0,33	0,33

Validação da Implementação

Conforme já mencionado, uma das dificuldades para a reutilização de código em modelos de armazenamento para RSSFs é a falta de códigos disponibilizados à comunidade. O modelo DCSSC é um dos que apresenta essa falha. Sua implementação não foi fornecida pelos autores do artigo, portanto a comparação do código original com o código gerado não foi possível. Para validar a implementação realizada, foi comparado o número de mensagens necessárias para a formação dos *clusters* demonstrado no artigo com o que foi observado neste estudo. O objetivo é verificar se o código implementado possui o mesmo comportamento do que o que foi reportado no artigo do modelo DCSSC (Le et al., 2008). Esse aspecto foi escolhido por ser o diferencial do DCSSC em relação a outros modelos similares que foram analisados no artigo original (Le et al., 2008). O DCSSC executa o algoritmo de formação de *clusters* com um número linear de mensagens, fato que deve também ser observado na implementação desse estudo de caso.



(a) Reportado em Le et al. (2008) (b) Implementação no RCBM

Figura 5.4: Número de sensores versus quantidade de mensagens de formação

Para a execução dos experimentos, foram feitas execuções com 200, 400 e 600 nodos, considerando uma área monitorada de 1400m X 1000m, e o alcance dos sensores em 100m. Foi utilizado como parâmetro um limiar de 20%. O cenário foi adaptado, já que os autores do artigo original não disponibilizaram os arquivos de cenário utilizados. Os resultados originais estão representados na Figura 5.4(a), e os obtidos durante experimentos são retratados na Figura 5.4(b).

Nota-se pela comparação dos gráficos que a implementação obteve o resultado esperado, ou seja, é condizente com o encontrado em Le et al. (2008). Os valores originais não foram mostrados no artigo, sendo reportados apenas em formato de gráfico, e por isso não foram citados. Observa-se o crescimento linear do número de mensagens conforme a densidade da rede aumenta em ambas as implementações.

5.6 RESUMO

Este capítulo apresentou uma proposta de máquina de estados para fazer o detalhamento do fluxo de operações em um simulador baseado em eventos. Através de um estudo de caso e a sua codificação no simulador NS2, foi mostrado que há uma correspondência direta entre a máquina proposta e o programa desenvolvido. Os experimentos demonstram que o percentual de reutilização de código utilizando o RCBM para a implementação do DCSSC foi de até 71,6%, considerando o componente de biblioteca como reutilizável, e de até 67,9%, considerando o componente de biblioteca criado como linhas de código novas. Os valores são significativos considerando a complexidade dos códigos de simulação para redes de sensores sem fio.

Experiências anteriores com a implementação *from scratch* de modelos de armazenamento em redes de sensores sem fio possibilitou-nos perceber o aumento significativo da facilidade do processo de desenvolvimento. Com o RCBM, os componentes e suas funcionalidades estão bem definidos, o que torna o código mais modular e claro.

6 SLEDS: UMA LINGUAGEM ESPECÍFICA DE DOMÍNIO PARA ARMAZENAMENTO CENTRADO EM DADOS EM REDES DE SENSORES

Os requisitos de dinamicidade das redes de sensores urbanas geram novos desafios para o desenvolvimento de modelos de gerenciamento e armazenamento de dados. As técnicas de componentes de software permitem que os desenvolvedores criem sistemas de software a partir de componentes reutilizáveis, compartilhando uma interface comum. Além disso, o desenvolvimento de aplicações de redes de sensores urbanas beneficiariam-se muito com a existência de um ambiente de programação dedicado. Este capítulo propõe SLEDS, uma linguagem específica de domínio para armazenamento centrado em dados em redes de sensores sem fio. A linguagem inclui primitivas de composição de alto nível, para promover um fluxo de execução de coordenação flexível e interação entre os componentes. É apresentada a especificação da linguagem, bem como um estudo de caso da coordenação de armazenamento de dados em redes de sensores. A especificação atual da linguagem gera código para o ambiente de simulação NS2. O estudo de caso mostra que a linguagem implementa um modelo flexível, que é geral o suficiente para ser usado em uma ampla variedade de aplicativos de redes de sensores.

6.1 INTRODUÇÃO E MOTIVAÇÃO

As redes de sensores usadas no monitoramento de áreas urbanas são geralmente densas. Eles são compostos de milhares de dispositivos que se comunicam via rádio e têm recursos limitados para processamento e armazenamento de dados. Existem três categorias de modelos de armazenamento de dados para RSSFs: local, externo e agrupada em *repositórios*. As categorias local e externa armazenam dados detectados no dispositivo sensor e em um dispositivo externo com mais recursos (geralmente chamado de estação base), respectivamente.

Essas categorias não são apropriadas para redes densas. Isso ocorre porque o armazenamento local exige que todos os dispositivos sejam contatados durante a disseminação de consultas, o que pode resultar no baixo desempenho do tempo de resposta. Por outro lado, o armazenamento externo requer que os sensores enviem periodicamente suas leituras para a estação base, o que pode causar tráfego desnecessário de mensagens. A abordagem agrupada em *repositórios*, por outro lado, combina as duas técnicas, elegendo um subconjunto de sensores para atuar como representantes de um conjunto de dispositivos, que armazenam suas leituras. Nesse modelo, grupos de sensores compõem agrupamentos, representados por líderes (CHs). Assim, para responder a consultas, apenas os CHs precisam ser contatados, fornecendo escalabilidade para redes densas, como as RSSFs urbanas.

A validação dos modelos propostos geralmente envolve programá-los em um ambiente de simulação, dados os custos e as dificuldades de implantar redes tão grandes em ambientes reais. Os simuladores NS2, NS3 e OMNeT++ estão entre os ambientes de simulação usados para RSSFs. Ao desenvolver trabalhos anteriores sobre modelos agrupados em *repositórios*, notamos que: *(1)* a maioria dos programas é desenvolvida a partir do zero e há pouco ou nenhum suporte para reusabilidade de código; e *(2)* existem semelhanças entre os modelos no fluxo de atividades, que podem ser modelados como máquinas de estado. Uma solução para estes problemas foi a elaboração de um modelo baseado em componentes para RSSFs, descrito no Capítulo 4, e uma máquina de estados para formalizar a interação entre os componentes (Carrero et al., 2018a). Embora a máquina de estados auxilie a especificação do fluxo de atividades, o programador ainda é responsável pelo desenvolvimento do código especificado pela máquina de estados. Este capítulo descreve uma linguagem que se assemelha a uma máquina de estados, que permite ao programador definir o fluxo de atividades em um nível mais alto de abstração. A especificação atual gera código para o simulador NS2. No entanto, espera-se que o mesmo programa possa ser usado para gerar código para dispositivos sensores usando uma biblioteca independente de plataforma, como a wiselib (Baumgartner et al., 2010).

A ideia de especificar o fluxo de controle em uma linguagem de nível pode minimizar a complexidade do desenvolvimento de programas baseados em eventos. A programação baseada em eventos é frequentemente usada como um mecanismo de abstração para dispositivos com recursos limitados. Nas RSSFs, este modelo de programação é adotado por sistemas operacionais como o TinyOS e o Contiki, bem como por ambientes de simulação: NS2, NS3 e OMNeT++. No entanto, o fluxo de eventos nesses programas é difícil de entender e manter(Fischer et al., 2007). Examinando programas NS2 codificados por diferentes desenvolvedores, nota-se que a dificuldade de codificar o controle do fluxo de atividades quando eles não são acionados por um evento, mas por uma condição lógica ou por um temporizador. Cada programador usou uma abordagem diferente para lidar com esse tipo de mudança de estado, gerando programas completamente diferentes, que são difíceis de manter.

A linguagem proposta, chamada SLEDS (***S**tate **M**achine-based **L**anguage for **E**vent **D**riven **S**ystems*), trata esse problema, definindo diretamente estados e transições entre eles. As transições podem ser baseadas em eventos e baseadas em lógica. A linguagem SLEDS também suporta primitivas para comunicação ponto-a-ponto e de transmissão entre sensores. A linguagem se concentra no fluxo de coordenação de entidades dos modelos de armazenamento agrupados em *repositórios*, que estão associadas a um conjunto de componentes que implementam funcionalidades comuns. Neste capítulo é apresentado a especificação da linguagem, bem como um esquema de tradução dirigido pela sintaxe (de SLEDS para NS2). Um estudo de caso que implementa modelos de armazenamento agrupado em *repositórios* para RSSFs mostra que a linguagem é geral o suficiente para ser usada em uma ampla variedade de aplicações.

6.2 TRABALHOS RELACIONADOS

O armazenamento agrupado em *repositórios* é uma abordagem descentralizada mais escalonável para RSSFs urbanas de larga escala do que os modelos de armazenamento externo e local. Nesse modelo, alguns sensores da rede são responsáveis por armazenar as leituras de um grupo de sensores. O MKSP (D´ Angelo et al., 2016) segue essa abordagem mapeando dados brutos para nós de armazenamento. A fim de explorar a similaridade de dados espaciais das leituras dos sensores, o AQPM (Carrero et al., 2015a) e o SILENCE (Lee et al., 2015) consideram alguns sensores eleitos para atuarem como representantes do grupo, minimizando a sobrecarga de comunicação. Embora esforços recentes tenham sido feitos para construir sistemas

de armazenamento de dados eficientes, a natureza específica das RSSFs e a falta de uma estrutura comum de desenvolvimento de propósito geral tornam o projeto desses aplicativos uma tarefa difícil.

As linguagens específicas de domínio (DSLs) são linguagens de programação usadas em um contexto bem definido. Ao contrário das linguagens de programação de propósito geral, as DSLs são projetadas para seguir de perto as práticas do seu domínio de aplicação (van Deursen et al., 2000). As DSLs são comumente usadas no contexto das RSSFs (Chandra e Dwivedi, 2015), bem como para a definição de sistemas de transição de estados. No contexto das RSSFs, o Hood (Whitehouse et al., 2004) fornece uma abstração de programação de vizinhança. Os algoritmos são projetados com base em um conjunto de critérios para escolha de vizinhos e definição de variáveis para compartilhar entre eles. O Hood visa simplificar o uso de operações como sincronização e comunicação com sensores vizinhos. O SenNet (Salman e Al-Yasiri, 2016) é uma plataforma que reduz a complexidade da programação, pois abstrai detalhes em diferentes níveis da rede, desde o desenvolvimento aplicativos em nível de nó ou nível de grupo. Embora o propósito de Hood, SenNet e SLEDS seja semelhante, fornecer uma abstração de alto nível para o desenvolvimento de aplicativos para sensores, as abordagens adotadas por cada um deles diferem. SLEDS é baseado em uma máquina de estados, enquanto Hood é baseado no conceito de vizinhança. O SenNet não adota um modelo de fluxo de execução flexível baseado em um coordenador, como o proposto por SLEDS.

No contexto dos modelos de armazenamento agrupados em *repositórios*, o Regiment (Newton et al., 2007) é uma DSL que fornece uma visão geotemporal da RSSF. A linguagem fornece primitivas para manipular conjuntos de fluxos de dados geo-localizados. Essa visão centralizada é traduzida pelo compilador em código específico a ser executado por cada sensor na rede. O uso de DSLs para a definição de máquinas de estado é um tópico bem estudado (Fowler, 2010). Por exemplo, em (Murr e Mauerer, 2017), os autores propõem uma DSL para implementar um tipo específico de máquina de estados para descrever sistemas complexos. No entanto, o foco não está no desenvolvimento de aplicativos para sensores, como SLEDS. Observa-se que o desenvolvimento de programas para controlar a coordenação do armazenamento agrupado em *repositórios* segue padrões que podem ser modelados por máquinas de transição de estado. Além disso, o desenvolvimento de tais aplicações é geralmente considerado complexo por um programador com pouca experiência. SLEDS propõe-se resolver esses problemas, facilitando o desenvolvimento de aplicações de armazenamento em RSSFs urbanas que seguem a classificação agrupada em *repositórios*.

6.3 LINGUAGEM SLEDS

SLEDS - State Machine-based Language for Event Driven Systems é uma linguagem de domínio específico projetada para desenvolver programas de simulação baseados em eventos. A gramática da Figura 6.1 descreve a sintaxe da linguagem SLEDS.

Cada nó sensor executa uma instância da máquina de estados escrita em SLEDS. Uma máquina de estado de um sensor comunica-se com outras máquinas de outros sensores através de trocas de mensagens assíncronas. Um programa SLEDS é composto por (*i*) declaração `Program`, um identificador e uma lista de parâmetros entre parênteses separados por vírgulas, (*ii*) por uma sequência de constantes *const*, (*iii*) por uma sequência de variáveis, e (*iv*) por uma sequência de estados `StateDef`. Uma constante pode ser um número `number` ou uma cadeia de caracteres `string` e uma declaração de variável é composta por um tipo e um identificador. Em um programa SLEDS, todas as constantes e variáveis devem ser declaradas antes do uso.

Program	::=	**Program** *Id*(*Type Id*(**,** *Type Id*)***)** **{**
		(**const** *Id* **=** *(Num-Literal* | *Str-Literal)*;)*
		(*Type VarList*;)*
		StateDef* **}**
VarList	::=	*Var* (**,** *Var*)***;**
Var	::=	*Assignment* | *Id*
StateDef	::=	**State** *Id*(*Type Id*(**,***Type Id*)***)** **{** *ActionList* **}**
State	::=	*Id* **(** *ExpList*? **)** | ***exit***
ActionList	::=	*Action* (*Action*)*
Action	::=	**nextState** *State***;**
	|	**broadcast(***Exp***,** *Exp***,** *ExpList***);**
	|	**send(***Exp***,** *Exp***,** *ExpList***,** *ExpList***);**
	|	**on recvBroadcast(***Id***,** *Id***,** *IdList***)** **{***ActionList***}**
	|	**on recv(***Id***,** *Id***,** *IdList***,** *IdList***)** **{** *ActionList* **}**
	|	**during(***Exp***) on recvBroadcast(***Id***,** *Id***,** *IdList***)**
		{ *ActionList* **}** **nextState** *State***;**
	|	**during (***Exp***) on recv(***Id***,** *Id***,** *IdList***,** *IdList***)**
		{ *ActionList* **}** **nextState** *State***;**
	|	**while (***Exp***)** **{** *ActionList* **}**
	|	**for** *Id* **in** *Exp* **{** *ActionList* **}**
	|	**if (***Exp***)** **{** *ActionList* **}** (**else {** *ActionList* **}**)?
	|	*Assignment***;**
	|	*Method-call***;**
Method-call	::=	*Exp*
Assignment	::=	*Id* **=** *Exp*
Exp	::=	*Exp* **- >** *Exp* | *Exp* **.** *Id* | *Exp***(***Exp*?**)** | *Id*
ExpList	::=	*Exp* (**,** *Exp*)*
IdList	::=	*Id* (**,** *Id*)*
Num_Literal	::=	$[0-9]+$
Str_Literal	::=	$[a-z]+$
Type	::=	*Simple_Type* | *Map_Type* | *List_Type*
Simple_Type	::=	*int* | *double* | *string*
Map_Type	::=	**map <** *Type Id***,** *Type Id* **>**
List_Type	::=	**list <** *Type Id* **>**

Figura 6.1: Sintaxe de SLEDS

Um `StateDef` é composto por um identificador, uma lista de parâmetros entre parênteses separados por vírgulas e por uma sequência de ações `ACTION`, que definem as instruções executadas por um estado. Uma `Action` é definida pela composição de ações de controle de fluxo padrões, tais como composição sequencial, condicional e iterações. As ações são detalhadas a seguir:

- Action ::= **nextState** *State*: corresponde a uma mudança do estado atual para o estado definido por `State`. Cada declaração `State` possui um nome e uma lista de parâmetros passados como entrada para o próximo estado. O estado `exit` finaliza o programa.
- Action ::= **broadcast (***Exp*, *Exp*, *ExpList***)**: define o envio assíncrono de uma mensagem do sensor para todos seus vizinhos N_i, ou seja, que estejam dentro do seu raio

de comunicação. Os argumentos são o tipo da mensagem , o identificador da mensagem e uma lista de parâmetros.

- Action ::= **send (***Exp*, *Exp*, *ExpList*, *ExpList***)**: corresponde ao envio assíncrono de uma mensagem do sensor para um conjunto de sensores. Os argumentos são o tipo da mensagem, o identificador da mensagem, uma lista de destinatários e uma lista de parâmetros.

- Action ::= **on recvBroadcast (***Id*, *Id*, *IdList***){** *ActionList* **}**: corresponde ao recebimento de uma mensagem enviada pela ação **broadcast**.

- Action ::= **on recv (***Id*, *Id*, *IdList*, *IdList***) {** *ActionList* **}**: corresponde ao recebimento de uma mensagem enviada pela ação **send**.

- Action ::= **during (***Exp***) on recvBroadcast (***Id*, *Id*, *IdList***) {** *ActionList* **}** **nextState** *State*: corresponde ao recebimento de uma mensagem enviada pela ação **broadcast** durante um certo intervalo de tempo. Após o tempo expirar, ocorre uma mudança para estado definido por `State`.

- Action ::= **during (***Exp***) on recv (***Id*, *Id*, *IdList*, *IdList***) {** *ActionList* **}** **nextState** *State*: corresponde ao recebimento de uma mensagem enviada pela ação **send** durante um certo intervalo de tempo. Após o tempo expirar, ocorre uma mudança para estado definido por `State`.

Como exemplo, considere o algoritmo de descoberta de vizinhos por inundação na rede, executado por muitos modelos durante a inicialização da rede, representado pela máquina de estados da Figura 6.2. No exemplo, as mudanças de estado estão classificadas em dois tipos:

- Mudança de estado por evento: são aquelas que ocorrem quando um sensor recebe algum tipo de mensagem. Elas estão representadas pela linha de cor azul.

- Mudança de estado lógica: é executada com base no resultado de uma computação ou então após o tempo de expiração de um *timer*. Elas estão representadas pela linha de cor vermelha.

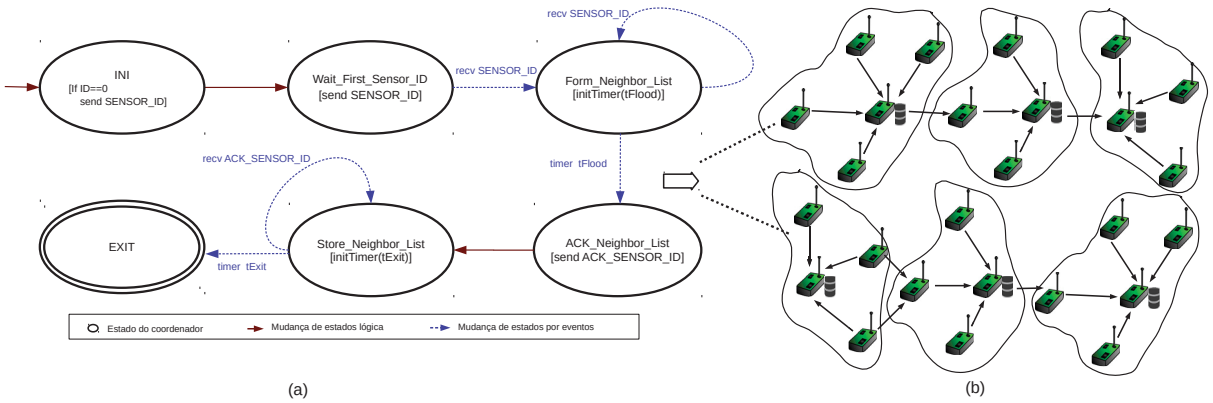


Figura 6.2: Máquina de estados para descoberta de vizinhos

A representação desta máquina no simulador NS2 não é trivial pois o desenvolvedor precisa criá-la *on-the-fly* durante a implementação do sistema. De fato, o NS2 fornece uma

abstração limitada que dificulta a implementação de modelos de máquinas de estado. Consequentemente, é provável que dois desenvolvedores produzam códigos de simulação distintos para a máquina de estados ilustrada pela Figura 6.2. A linguagem SLEDS facilita esta tarefa, pois possibilita descrever a máquina de estados em um formato de alto nível e, posteriormente, traduzir esta representação para código de simulação do NS2. O Código 6.1 ilustra o programa SLEDS correspondente à máquina de estados para descoberta de vizinhos.

```
1  use compSensor as ComponentsSensor;
2  use compLibMSG as ComponentsLibMessage;
3
4  Program Coordinator() {
5    const tFlood=25;
6    const tExit =0.1;
7    int myID = compSensor->getSensorId();
8    list <int> listSensorAnnouncements;
9    int msgID;
10
11   STATE INI() {
12     if (myID == 0) {
13       msgID = compLibMSG->GetNextMsgId();
14       broadcast(SENSOR_ID, msgID, myID);
15       compLibMSG->addSeenMsg(SENSOR_ID, msgID); }
16     nextState Wait_First_Sensor_ID (); }
17
18   STATE Wait_First_Sensor_ID() {
19     on recvBroadcast(SENSOR_ID, msgID, ID) {
20         listSensorAnnouncements. insert (ID);
21         if (!compLibMSG->seenMsg(SENSOR_ID, msgID)) {
22           compLibMSG->addSeenMsg(SENSOR_ID, msgID);
23           broadcast(SENSOR_ID, msgID, myID); }
24         nextState Form_Neighbor_List(); } }
25
26   STATE Form_Neighbor_List() {
27     during (tFlood) on recvBroadcast(SENSOR_ID, msgID, ID) {
28       listSensorAnnouncements. insert (ID); }
29     nextState ACK_Neighbor_List(); }
30
31   STATE ACK_Neighbor_List() {
32     send(ACK_SENSOR_ID, compLibMSG->GetNextMsgId(),
33         listSensorAnnouncements, myID);
34     nextState Store_Neighbor_List (); }
35
36   STATE Store_Neighbor_List() {
37     during (tExit) on recv(ACK_SENSOR_ID, msgID,
38                            listSensorAnnouncements, fromID) {
39       for v in listSensorAnnouncements
40         if (v == myID)
41           compSensor->listKnownNeighbors.insert(fromID); }
42     nextState exit ;}
```

Código 6.1: Programa SLEDS para descoberta de vizinhos

O programa assume a existência de componentes: `compSensor` e `compLibMSG`. O primeiro oferece funcionalidades do sensor, como consultar seu identificador (função *getSensorID*) e armazenar a lista de identificadores de seus vizinhos (função *listKnownNeighbors*). O componente `compLibMSG` possui funcionalidades para troca de mensagens entre os sensores. Por exemplo, há uma função para criar um novo identificador da mensagem (*GetNextMsgId*) e para armazenar, para cada sensor, a identificação do tipo para a mensagem recebida (*addSeenMsg*). De fato, em um ambiente de programação baseado em componentes, um programa SLEDS desempenha o papel de um coordenador, que é o responsável pelo fluxo de atividades que conectam os componentes de software, especificando as interações entre eles. Após a inclusão das referências *(l.1-2)*, o programa declara um conjunto de constantes e variáveis. A constante *tFlood (l.5)* define o atraso das transmissões de mensagens na rede e a constante *tExit (l.6)* determina o atraso necessário entre enviar uma mensagem e receber uma confirmação para evitar colisões. A variável *myID* armazena o identificador único do sensor, que é obtido executando a função *getSensorId()* do componente *compSensor (l.7)*.

O Código descreve o fluxo de execução do coordenador e cada nó sensor executa uma instância da máquina de estados escrita em SLEDS, iniciando pelo estado `STATE INI` *(l.11)*. No estado inicial `STATE INI`, o sensor com identificador (`myID`) zero envia a mensagem do tipo `SENSOR_ID` para todos seus vizinhos *(l.12-14)* e armazena o identificador da mensagem e o identificador do tipo da mensagem, para evitar o envio da mensagens duplicadas *(l.15)*. O sensor com identificador `myID` zero e os demais sensores realizam uma mudança lógica do estado `INI` para o estado `Wait_First_Sensor_ID` *(l.16)*. Os sensores que receberem a primeira mensagem `SENSOR_ID` armazenam o `ID` contido na mensagem e enviam seu identificador para seus vizinhos *(l.19-23)* e realizam uma mudança por evento do estado `Wait_First_Sensor_ID` para o estado `Form_Neighbor_List` *(l.24)*. No estado `Form_Neighbor_List`, os sensores continuam armazenando o `ID` dos demais vizinhos durante um certo tempo *tFlood* (até o fim da inundação) *(l.26-28)*. Quando o tempo expirar, os sensores realizam uma mudança lógica para o estado `ACK_Neighbor_List` *(l.29)* e enviam uma mensagem de `ACK` para os sensores conhecidos, que estão armazenados na variável *listSensorAnnouncements (l.32)*. Após o envio da mensagem de `ACK`, os sensores realizam mudança lógica para o estado `Store_Neighbor_List` *(l.34)*. No estado `Store_Neighbor_List` , durante um certo tempo *tExit (l.36-38)*, os sensores que receberam a mensagem verificam se existe uma mensagem de `ACK` destinada à ele para então atualizar a lista de vizinhos *(l.40-41)*. Ao final da inundação, cada sensor possui armazenado em sua variável local *listKnowNeighbors*, a lista de seus vizinhos. A próxima seção apresenta uma proposta para traduzir programas SLEDS em códigos de simulação do ambiente NS2.

6.4 TRADUÇÃO DO CÓDIGO SLEDS PARA NS2

No NS2, o coordenador tem duas funções principais a serem implementadas pelo desenvolvedor: `recv` e `TimerHandle`. A função `recv` é responsável pelo recebimento de pacotes pelos sensores e trata essas mensagens. É a parte do código que faz as transições do estado do tipo “mudança por evento". A função `TimerHandle` é executada quando existe a transição do tipo "mudança de estado lógica" e é ativada quando um *timer* expira. Como exemplo, considere os trechos de código SLEDS e NS2 ilustrados pela Figura 6.3, executados pelo estado `INI`.

```
1 STATE INI() {
2   if (myID == 0) {
3       msgID = compLibMSG→GetNextMsgId();
4       broadcast(SENSOR_ID, msgID, myID);
5       compLibMSG→addSeenMsg(SENSOR_ID, msgID);
6   }
7   nextState Wait_First_Sensor_ID();
8 }
```

```
1  void WSN_ComponentsAgent::TimerHandle(State st) {
2    switch (st) {
3        case INI: {
4          if (myID == 0) {
5            msgID = compLibMSG→GetNextMsgId();
6            broadcast(SENSOR_ID, msgID, myID);
7            compLibMSG→addSeenMsg(SENSOR_ID, msgID);
8          }
9          nextState=Wait_First_Sensor_ID;
10       }
11   }
12 }
```

Figura 6.3: Código SLEDS e NS2 para o estado INI

```
1  STATE Wait_First_Sensor_ID() {
2    on recvBroadcast(SENSOR_ID, msgID, ID) {
3      listSensorAnnouncements.insert(ID);
4      if  (!compLibMSG→seenMsg(SENSOR_ID, msgID)) {
5        compLibMSG→addSeenMsg(SENSOR_ID, msgID);
6        broadcast(SENSOR_ID, msgID, myID);
7      }
8      nextState Form_Neighbor_List();
9    }
10 }
```

```
1  void WSN_ComponentsAgent::recv(Packet* pkt, Handler *) {
2    WSN_Components_Message p = pkt;
3    switch(nextState) {
4      case (Wait_First_Sensor_ID): {
5        if (param.getMsgType() == SENSOR_ID) {
6          listSensorAnnouncements.insert(param.getSensorID());
7          if  (!compLibMSG→seenMsg(p.getSensorID(), p.getMsgID())) {
8            compLibMSG→addSeenMsg(p.getSensorID(), p.getMsgID());
9            broadcast(SENSOR_ID, msgID, myID);
10         }
11         nextState=Form_Neighbor_List;
12       }
13     }
14   }
15 }
```

Figura 6.4: Código SLEDS e NS2 para o estado Wait_First_Sensor_ID

```
1 STATE Form_Neighbor_List() {
2   During (tFlood) on recvBroadcast(SENSOR_ID, msgID, ID) {
3       listSensorAnnouncements.insert(ID);
4   }
5   nextState ACK_Neighbor_List();
6 }
```

```
1  void WSN_ComponentsAgent::recv(Packet* pkt, Handler *) {
2    WSN_Components_Message p = pkt;
3    switch(nextState) {
4      case (Form_Neighbor_List): {
5        if (param.getMsgType() == SENSOR_ID) {
6          if (nextState != previousState) {
7            libTimer.resetTimer(tFlood);
8            previousState=Form_Neighbor_List;
9          }
10         listSensorAnnouncements.insert(p.getSensorID());
11       }
12 }
```

```
1  void WSN_ComponentsAgent::TimerHandle(State st) {
2    switch (st) {
3        case Form_Neighbor_List: {
4          nextState=ACK_Neighbor_List;
5          libTimer.resetTimer(0);
6        }
7    }
8  }
```

Figura 6.5: Código SLEDS e NS2 para o estado Form_Neighbor_List

O código do estado `INI` de SLEDS é traduzido para uma cláusula *case* da função `TimerHandle` do NS2, que é executada no início da simulação. O `TimerHandle` é composto por um comando *switch*, com vários *cases*, cada um equivalente a um estado que pode ser atingindo pelas "mudanças de estado lógica". A cláusula *case* do estado estado `INI` contém o mesmo código fornecido pelo programa SLEDS. Observe na *linha 9* do código NS2 que o fluxo de ações do estado `INI` termina com uma transição para o estado `Wait_First_Sensor_ID`, ilustrado pela Figura 6.4.

O código SLEDS do estado `Wait_First_Sensor_ID`, executada pelos sensores que receberem a primeira mensagem identificada por `SENSOR_ID`, é traduzido para a função `recv` do NS2. Assim como a função `TimerHandle`, a função `recv` é composta por um comando *switch*, com vários *cases*, cada um equivalente a um estado que pode ser atingindo pelas mudanças de "estado por evento" *(l. 3-4)*. Parte do código gerado para o NS2 é para se obter os parâmetros do pacote recebido, mas a maior parte do código dentro de cada cláusula *case* é

```
1 STATE ACK_Neighbor_List() {
2   send(ACK_SENSOR_ID, compLibMSG→GetNextMsgId(),
3         listSensorAnnouncements, myID);
4   nextState Store_Neighbor_List();
5 }
```

```
1 void WSN_ComponentsAgent::TimerHandle(State st) {
2   switch (st) {
3     case ACK_Neighbor_List: {
4       send(ACK_SENSOR_ID, compLibMSG→GetNextMsgId(),
5             listSensorAnnouncements, myID);
6       nextState=Store_Neighbor_List;
7     }
8   }
9 }
```

Figura 6.6: Código SLEDS e NS2 para o estado ACK_Neighbor_List

```
1 STATE Store_Neighbors_List() {
2   During (tExit) on recv(ACK_SENSOR_ID, msgID,
3                 listSensorAnnouncements, fromID) {
4     for v in listSensorAnnouncements
5       if ( v == myID )
6         compSensor→listKnownNeighbors.insert(fromID);
7   }
8   nextState exit;
9 }
```

```
1  void WSN_ComponentsAgent::recv(Packet* pkt, Handler *) {
2    WSN_Components_Message p = pkt;
3    switch(nextState) {
4      case (Store_Neighbors_List): {
5        if (param.getMsgType() == ACK_SENSOR_ID) {
6          if (nextState != previousState) {
7            libTimer.resetTimer(tExit);
8            previousState=Store_Neighbors_List;
9          }
10         for (int<int>::iterator v=listSensorAnnouncements.begin();
11                   v!=listSensorAnnouncements.end(); v++) {
12           if (*v == myID)
13             compSensor→listKnownNeighbors.insert(p.fromID);
14         }
15       }
16   }
```

```
1 void WSN_ComponentsAgent::TimerHandle(State st) {
2   switch (st) {
3     case Store_Neighbors_List: {
4       nextState=exit;
5       libTimer.resetTimer(0);
6     }
7   }
8 }
```

Figura 6.7: Código SLEDS e NS2 para o estado Store_Neighbor_List

idêntica ao programa SLEDS. Portanto, cada mensagem possui um `ID`, e dentro dela existem alguns parâmetros como o `ID` do emissor e o `ID` do tipo da mensagem. No entanto, a tradução não é tão direta quando a transição envolve a geração de código tanto para a função`TimerHandle` quanto para a função `recv`, como ilustra a Figura 6.5, referente à transição para o estado `Form_Neighbor_List`.

Note que o código SLEDS do estado `Form_Neighbor_List` gera trechos de código tanto na função `recv` quanto na função `TimerHandle` do NS2. O trecho de código da função `recv` trata as mudanças de "estado por evento", armazenando as mensagens recebidas (*l.10*) durante um certo intervalo de tempo $tFlood$ (*l.7*). Durante este período, o sensor armazena os anúncios dos vizinhos (*l.10*) para cada mensagem `SENSOR_ID` recebida. No entanto, a transição para o próximo estado `ACK_Neighbor_List` não pode ser feita nesta função, uma vez que o sensor pode receber várias mensagens deste tipo. Assim, a transição é codificada na função `TimerHandle`, que é disparada quando o temporizador $tFlood$ expirar. O trecho de código ilustrado pela Figura 6.6 é executado após esta transição.

O código SLEDS do estado `ACK_Neighbor_List` gera trechos de código na função `TimerHandle`. A ação associada ao estado envia mensagens de `ACK` para um conjunto de sensores armazenados em uma lista (*l.4-5*). Os sensores que receberem as mensagens de `ACK` executam o código do estado `Store_Neighbor_List`, ilustrado pela Figura 6.7.

O código SLEDS do estado `Store_Neighbor_List` gera trechos de código tanto na função `recv` quanto na função `TimerHandle`. Durante um certo tempo (*l.7*), o sensor armazena em uma lista os identificadores `myID` dos sensores conhecidos (*10-13*). Quando o tempo expirar, ocorre uma "mudança de estado por evento" para o estado `EXIT`, finalizando a simulação.

(r1) $StateDef$ ::= **State** Id_1 "(" $Type\ Id_2$ (, $Type\ Id_3$)*) "{" $ActionList$ "}"

```
if (ActionList.tc != null)
    State.tc = "switch(nextState):{ case" + Id1.txt + ":"
                    + "{" ActionList.tc + "}}"
```

(r2) $ActionList$ ::= $Action_1$ $(Action)^*$

```
if (ActionList.dest == rc) {
    Action1.dest = ActionList.dest
    Action2.dest = ActionList.dest
    ActionList.rc+ = Action1.rc + ";" + Action2.rc
 }
else { ActionList.tc+ = Action1.tc + ";" + Action2.tc }
```

(r3) $Action$ ::= **if** (Exp) "{" $ActionList$ "}"

```
if (Action.dest == rc) {
    ActionList.dest = Action.dest
    Action.rc = "if (" + Exp.txt + ") {" + ActionList.rc + "}"
 }
else {
    Action.tc = "if (" + Exp.txt + ") {" + ActionList.tc + "}"
 }
```

(r4) $Action$ ::= **nextState** $State$

```
if (Action.dest == rc) { Action.rc = "nextState" + State }
```

(r5) $Action$ ::= **broadcast** "(" + $Exp_1.txt$ + $Exp_2.txt$ + $ExpList$) ")"

```
if(Action.dest == rc)
    Action.rc = "broadcast (" + Exp1.txt + Exp2.txt + ExpList + ")"
else
    Action.tc = "broadcast (" + Exp1.txt + Exp2.txt + ExpList + ")"
```

Figura 6.8: Gramática de atributos para avaliação do estado INI

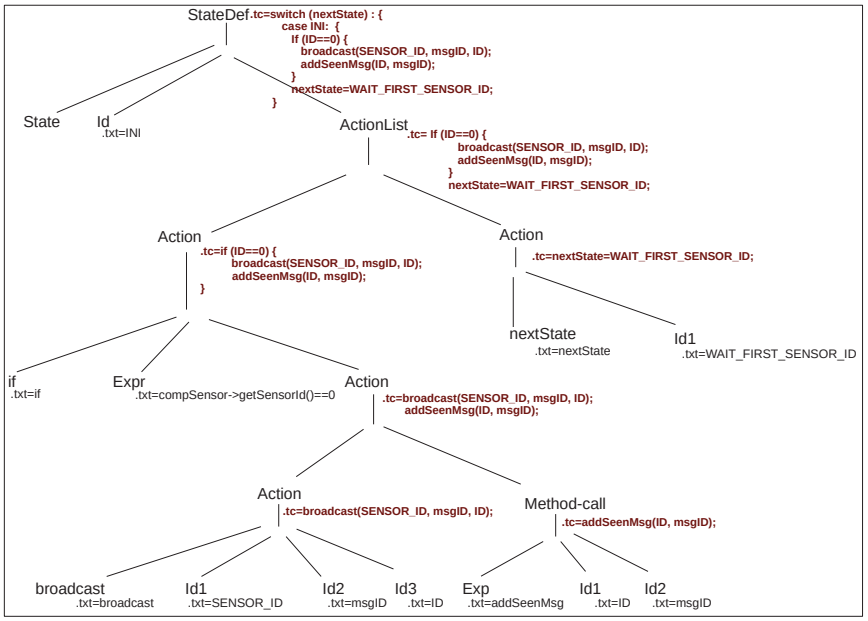


Figura 6.9: Árvore de análise sintática para o estado INI

Uma técnica utilizada para gerar código é a tradução dirigida por sintaxe, na qual *atributos* e *regras semânticas* são associadas às regras de produção de uma gramática (Aho

et al., 2006). As regras semânticas expressam como a computação dos atributos se relaciona com as regras gramaticais da linguagem, conectando fragmentos de código às produções de uma gramática. O conjunto de atributos juntamente com as regras gramaticais é denominado *gramática de atributos* (Louden, 2004). Existem dois tipos de atributos: sintetizados e herdados. Em nossa abordagem, os atributos sintetizados são usados para passar informações semânticas para cima da árvore de análise, enquanto os atributos herdados ajudam a passar as informações semânticas para baixo.

Um atributo é qualquer informação associada a terminais e não-terminais da gramática, como tipos de dados, números, referências para tabelas ou fragmentos de código. Em nossa gramática, existem três atributos: `rc`, `tc` e `dest`. Os dois primeiros são atributos sintetizados e determinam se o fragmento de código será incluído na função `recv` ou `TimerHandle`, respectivamente. O atributo `dest` é herdado e pode conter o valor `rc` ou `tc` e é usado para transmitir para baixo da árvore a função na qual o código será gerado, com base no contexto do nó. No esquema usado, a cada produção um conjunto de regras semânticas conectam fragmentos de código que produz a tradução do programa SLEDS para código de simulação, ou seja, a tradução de um programa SLEDS para um programa C++ do simulador de redes NS2. A gramática de atributos ilustrada pela Figura 6.8 mostra a geração de código para o estado `INI`, e a Figura 6.9 o resultado da análise da árvore sintática.

Observe na parte inferior da árvore que o código é gerado para o atributo sintetizado `tc`. Um atributo é considerado *s*intetizado se o seu valor no nó da árvore de derivação N for determinado a partir dos valores dos atributos dos filhos de N e no próprio N. Os atributos *s*intetizados podem ser computados por um único percurso ascendente, ou em pó-ordem, da árvore. Portanto, o valor de `tc` é usado para passar passar informação para cima da árvore, agregando código no atributo `tc` do nó $StateDef$. Este código corresponde aos fragmentos de código gerados para a função `TimerHandle` do NS2. O símbolo +, que aparece no lado direito das regras, é usado para concatenar texto (código). Os demais não-terminais possuem o atributo `txt`, cujo valor é fornecido pelo analisador léxico.

Considere agora a gramática de atributos da Figura 6.10 e a árvore de análise sintática do estado `Wait_First_Sensor_ID`, ilustrada pela Figura 6.11. Observe na parte inferior da árvore que o atributo herdado `dest` do nó *Action* contém o valor `rc` para passar a informação para baixo da árvore, pois o nó condicional if deve gerar código para o atributo `rc`. De fato, o código para o atributo `rc` é gerado para todo não-terminal `Action` que estiver no contexto de **on recv** ou **on recvBroadcast**. Este atributo então receberá o fragmento de código, que é passado pela árvore para compor o valor `rc` final na raiz da árvore ($StateDef$). Este é o código que será incluído na função `recv` do NS2.

Ao contrário do código gerado para os exemplos anteriores, a avaliação do estado `Form_Neighbor_List` gera código tanto para o atributo `tc` quanto para o atributo `rc`. Considere a gramática de atributos ilustrada pela Figura 6.12 e a árvore de análise sintática da Figura 6.13 que serão usadas para gerar código referente ao estado `Form_Neighbor_List`. Note na parte inferior da árvore que o atributo herdado `dest` do nó *Action* contém o valor `rc` para passar a informação para baixo da árvore, pois o nó $Method-call$ deve gerar código para o atributo `rc`. Este atributo então receberá o fragmento de código, que é passado pela árvore para compor o valor `rc` final na raiz da árvore ($StateDef$). O mesmo processo é usado para compor o valor final do atributo `tc`, com o código a ser incluído na função `TimerHandle` do NS2.

(r1) $StateDef$ ::= **State** Id_1"("$Type\ Id_2(,Type\ Id_3)^*$) "{" $ActionList$ "}"

```
if (ActionList.tc != null)
    State.tc = "switch(nextState):{ case" + Id1.txt + ":"
                    + "{" ActionList.tc + "}}"
```

(r2) $ActionList$ ::= $Action_1$ $(Action)^*$

```
if (ActionList.dest == rc) {
    Action1.dest = ActionList.dest
    Action2.dest = ActionList.dest
    ActionList.rc+ = Action1.rc + ";" + Action.rc
 }
else {
    ActionList.tc+ = Action1.tc + ";" + Action.tc
 }
```

(r3) $Action$ ::= **on recvBroadcast** "(" + $Id_1.txt$ + $Id_2.txt$ + $IdList$ ")"

```
ActionList.dest = rc
Action.rc = "if (param.getMsgType() == " + Id1.txt + "{" + ActionList.rc + "}"
```

(r4) $Action$ ::= **if** (Exp) "{" $ActionList$ "}"

```
if (Action.dest == rc) {
    ActionList.dest = Action.dest
    Action.rc = "if (" + Exp.txt + ") {" + ActionList.rc + "}"
 }
else {
    Action.tc = "if (" + Exp.txt + ") {" + ActionList.tc + "}"
 }
```

(r5) $Action$ ::= **nextState** $State$

```
if (Action.dest == rc) { Action.rc = "nextState" + State }
else { Action.tc = "nextState" + State }
```

(r6) $Action$ ::= **broadcast** "(" + $Exp_1.txt$ + $Exp_2.txt$ + $ExpList$ ")"

```
if(Action.dest == rc)
    Action.rc = "broadcast (" + Exp1.txt + Exp2.txt + ExpList + ")"
else
    Action.tc = "broadcast (" + Exp1.txt + Exp2.txt + ExpList + ")"
```

Figura 6.10: Gramática de atributos para avaliação do estado Wait_First_Sensor_ID

6.5 VALIDAÇÃO

O RCBM, como visto no Capítulo 4, é uma plataforma baseada em componentes de software para desenvolver modelos de armazenamento em RSSFs que promovam a reutilização de código. Ele é representado na caixa central da Figura 6.14. O RCBM trata de entidades do modelo agrupado em *repositórios* que compartilham conceitos e funcionalidades, que representam várias instâncias de sistemas de armazenamento em RSSFs. Essas funcionalidades comuns são os componentes do sistema. Embora tenha sido demonstrado que a estrutura é eficiente para promover a reutilização do código, o desenvolvedor da aplicação ainda é responsável por codificar a coordenação entre os componentes. SLEDS, com suas primitivas de composição de alto nível, pode ser usado para gerar código para o coordenador do RCBM, como ilustra esta seção.

O RCBM foi implementado no simulador NS2 e considera três tipos de componentes: componentes da biblioteca, componentes da aplicação e o coordenador. Os componentes da biblioteca fornecem um conjunto de ferramentas, que pode ser usada para implementar componentes do aplicativo, associados a entidades das RSSFs. Para modelos agrupado em

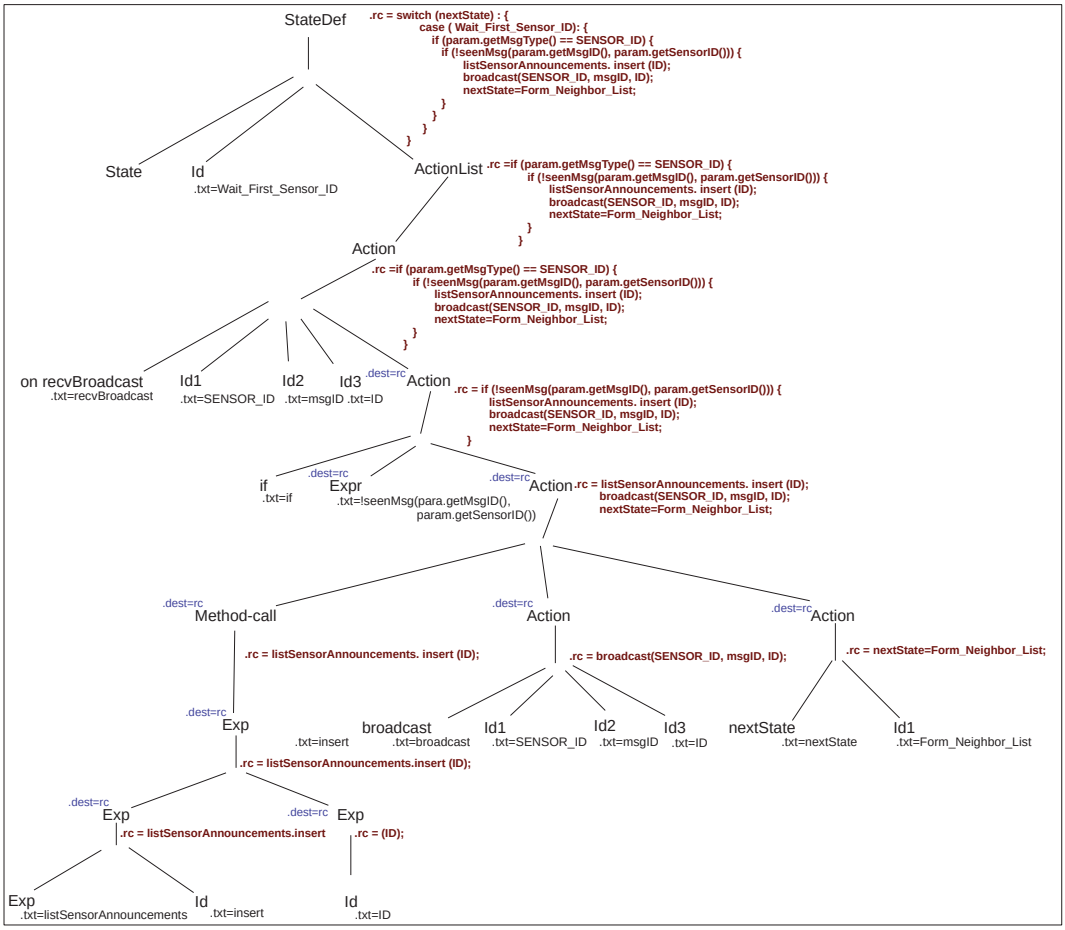


Figura 6.11: Árvore de análise sintática para o estado Wait_First_Sensor_ID

(r1)	$StateDef$ ::= **State** Id_1 "(" $Type\ Id_2$(, $Type\ Id_3$)*) "{" $ActionList$ "}"

	`if (`$ActionList.tc$ `!= null)`
	$State.tc$ = "`switch (nextState):{ case`" + $Id_1.txt$ + ":" + "{" $ActionList.tc$ + "}}"
	`if (`$ActionList.rc$ `!= null)`
	$State.rc$ = "`switch (nextState):{ case`" + $Id_1.txt$ + ":" + "{"+ $ActionList.rc$ + "}}"

(r2)	$Action$ ::= **during** "(" Exp ")" **on recvBroadcast** "(" Id_1, Id_2, $IdList$ ")" "{" $ActionList$ "}" $nextState\ State$ ")"
	$Action.rc$ = "`if (nextState != previousState) {`" $Action.rc$ += "`libTimer.resetTimer(`"+$Exp.txt$+")" $Action.rc$ += "`previousState=`"+$Action.pst$ $Action.rc$ += $ActionList.rc$ + "}"
(r3)	$Method-call$ ::= Exp
	`if(`$Method-call.dest$`==rc)` {$Method-call.rc$ = $Exp.rc$} `else` { $Method-call.tc$ = $Exp.tc$ }

Figura 6.12: Gramática de atributos para avaliação do estado Form_Neighbor_List

repositórios, essas entidades incluem: *(i)* dispositivos sensores; *(ii)* membros do *cluster* (CM), que consistem em um conjunto de sensores; e *(iii)* líderes de agrupamento (CHs) que são sensores responsáveis por armazenar as informações de todos os membros do agrupamento. Essas

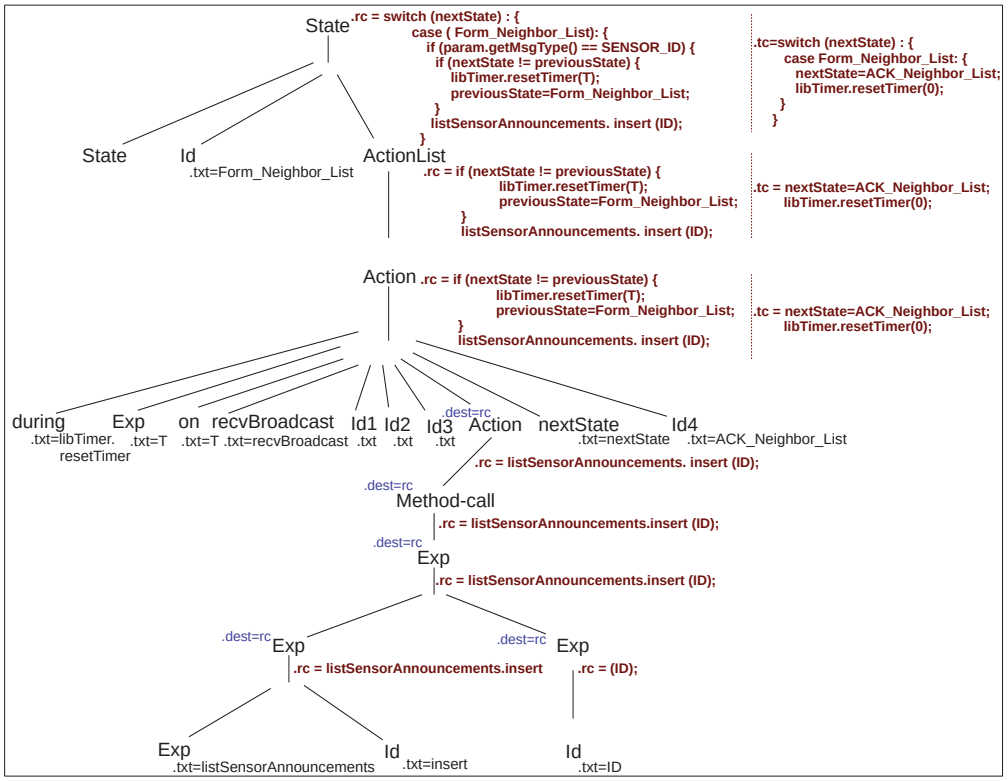


Figura 6.13: Árvore de análise sintática para o estado Form_Neighbor_List

entidades definem um modelo de armazenamento hierárquico, onde cada agrupamento designa um sensor como o líder do grupo para armazenar as leituras de seus membros. O coordenador é o responsável pelo fluxo de execução e troca de mensagens. A próxima seção mostra um estudo de caso que implementa modelos de armazenamento agrupados em *repositórios* para RSSFs.

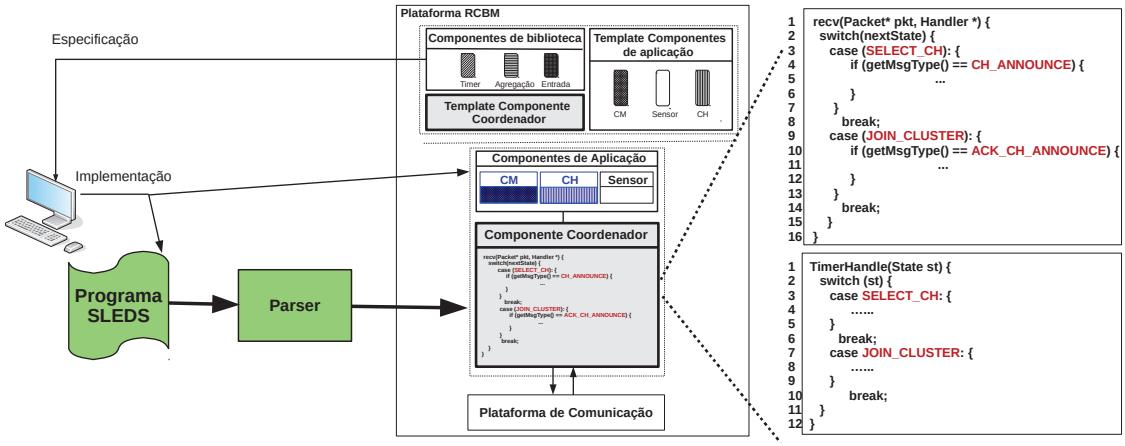


Figura 6.14: Arquitetura do *back-end* da linguagem SLEDS

6.5.1 Implementação da coordenação do LEACH

O LEACH (***L**ow-**E**nergy **A**daptive **C**lustering **H**ierarchy*) (Heinzelman et al., 2000) é um modelo probabilístico que forma *clusters* cuja distância entre os sensores é de um salto. O LEACH assume que todos os nós estão dentro do alcance de comunicação um do outro. Os

sensores se elegem como líderes dada uma probabilidade p. No RCBM, o componente `compCH` define a função `selectCH(map <K, V>)` que o desenvolvedor deve implementar de acordo com o critério para seleção de líder do modelo em questão. Portanto, cada nó sensor s_i executa `selectCH (`s_i`, p)`, onde $K = s_i$ e $V = p$. A Listagem 6.2 mostra o código de coordenação SLEDS da fase de eleição do líder.

```
1  // Programa executado por cada nó sensor
2  use compSensor as WSN_ComponentsSensor;
3  use compCH as WSN_ComponentsCH;
4  use compCM as WSN_ComponentsCM;
5  use compLibMSG as ComponentsLibMessage;
6
7  Program Coordinator() {
8    const p=0.2;
9    const tCluster =25;
10   const tExit =0.1;
11
12   double RSS;
13   int myCH;
14   int myID=compSensor->getSensorId();
15   list <int, double> knownCHs;
16   list <int> sensorList ;
17
18   STATE Select_CH() {
19   if (compCH->selectCH(myID, p)) {
20     broadcast(CH_ANNOUNCE, compLibMSG->GetNextMsgId(), myID);
21     compSensor->role = CH; }
22   else {
23     compSensor->role = CM; }
24   nextState Join_Cluster (); }
```

Código 6.2: Coordenação do estado Select_CH

O estado `Select_CH` (*l.18*) descreve as ações que devem ser executadas durante a fase de eleição do líder. Inicialmente, cada nó sensor executa `selectCH()` (*l.19*). Se o resultado da computação da função retornar um valor verdadeiro, o sensor transmitirá seu papel como CH para a rede (*l.20-21*) e realizará uma transição de estado para `Join_Cluster` (*l.24*). Caso contrário, o papel do sensor é definido como um membro de *cluster* (CM) (*l.22-23*). A Listagem 6.3 ilustra o código de estado do `Join_Cluster`.

```
25   STATE Join_Cluster() {
26   During ( tCluster ) on recvBroadcast(CH_ANNOUNCE, msgID, ID) {
27     RSS = compSensor->getRSS(ID);
28     knownCHs.insert(ID, RSS); }
29   nextState Cluster_Formation (); }
```

Código 6.3: Coordenação do estado Join_Cluster

No próximo estado (*l.25*), os nós sensores esperam por `tCluster` unidades de tempo por anúncios dos CHs e atualizam o valor de `knownCHs` com base na intensidade do sinal recebido (RSS) (*l.26-28*). Quando o temporizador expirar, os demais sensores ingressarão no *cluster*, conforme ilustrado em Listagem 6.4.

```
30 STATE Cluster_Formation() {
31   if (compSensor->role = CM) {
32       myCH = compCM->joinCluster(knownCHs);
33       sensorList . insert ( myCH );
34       send(ACK_CH_ANNOUNCE, compLibMSG->GetNextMsgId(), sensorList, myID);
35       nextState EXIT;
36   } else {
37   nextState Store_Members(); }
```

Código 6.4: Coordenação do estado Cluster_Formation

No LEACH, um membro do agrupamento (*l.31*) decide se juntar ao líder que requer o menor consumo de energia para se comunicar. Assim, define-se como CH o sensor com o máximo valor de RSS registrado pelos CHs conhecidos (*l.32*). Em seguida, envia uma mensagem ACK_CH_ANNOUNCE para o líder escolhido e passa para o estado EXIT (*l.33-35*). Caso contrário, os CHs realizam uma mudança de estado lógica para Store_Members (*l.36-37*). A Listagem 6.5 descreve a implementação do código SLEDS.

```
38  STATE Store_Members() {
39   During ( tExit ) on recv(ACK_CH_ANNOUNCE, msgID, destListID, fromID) {
40     for v in destListID {
41       if (v = myID) {
42        compCH->members.insert( fromID ); } } }
43     nextState EXIT ;}
```

Código 6.5: Coordenação do estado Store_Members

Os CHs executam as ações correspondentes ao estado `Store_Members` (*l.38*). Primeiro, os CHs aguardam `tExit` unidades de tempo por mensagens `ACK_CH_ANNOUNCE` dos membros do grupo (*l.39*). Se receber uma mensagem, o sensor atualiza sua estrutura de lista de membros (*l.40-42*). Quando o temporizador `tExit` expirar, o programa termina sua execução (*l.43*). A Figura 6.15 ilustra uma máquina de estado do fluxo de coordenação do LEACH, uma instância de um sistema de armazenamento agrupado em *repositórios*.

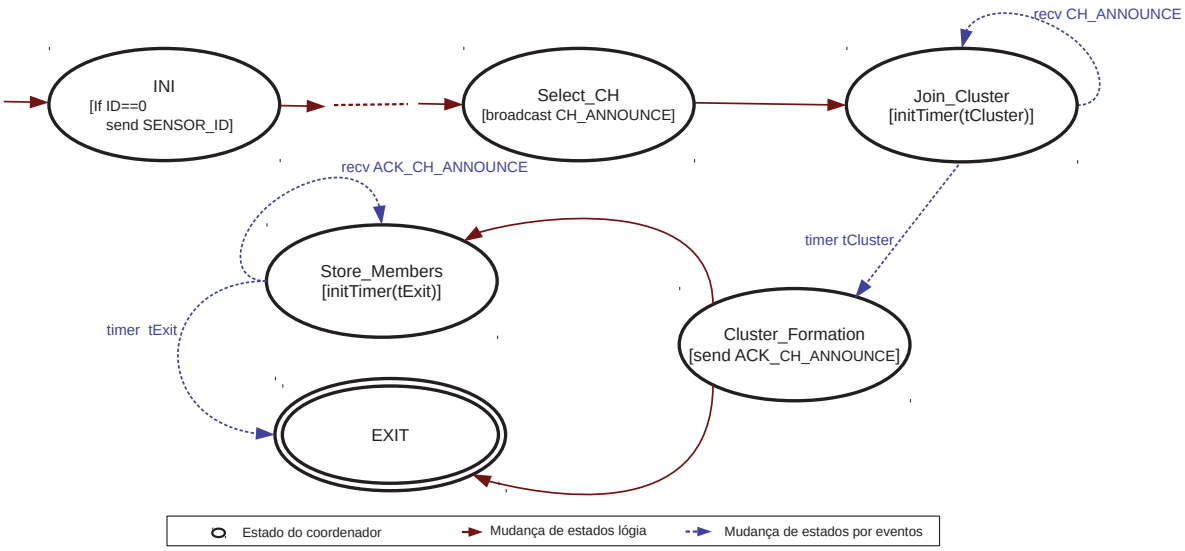


Figura 6.15: Máquina de estados para o modelo de armazenamento agrupado em repositórios.

O fluxo de execução ilustrado na Figura 6.15 é semelhante ao adotado pela estrutura baseada em componente CBCWSN (Amaxilatis et al., 2011b), que demonstrou expressar várias instâncias de armazenamento agrupado em *repositórios*. Como descrito a seguir, o mesmo pode ser dito do programa SLEDS. Para implementar o LCA (Baker e Ephremides, 1981), apenas três linhas de código devem ser modificadas, principalmente para levar em consideração o critério distinto para a eleição do líder. O LCA elege como CH o sensor com o menor ID entre seus vizinhos que não receberam um anúncio de eleição. A Listagem 6.6 mostra os dois estados em que existem linhas no programa SLEDS que diferem do código LEACH. A linha 27 da Listagem 6.3 foi removida e as Linhas 5 e 14 diferem nos argumentos das funções `selectCH` e `knownCHs.insert`.

```
1  Program Coordinator() {
2     list <int> knownNeighbors;
3
4    STATE Select_CH() {
5    if (compCH->selectCH(myID, knownNeighbors)) {
6       broadcast(CH_ANNOUNCE, GetNextMsgId(), myID);
7       compSensor->role = CH; }
8    else {
9       compSensor->role = CM; }
10   nextState Join_Cluster (); }
11
12   STATE Join_Cluster() {
13   During ( tCluster ) on recvBroadcast(CH_ANNOUNCE, msgID, ID) {
14      knownCHs.insert(ID); }
15   nextState Cluster_Formation (); }
```

Código 6.6: Coordenação do LCA

Os dois estudos de caso apresentados neste capítulo mostram que as instâncias do modelo compartilham a mesma especificação da máquina de estados, promovendo a capacidade de reutilização de código. O programador desenvolve algumas linhas de código levando em consideração as especificidades de cada modelo. Além disso, as primitivas da máquina de estados adotadas pela linguagem SLEDS não impõem nenhum fluxo fixo de atividades (como o CBCWSN), mas permitem que o desenvolvedor defina a coordenação de qualquer modelo agrupado em *repositórios*.

6.6 RESUMO

Este capítulo apresentou uma linguagem específica de domínio, denominada SLEDS, para prototipagem de aplicativos para RSSFs que adotam uma abordagem de armazenamento agrupado em *repositórios*. A especificação atual da linguagem gera código para o ambiente de simulação NS2, usando uma biblioteca de componentes fornecida pelo RCBM. A abordagem foi validada definindo-se um esquema de tradução dirigida pela sintaxe para geração de código no simulador NS2. Como estudos de caso, foram desenvolvidos programas SLEDS para modelos agrupados em *repositórios* de dados (LEACH e LCA). Os experimentos mostraram que ambos os modelos compartilham muitas semelhanças no fluxo de execução. Os resultados obtidos mostram que os SLEDS promoveu reutilização de código e desenvolvimento ágil para a especificação do LCA. A proposta responde a alguns dos desafios identificados em Estrin et al. (1999).

7 CONCLUSÃO

Esta tese investigou princípios e técnicas para facilitar a geração de código de sistemas de armazenamento em ambientes de simulação de RSFFs urbanas. As dificuldades encontradas para gerar código de simulação são várias e foram apontadas soluções para auxiliar o desenvolvimento de código de maneira produtiva e ágil. Foi apresentada uma classificação para os modelos de armazenamento e o desenvolvimento de um novo modelo autônoma e escalável de sensoriamento urbano. Em seguida, com base na taxonomia, foi proposto um modelo de componentes e uma máquina de estados para especificar o controle de fluxo e interação entre os componentes. Em direção à um *framework* de desenvolvimento, foi proposto SLEDS, uma linguagem de programação inspirada no modelo da máquina de estados para programadores implementarem o fluxo de execução da coordenação dos componentes.

O Capítulo 2 introduziu uma classificação para os modelos de armazenamento de dados em RSSFs. Esta classificação possibilitou tratar da **separação de conceitos**, associando entidades e funcionalidades que são comuns a vários sistemas. A escolha do modelo de armazenamento depende do contexto da aplicação. Porém nota-se que o modelo de armazenamento agrupado em *repositórios* oferece um compromisso entre o custo da consulta e o custo do armazenamento dos dados. Além disso, foram apontadas abordagens para minimizar o tráfego de transmissão da rede, como as técnicas de predição de dados e de formação de *clusters* que exploram a similaridade de dados. Por fim, apontou-se o uso de técnicas modernas de desenvolvimento de software para reduzir o esforço de análise, codificação e testes de códigos de simulação para modelos de armazenamento de RSSFs, que integram reuso de componentes de software e máquina de estados para descrever o controle de fluxo do sistema.

O Capítulo 3 apresentou o AQPM, um modelo de armazenamento para tratar a dinamicidade de geração de dados da rede, de maneira autônoma e escalável. O AQPM realiza a formação de *clusters* com base na similaridade dos dados aliado ao conceito de *repositórios*, que armazenam dados de um conjunto de *clusters*. Portanto, o AQPM é **escalável** em termos do número de transmissões trafegadas na rede, reduzindo o consumo de energia durante o processamento de consultas. Além disso o AQPM é autônomo, pois não depende de uma entidade externa para realizar a formação dos *clusters* e a escolha dos *repositórios* de dados é feita pela própria rede. Nas avaliações, o AQPM foi mais eficiente que o IBIS (da Silva et al., 2011) em relação ao consumo de energia, e mostrou bons resultados com relação ao tempo de resposta e ao erro médio do resultado.

O Capítulo 4 apresentou um modelo de desenvolvido baseado em componentes de software, motivado pela dificuldade de reuso de código em novas instâncias de sistemas de armazenamento. O modelo, chamado de RCBM, aplica **separação de conceitos**, identificando entidades e funcionalidades que são comuns a vários sistemas de armazenamento, e é aplicável a redes de sensores em larga escala. Os experimentos mostraram que a reutilização de código do modelo MAX-MIN é melhor no RCBM quando comparado com o CBCWSN(Amaxilatis et al., 2011b), e um pouco inferior com relação aos modelos LEACH e LCA. Além disso, observou-se que os resultados obtidos pelo programa gerado utilizando-se a plataforma RCBM

foram condizentes com o que foi apresentado no trabalho original (Amis et al., 2000), o que corrobora a validade da solução implementada.

O Capítulo 5 propôs uma **especificação formal** que usa máquina de estados para descrever a coordenação do fluxo de execução dos componentes do RCBM. Através de um estudo de caso e a sua codificação no simulador NS2, mostrou-se que há uma correspondência direta entre a máquina proposta e o programa desenvolvido. Esta correspondência entre o conceito e o código facilita o desenvolvimento do coordenador. A integração da especificação da coordenação utilizando a máquina de estados e o *framework* RCBM foram utilizados para implementar o modelo DCSSC(Le et al., 2008), que possui uma coordenação mais complexa que os estudos de caso apresentados usadando o *framework* RCBM.

O Capítulo 6 apresentou a especificação de uma linguagem de geração de código para o coordenador do modelo de componentes RCBM. A linguagem possui uma sintaxe orientada a máquina de estados, que é um modelo usado por desenvolvedores para representar o fluxo de execução do coordenador no ambiente de simulação NS2. A geração de código de SLEDS para NS2 não é trivial, mas um estudo de caso da coordenação de armazenamento de dados em redes de sensores mostrou a viabilidade de uso da linguagem.

7.1 TRABALHOS FUTUROS

Além das contribuições apresentadas anteriormente, foram identificadas várias questões de pesquisa interessantes que merecem ser investigadas para dar continuidade a este trabalho. Estas questões incluem estudos que envolvem o desenvolvimento de novos modelos de armazenamento, técnicas de componentes de software para reuso de código, modelos de máquina de estados para coordenação de componentes e linguagens de coordenação para especificar e implementar os modelos de máquina de estados.

1. **Novos modelos de armazenamento em redes urbanas**
   O modelo AQPM, descrito no Capítulo 3, apresentou uma estratégia de armazenamento em estruturas denominadas de *repositórios*, para minimizar o custo do processamento de consultas. Nesta abordagem, a leitura reportada pelo *repositório* representa a leitura do seu agrupamento. Portanto, é necessário uma análise mais aprofundada nos seguintes detalhes:

   - *Erro médio:* o AQPM realiza este cálculo a partir das leituras de 5 atributos de sensoriamento, o que não ocorre nos modelos tradicionais, onde a média é calculada a partir de um único atributo do ambiente. Nesse sentido, pode-se estudar novas estratégias para minimizar o erro, por exemplo, criando-se critérios para calibrar o **peso** com o tipo de leitura.
   - *Comparação com outros modelos:* as técnicas de predição de dados estão sendo bastante exploradas em redes urbanas. Assim, pretende-se realizar experimentos com modelos que exploram a predição de dados com o objetivo de comparar os resultados com o AQPM.

2. ***Framework* de componentes**
   O *Framework* RCBM, apresentado no Capítulo 4, descreveu uma abordagem de desenvolvimento de componentes e reuso de código para modelos da classe *repositórios*. Embora foram identificados os principais componentes comuns desta classe, novos componentes precisam ser identificados para se estabelecer uma biblioteca de componentes para

serem reutilizados no desenvolvimento de novos sistemas de armazenamento. Logo, a proposta pode ser estendida para:

- *Considerar outras classes de armazenamento:* Identificar novas entidades dos modelos de armazenamento das classes externo, local e *repositórios*, e associá-las a um conjunto de componentes que implementam as funcionalidades comuns aos modelos destas classes.
- *Considerar outros cenários de aplicação:* A especificação atual dos componentes considerou o desenvolvimento de suas funcionalidades no simulador de redes NS2. Pretende-se considerar o desenvolvimento dos componentes para o simulador NS3, que é a versão atual mais intuitiva que o NS2, bem como para redes reais.

3. **Modelo de máquina de estados**
   A máquina de estados especificada no Capítulo 5, descreveu um modelo formal para a coordenação do fluxo de execução em um modelo de componentes de software. A especificação atual da máquina está de acordo com a programação orientada a eventos do simulador NS2. No entanto, pretende-se verificar se o modelo atual da máquina oferece suporte para:

   - *Especificar programas desenvolvidos para outros contextos:* Estender o modelo para especificar códigos para o simulador NS3 bem como para redes reais.
   - *Outros modelos de armazenamento:* Realizar novos estudos de caso com outros modelos da classe *repositório* bem como para as classes de armazenamento externo e local.
   - *Novos experimentos:* Realizar estudos experimentais adicionais envolvendo outras métricas.
   - *Eventos críticos:* Desenvolver estudos de caso da máquina para especificar código de sistemas críticos.

4. **Linguagem de coordenação de componentes**
   A especificação atual da linguagem SLEDS, vista no Capítulo 6, descreveu em detalhes a sintaxe, a semântica e um esquema de tradução para gerar códigos de simulação para o simulador NS2. As questões que precisam ser exploradas são:

   - *Tradução automática:* Implementar o tradutor da linguagem SLEDS, baseado na especificação atual, para gerar automaticamente o código de simulação do coordenador para o simulador NS2.
   - *Outros ambientes:* Estender a especificação atual considerando outros ambientes, como a tradução de SLEDS para o simulador NS3, uma evolução mais intuitiva do NS2, bem como para outros simuladores e redes reais.

7.2 COLABORAÇÕES

Muitas pessoas contribuíram de maneira efetiva para que este trabalho pudesse ser concluído. A colaboração do Dr. Rone Ilídio com o desenvolvimento e a parte experimental do AQPM, foi fundamental para a validação do modelo apresentado no capítulo 3. A Katriny Zamproni contribuiu com a parte da máquina de estados do capítulo 5. O Prof. Martin Musicante colaborou no modelo de componentes e na especificação da linguagem, referentes aos capítulos 4 e 6, respectivamente.

7.3 LISTA DE PUBLICAÇÕES RELACIONADAS À TESE

Carrero, M. A., da Silva, R. I., dos Santos, A. L. e Hara, C. S. (2015). An Autonomic Spatial Query Processing Model for Urban Sensor Networks. *Em XXXIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC), 2015. IEEE*. páginas 81-90.

Carrero, M. A., da Silva, R. I., dos Santos, A. L. e Hara, C. S. (2015). An autonomic in-network query processing for urban sensor networks. *Em Computers and Communication (ISCC), 2015. IEEE*, páginas 968–973.

Carrero, M. A., Santos, A. L. d., Musicante, M. A. e Hara, C. S. (2017). A Reusable Component-based Model for WSN Storage Simulation. *Em: Proceedings of the 13th ACM Symposium on QoS and Security for Wireless and Mobile Networks*, páginas 31–38.

Carrero, M. A., Zamproni, K., Musicante, M. A., dos Santos, A. L. e Hara, C. S. (2018). Uma Máquina de Estados para Especificação de Códigos de Simulação para Redes de Sensores sem Fio Urbanas. *Em X Simpósio Brasileiro de Computação Ubíqua e Pervasiva (SBCUP), 2018*.

Carrero, M. A., Musicante, M. A., dos Santos, A. L. e Hara, C. S. (2018). SLEDS: A DSL for Data-Centric Storage on Wireless Sensor Networks. *Em Workshop on Big Social Data and Urban Computing (BIDU), 2018*. Submetido à publicação.

REFERÊNCIAS

Abbasi, A. A. e Younis, M. (2007). A survey on clustering algorithms for wireless sensor networks. *Computer communications*, 30(14):2826–2841.

Afsar, M. M. e Tayarani-N, M.-H. (2014). Clustering in sensor networks: A literature survey. *Journal of Network and Computer Applications*, 46:198–226.

Ahmed, K. e Gregory, M. A. (2012). Techniques and challenges of data centric storage scheme in wireless sensor network. *Journal of Sensor and Actuator Networks*, 1(1):59–85.

Aho, A. V., Lam, M. S., Sethi, R. e Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Albano, M. e Chessa, S. (2015). Replication vs erasure coding in data centric storage for wireless sensor networks. *Computer Networks*, 77:42–55.

Ali, M. S., Babar, M. A., Chen, L. e Stol, K.-J. (2010). A systematic review of comparative evidence of aspect-oriented programming. *Information and software Technology*, 52(9):871–887.

Alliance, O. (2007). About the osgi service platform, technical whitepaper, revision 4.1. *June*.

Amaxilatis, D., Chatzigiannakis, I., Dolev, S., Koninis, C., Pyrgelis, A. e Spirakis, P. G. (2011a). Adaptive hierarchical network structures for wireless sensor networks. Em *Ad Hoc Networks - Third International ICST Conference, ADHOCNETS 2011, Paris, France, September 21-23, 2011, Revised Selected Papers*, páginas 65–80.

Amaxilatis, D., Chatzigiannakis, I., Koninis, C. e Pyrgelis, A. (2011b). Component based clustering in wireless sensor networks. *arXiv preprint arXiv:1105.3864*.

Amis, A. D., Prakash, R., Vuong, T. H. P. e Huynh, D. T. (2000). Max-min d-cluster formation in wireless ad hoc networks. Em *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*, páginas 32–41.

Anastasi, G., Conti, M., Di Francesco, M. e Passarella, A. (2009). Energy conservation in wireless sensor networks: A survey. *Ad hoc networks*, 7(3):537–568.

Araújo, G., Tostes, A., de LP Duarte-Figueiredo, F. e Loureiro, A. (2014). Um protocolo de identificação e minimização de congestionamentos de tráfego para redes veiculares. *XXXII Simp. Bras. de Redes de Computadores e Sistemas Distribuídos*, páginas 207–220.

Baker, D. J. e Ephremides, A. (1981). A distributed algorithm for organizing mobile radio telecommunication networks. Em *Proc. of the IEEE International Conference on Distributed Systems (ICDCS)*, páginas 476–483.

Barrenetxea, G., Ingelrest, F., Schaefer, G., Vetterli, M., Couach, O. e Parlange, M. (2008). Sensorscope: Out-of-the-box environmental monitoring. Em *Information Processing in Sensor Networks, 2008. IPSN'08. International Conference on*, páginas 332–343. IEEE.

Baumgartner, T., Chatzigiannakis, I., Fekete, S., Koninis, C., Kroller, A. e Pyrgelis, A. (2010). Wiselib: A generic algorithm library for heterogeneous sensor networks. *EWSN: LNCS*, 5970:162–177.

Boulis, A., Ganeriwal, S. e Srivastava, M. B. (2003). Aggregation in sensor networks: an energy-accuracy trade-off. Em *Proceedings of the First IEEE International Workshop on Sensor Network Protocols and Applications, 2003.*, páginas 128–138.

Can, Z. e Demirbas, M. (2013). A survey on in-network querying and tracking services for wireless sensor networks. *Ad Hoc Networks*, 11(1):596–610.

Canal, C., Poizat, P. e Salaün, G. (2008). Model-based adaptation of behavioral mismatching components. *IEEE Transactions on Software Engineering*, 34(4):546–563.

Cañete, E., Chen, J., Díaz, M., Llopis, L. e Rubio, B. (2011). A service-oriented approach to facilitate wsan application development. *Ad Hoc Networks*, 9(3):430–452.

Carrero, M., Zamproni, K., Musicante, M. A., Santos, A. e Hara, C. (2018a). Uma máquina de estados para especificação de códigos de simulação para redes de sensores sem fio urbanas. Em *Simpósio Brasileiro de Computação Ubíqua e Pervasiva*.

Carrero, M. A., da Silva, R. I., dos Santos, A. L. e Hara, C. S. (2015a). An autonomic in-network query processing for urban sensor networks. Em *Computers and Communication (ISCC), 2015 IEEE Symposium on*, páginas 968–973. IEEE.

Carrero, M. A., da Silva, R. I., dos Santos, A. L. e Hara, C. S. (2015b). An autonomic spatial query processing model for urban sensor networks. Em *XXXIII Brazilian Symposium on Computer Networks and Distributed Systems, SBRC 2015, Vitória, Brazil, May 18-22, 2015*, páginas 81–90.

Carrero, M. A., Musicante, M. A., dos Santos, A. L. e Hara, C. S. (2017). A reusable component-based model for wsn storage simulation. Em *Proceedings of the 13th ACM Symposium on QoS and Security for Wireless and Mobile Networks*, páginas 31–38. ACM.

Carrero, M. A., Musicante, M. A., dos Santos, A. L. e Hara, C. S. (2018b). Sleds: A dsl for data-centric storage on wireless sensor networks. Workshop on Big Social Data and Urban Computing, BIDU. Submetido à publicação.

Cecílio, J. e Furtado, P. (2012). A state-machine model for reliability eliciting over wireless sensor and actuator networks. *Procedia Computer Science*, 10:422–431.

Ceriotti, M., Mottola, L., Picco, G. P., Murphy, A. L., Guna, S., Corra, M., Pozzi, M., Zonta, D. e Zanon, P. (2009). Monitoring heritage buildings with wireless sensor networks: The torre aquila deployment. Em *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, páginas 277–288. IEEE Computer Society.

Chandra, T. B. e Dwivedi, A. K. (2015). Programming languages for wireless sensor networks: A comparative study. Em *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, páginas 1702–1708.

Chatterjea, S. e Havinga, P. (2007). A taxonomy of distributed query management techniques for wireless sensor networks. *International Journal of Communication Systems*, 20(7):889–908.

Chen, M., Mao, S. e Liu, Y. (2014). Big data: A survey. *Mobile Networks and Applications*, 19(2):171–209.

Cheng, B., Xu, Z., Chen, C. e Guan, X. (2011). Spatial correlated data collection in wireless sensor networks with multiple sinks. Em *Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on*, páginas 578–583. IEEE.

Cheong, E., Liebman, J., Liu, J. e Zhao, F. (2003). Tinygals: A programming model for event-driven embedded systems. Em *Proceedings of the 2003 ACM Symposium on Applied Computing*, SAC '03, páginas 698–704, New York, NY, USA. ACM.

Chong, S., Gaber, M., Krishnaswamy, S. e Loke, S. (2011). Energy-aware data processing techniques for wireless sensor networks: a review. *Transactions on large-scale data-and knowledge-centered systems III*, páginas 117–137.

Clements, P. e Northrop, L. (2015). *Software Product Lines: Practices and Patterns: Practices and Patterns*. The SEI series in software engineering. Addison-Wesley.

Coman, A., Sander, J. e Nascimento, M. A. (2007). Adaptive processing of historical spatial range queries in peer-to-peer sensor networks. *Distributed and Parallel Databases*, 22(2-3):133–163.

Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J. e Sivaharan, T. (2008). A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26(1):1:1–1:42.

Coulson, G., Porter, B., Chatzigiannakis, I., Koninis, C., Fischer, S., Pfisterer, D., Bimschas, D., Braun, T., Hurni, P., Anwander, M. et al. (2012). Flexible experimentation in wireless sensor networks. *Communications of the ACM*, 55(1):82–90.

Cremonezi, B. M., Vieira, A. B., Nogueira, M. e Nacif, J. A. (2017). Um protocolo de alocação dinâmica de canais para ambientes médicos sob múltiplas estações base. Em *XXXV Simp. Bras. de Redes de Computadores e Sistemas Distribuídos*, páginas 272–285.

Crnkovic, I., Sentilles, S., Vulgarakis, A. e Chaudron, M. R. V. (2011). A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5):593–615.

D´ Angelo, G., Diodati, D., Navarra, A. e Pinotti, C. M. (2016). The minimum k-storage problem: Complexity, approximation, and experimental analysis. *IEEE Transactions on Mobile Computing*, 15(7):1797–1811.

da Silva, R. I., Macedo, D. F. e Nogueira, J. M. S. (2011). Contornos irregulares no processamento de requisições espaciais para redes de sensores sem fio. Em *XXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*.

Da Silva, R. I., Macedo, D. F. e Nogueira, J. M. S. (2014). Spatial query processing in wireless sensor networks–a survey. *Information Fusion*, 15:32–43.

Desai, A., Gupta, V., Jackson, E., Qadeer, S., Rajamani, S. e Zufferey, D. (2013). P: safe asynchronous event-driven programming. *ACM SIGPLAN Notices*, 48(6):321–332.

Desnoyers, P., Ganesan, D. e Shenoy, P. (2005). Tsar: a two tier sensor storage architecture using interval skip graphs. Em *Proceedings of the 3rd international conference on Embedded networked sensor systems*, páginas 39–50. ACM.

Diallo, O., Rodrigues, J. J., Sene, M. e Lloret, J. (2015). Distributed database management techniques for wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems*, 26(2):604–620.

Dunkels, A., Gronvall, B. e Voigt, T. (2004). Contiki-a lightweight and flexible operating system for tiny networked sensors. Em *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, páginas 455–462. IEEE.

Dunkels, A., Schmidt, O., Voigt, T. e Ali, M. (2006). Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. Em *Proceedings of the 4th international conference on Embedded networked sensor systems*, páginas 29–42. Acm.

Erl, T. (2005). *Service-Oriented Architecture: Concepts, Technology, and Design*. The Prentice Hall Service Technology Series from Thomas Erl. Pearson Education.

Estrin, D., Govindan, R., Heidemann, J. S. e Kumar, S. (1999). Next century challenges: Scalable coordination in sensor networks. Em Kodesh, H., Bahl, V., Imielinski, T. e Steenstrup, M., editores, *MobiCom*, páginas 263–270. ACM.

Figueiredo, C. M. S., dos Santos, A. L., Loureiro, A. A. F. e Nogueira, J. M. S. (2005). Policy-based adaptive routing in autonomous wsns. Em *Ambient Networks, 16th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM 2005, Barcelona, Spain, October 24-26, 2005, Proceedings*, páginas 206–219.

Filho, G. P., Ueyama, J., Faiçal, B. S., Guidoni, D. L., Villas, L. A. e Carlos-SP-Brasil, S. (2015). Residi: Um sistema de decisão inteligente para infraestruturas residenciais via sensores e atuadores sem fio. Em *XXXIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, páginas 53–66.

Filipponi, L., Santini, S. e Vitaletti, A. (2008). Data collection in wireless sensor networks for noise pollution monitoring. Em *Distributed Computing in Sensor Systems*, páginas 492–497.

Filman, R. (2005). *Aspect Oriented Software Development*. Addison-Wesley.

Fischer, J., Majumdar, R. e Millstein, T. (2007). Tasks: language support for event-driven programming. Em *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, páginas 134–143. ACM.

Fowler, M. (2010). *Domain Specific Languages*. Addison-Wesley Professional, 1st edition.

Furlaneto, S. S., Dos Santos, A. e Hara, C. S. (2012). An efficient data acquisition model for urban sensor networks. Em *Network Operations and Management Symposium (NOMS), 2012 IEEE*, páginas 113–120. IEEE.

Gabbrielli, M. e Martini, S. (2010). *Programming Languages: Principles and Paradigms*. Undergraduate Topics in Computer Science. Springer London.

Gielow, F., Jakllari, G., Nogueira, M. e Santos, A. (2015). Data similarity aware dynamic node clustering in wireless sensor networks. *Ad Hoc Networks*, 24:29–45.

Gielow, F., Nogueira, M. e Santos, A. (2014). Data similarity aware dynamic nodes clustering for supporting management operations. Em *Network Operations and Management Symposium (NOMS), 2014 IEEE*, páginas 1–8. IEEE.

Gielow, F. H. e dos Santos, A. L. (2009). Um protocolo de roteamento baseado em clusters desiguais para minimizar hot spots em rssf. Em *XIV Workshop de Gerência e Operação de Redes e Serviços (WGRS 2009)*, páginas 140–153.

Gil, T. M. e Madden, S. (2007). Scoop: An adaptive indexing scheme for stored data in sensor networks. Em *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, páginas 1345–1349. IEEE.

Glombitza, N., Pfisterer, D. e Fischer, S. (2010). Using state machines for a model driven development of web service-based sensor network applications. Em *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*, páginas 2–7. ACM.

Gonçalves, N. M., dos Santos, A. L. e Hara, C. S. (2012). Dysto-a dynamic storage model for wireless sensor networks. *Journal of Information and Data Management*, 3(3):147.

Gonçalves, N. M. F., dos Santos, A. L. e Hara, C. S. (2014). A policy-based storage model for sensor networks. Em *2014 IEEE Network Operations and Management Symposium, NOMS 2014, Krakow, Poland, May 5-9, 2014*, páginas 1–8.

Gupta, H., Navda, V., Das, S. e Chowdhary, V. (2008). Efficient gathering of correlated data in sensor networks. *ACM Transactions on Sensor Networks (TOSN)*, 4(1):4.

Hartung, C., Han, R., Seielstad, C. e Holbrook, S. (2006). Firewxnet: A multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments. Em *Proceedings of the 4th international conference on Mobile systems, applications and services*, páginas 28–41. ACM.

Heinzelman, W. R., Chandrakasan, A. e Balakrishnan, H. (2000). Energy-efficient communication protocol for wireless microsensor networks. Em *Proc. of the 33rd IEEE Hawaii International Conference on System Sciences (HICSS)*.

Horneber, J. e Hergenröder, A. (2014). A survey on testbeds and experimentation environments for wireless sensor networks. *IEEE Communications Surveys and Tutorials*, 16(4):1820–1838.

Hung, C.-C., Peng, W.-C. e Lee, W.-C. (2012). Energy-aware set-covering approaches for approximate data collection in wireless sensor networks. *Knowledge and Data Engineering, IEEE Transactions on*, 24(11):1993–2007.

Huynh, D. F., Karger, D. R. e Miller, R. C. (2007). Exhibit: lightweight structured data publishing. Em *Proceedings of the 16th international conference on World Wide Web*, páginas 737–746.

Jiang, H., Jin, S. e Wang, C. (2011). Prediction or not? an energy-efficient framework for clustering-based data collection in wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):1064–1071.

Jo, K., Kim, J., Kim, D., Jang, C. e Sunwoo, M. (2015). Development of autonomous car, part ii: A case study on the implementation of an autonomous driving system based on distributed architecture. *IEEE Transactions on Industrial Electronics*, 62(8):5119–5132.

Jr., E. P. D. e dos Santos, A. L. (2001). Semi-active replication of SNMP objects in agent groups applied for fault management. Em *2001 IEEE/IFIP International Symposium on Integrated Network Management, IM 2001, Seattle, USA, May 14-18, 2001. Proceedings*, páginas 565–578.

Karp, B. e Kung, H.-T. (2000). Gpsr: Greedy perimeter stateless routing for wireless networks. Em *Proceedings of the 6th annual international conference on Mobile computing and networking*, páginas 243–254. ACM.

Kasten, O. e Römer, K. (2005). Beyond event handlers: Programming wireless sensors with attributed state machines. Em *Proceedings of the 4th Int. Symp. on Information processing in sensor networks*, página 7. IEEE Press.

Khan, M. Z., Askwith, B., Bouhafs, F. e Asim, M. (2011). Limitations of simulation tools for large-scale wireless sensor networks. Em *25th IEEE International Conference on Advanced Information Networking and Applications Workshops, WAINA 2011, Biopolis, Singapore, March 22-25, 2011*, páginas 820–825.

Kim, S., Pakzad, S., Culler, D., Demmel, J., Fenves, G., Glaser, S. e Turon, M. (2007). Health monitoring of civil infrastructures using wireless sensor networks. Em *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, páginas 254–263.

Kim, Y., Schmid, T., Charbiwala, Z. M., Friedman, J. e Srivastava, M. B. (2008). Nawms: nonintrusive autonomous water monitoring system. Em *Proceedings of the 6th ACM conference on Embedded network sensor systems*, páginas 309–322. ACM.

Kolcun, R., Boyle, D. E. e McCann, J. A. (2016). Efficient distributed query processing. *IEEE Transactions on Automation Science and Engineering*, 13(3):1230–1246.

Krämer, M., Bader, S. e Oelmann, B. (2013). Implementing wireless sensor network applications using hierarchical finite state machines. Em *10th IEEE Int. Conf. on Networking, Sensing and Control*, páginas 124–129. IEEE.

Kulkarni, R. V., Forster, A. e Venayagamoorthy, G. K. (2011). Computational intelligence in wireless sensor networks: A survey. *IEEE communications surveys & tutorials*, 13(1):68–96.

Kushwaha, M., Amundson, I., Koutsoukos, X., Neema, S. e Sztipanovits, J. (2007). Oasis: A programming framework for service-oriented sensor networks. Em *2007 2nd International Conference on Communication Systems Software and Middleware*.

Le, T. D., Pham, N. D. e Choo, H. (2008). Towards a distributed clustering scheme based on spatial correlation in wsns. Em *Wireless Communications and Mobile Computing Conference, 2008. IWCMC'08. International*, páginas 529–534. IEEE.

Lee, E. K., Viswanathan, H. e Pompili, D. (2015). Distributed data-centric adaptive sampling for cyber-physical systems. *TAAS*, 9(4):21:1–21:27.

Li, M., Ganesan, D. e Shenoy, P. (2009). Presto: Feedback-driven data management in sensor networks. *IEEE/ACM Transactions on Networking (TON)*, 17(4):1256–1269.

Lima, M. N., dos Santos, A. L. e Pujolle, G. (2009). A survey of survivability in mobile ad hoc networks. *IEEE Communications Surveys and Tutorials*, 11(1):66–77.

Louden, K. (2004). *Compiladores - Princípios e Práticas*. Pioneira Thomson Learning.

Ma, Y., Guo, Y., Tian, X. e Ghanem, M. (2011). Distributed clustering-based aggregation algorithm for spatial correlated sensor networks. *Sensors Journal, IEEE*, 11(3):641–648.

Macedo, D. F., dos Santos, A. L., Nogueira, J. M. S. e Pujolle, G. (2009). A distributed information repository for autonomic context-aware manets. *IEEE Trans. Network and Service Management*, 6(1):45–55.

Maia, J. E. B. e Brayner, A. (2013). Sensor-field modeling based on in-network data prediction: an efficient strategy for answering complex queries in wireless sensor networks. Em *Proceedings of the 28th annual ACM symposium on applied computing*, páginas 554–559. ACM.

Malavolta, I. e Muccini, H. (2014). A study on mde approaches for engineering wireless sensor networks. Em *Software engineering and advanced applications (SEAA), 2014 40th EUROMICRO conference on*, páginas 149–157. IEEE.

Mannes, E., Nogueira, M. e dos Santos, A. L. (2012a). A bio-inspired scheme on quorum systems for reliable services data management in manets. Em *2012 IEEE Network Operations and Management Symposium, NOMS 2012, Maui, HI, USA, April 16-20, 2012*, páginas 278–285.

Mannes, E., Nogueira, M. e dos Santos, A. L. (2012b). Reliable operational services in manets by misbehavior-tolerant quorum systems. Em *8th International Conference on Network and Service Management, CNSM 2012, Las Vegas, NV, USA, October 22-26, 2012*, páginas 343–349.

Mao, X., Miao, X., He, Y., Li, X.-Y. e Liu, Y. (2012). Citysee: Urban co 2 monitoring with sensors. Em *INFOCOM, 2012 Proceedings IEEE*, páginas 1611–1619. IEEE.

Marin, C. e Desertot, M. (2005). Sensor bean: A component platform for sensor-based services. Em *Proceedings of the 3rd International Workshop on Middleware for Pervasive and Ad-hoc Computing*, MPAC '05, páginas 1–8, New York, NY, USA. ACM.

Masrur, A., Kit, M., Matěna, V., Bureš, T. e Hardt, W. (2016). Component-based design of cyber-physical applications with safety-critical requirements. *Microprocessors and Microsystems*, 42:70–86.

Matthys, N., Hughes, D., Michiels, S., Huygens, C. e Joosen, W. (2009). Fine-grained tailoring of component behaviour for embedded systems. Em *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*, páginas 156–167. Springer.

Minakov, I., Passerone, R., Rizzardi, A. e Sicari, S. (2016). A comparative study of recent wireless sensor network simulators. *ACM Transactions on Sensor Networks (TOSN)*, 12(3):20.

Muller, C. L., Chapman, L., Grimmond, C., Young, D. T. e Cai, X. (2013). Sensors and the city: a review of urban meteorological networks. *International Journal of Climatology*, 33(7):1585–1600.

Muñoz, C. e Leone, P. (2017). A distributed event-based system based on compressed fragmented-iterated bloom filters. *Future Generation Computer Systems*, 75:108–127.

Murr, F. e Mauerer, W. (2017). Mcfsm: Globally taming complex systems. Em *Proceedings of the 3rd International Workshop on Software Engineering for Smart Cyber-Physical Systems*, SEsCPS '17, páginas 26–29, Piscataway, NJ, USA. IEEE Press.

Newton, R., Morrisett, G. e Welsh, M. (2007). The regiment macroprogramming system. Em *2007 6th International Symposium on Information Processing in Sensor Networks*, páginas 489–498.

Niekamp, R. (2005). Software component architecture. Em *Gestión de Congresos-CIMNE/Institute for Scientific Computing, TU Braunschweig*, página 4.

Patel, P. e Cassou, D. (2015). Enabling high-level application development for the internet of things. *Journal of Systems and Software*, 103:62–84.

Patel, P., Pathak, A., Teixeira, T. e Issarny, V. (2011). Towards application development for the internet of things. Em *Proceedings of the 8th Middleware Doctoral Symposium*, página 5. ACM.

Poizat, P., Royer, J.-C. e Salaün, G. (2004). Formal methods for component description, coordination and adaptation. Em *First International Workshop on Coordination and Adaptation Techniques for Software Entities*, páginas 89–100.

Ratnasamy, S., Karp, B., Yin, L., Yu, F., Estrin, D., Govindan, R. e Shenker, S. (2002). Ght: a geographic hash table for data-centric storage. Em *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, páginas 78–87. ACM.

Rawat, P., Singh, K. D., Chaouchi, H. e Bonnin, J. M. (2014). Wireless sensor networks: a survey on recent developments and potential synergies. *The Journal of Supercomputing*, 68(1):1–48.

Resch, B., Mittlboeck, M., Girardin, F., Britter, R. e Ratti, C. (2009). Live geography–embedded sensing for standarised urban environmental monitoring. *International Journal on Advances in Systems and Measurements*, páginas 156–167.

Riley, G. F. e Henderson, T. R. (2010). The ns-3 network simulator. Em *Modeling and tools for network simulation*, páginas 15–34. Springer.

Rumín, Á. C., Pascual, M. U., Ortega, R. R. e López, D. L. (2010). Data centric storage technologies: Analysis and enhancement. *Sensors*, 10(4):3023–3056.

Salman, A. J. e Al-Yasiri, A. (2016). Sennet: a programming toolkit to develop wireless sensor network applications. Em *New Technologies, Mobility and Security (NTMS), 2016 8th IFIP International Conference on*, páginas 1–7. IEEE.

Selavo, L., Wood, A., Cao, Q., Sookoor, T., Liu, H., Srinivasan, A., Wu, Y., Kang, W., Stankovic, J., Young, D. et al. (2007). Luster: wireless sensor network for environmental research. Em *Proceedings of the 5th international conference on Embedded networked sensor systems*, páginas 103–116. ACM.

Shen, H., Zhao, L. e Li, Z. (2011). A distributed spatial-temporal similarity data storage scheme in wireless sensor networks. *Mobile Computing, IEEE Transactions on*, 10(7):982–996.

Sheng, B., Li, Q. e Mao, W. (2010). Optimize storage placement in sensor networks. *Mobile Computing, IEEE Transactions on*, 9(10):1437–1450.

Shenker, S., Ratnasamy, S., Karp, B., Govindan, R. e Estrin, D. (2003). Data-centric storage in sensornets. *ACM SIGCOMM Computer Communication Review*, 33(1):137–142.

Skordylis, A., Trigoni, N. e Guitton, A. (2006). A study of approximate data management techniques for sensor networks. Em *Intelligent Solutions in Embedded Systems, 2006 International Workshop on*, páginas 1–12. IEEE.

Song, W.-Z., Huang, R., Xu, M., Ma, A., Shirazi, B. e LaHusen, R. (2009). Air-dropped sensor network for real-time high-fidelity volcano monitoring. Em *Proceedings of the 7th international conference on Mobile systems, applications, and services*, páginas 305–318. ACM.

Subramanian, N., Yang, C. e Zhang, W. (2007). Securing distributed data storage and retrieval in sensor networks. *Pervasive and Mobile Computing*, 3(6):659–676.

Sundani, H., Li, H., Devabhaktuni, V., Alam, M. e Bhattacharya, P. (2011). Wireless sensor network simulators a survey and comparisons. *International Journal of Computer Networks*, 2(5):249–265.

Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

Taherkordi, A., Johansen, C., Eliassen, F. e Römer, K. (2015). Tokenit: Designing state-driven embedded systems through tokenized transitions. Em *2015 Int. Conf. on Distributed Computing in Sensor Systems*, páginas 52–61. IEEE.

Takaishi, D., Nishiyama, H., Kato, N. e Miura, R. (2014). Toward energy efficient big data gathering in densely distributed sensor networks. *Emerging Topics in Computing, IEEE Transactions on*, 2(3):388–397.

Tei, K., Shimizu, R., Fukazawa, Y. e Honiden, S. (2015). Model-driven-development-based stepwise software development process for wireless sensor networks. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 45(4):675–687.

Thepvilojanapong, N., Ono, T. e Tobe, Y. (2010). A deployment of fine-grained sensor network and empirical analysis of urban temperature. *Sensors*, 10(3):2217–2241.

Tolle, G., Polastre, J., Szewczyk, R., Culler, D., Turner, N., Tu, K., Burgess, S., Dawson, T., Buonadonna, P., Gay, D. et al. (2005). A macroscope in the redwoods. Em *Proceedings of the 3rd international conference on Embedded networked sensor systems*, páginas 51–63. ACM.

Tubaishat, M., Yin, J., Panja, B. e Madria, S. (2004). A secure hierarchical model for sensor network. *ACM Sigmod Record*, 33(1):7–13.

Vale, T., Crnkovic, I., de Almeida, E. S., Neto, P. A. d. M. S., Cavalcanti, Y. C. e de Lemos Meira, S. R. (2016). Twenty-eight years of component-based software engineering. *Journal of Systems and Software*, 111:128–148.

van Deursen, A., Klint, P. e Visser, J. (2000). Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36.

Varga, A. e Hornig, R. (2008). An overview of the omnet++ simulation environment. Em *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, página 60. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Villas, L. A., Boukerche, A., De Oliveira, H. A., De Araujo, R. B. e Loureiro, A. A. (2014). A spatial correlation aware algorithm to perform efficient data collection in wireless sensor networks. *Ad Hoc Networks*, 12:69–85.

Vuran, M. C., Akan, Ö. B. e Akyildiz, I. F. (2004). Spatio-temporal correlation: theory and applications for wireless sensor networks. *Computer Networks*, 45(3):245–259.

Wang, F. e Liu, J. (2011). Networked wireless sensor data collection: issues, challenges, and approaches. *Communications Surveys & Tutorials, IEEE*, 13(4):673–687.

Welke, R., Hirschheim, R. e Schwarz, A. (2011). Service-oriented architecture maturity. *Computer*, 44(2):61–67.

Wen, T.-H., Jiang, J.-A., Sun, C.-H., Juang, J.-Y. e Lin, T.-S. (2013). Monitoring street-level spatial-temporal variations of carbon monoxide in urban settings using a wireless sensor network (wsn) framework. *International journal of environmental research and public health*, 10(12):6380–6396.

Werner-Allen, G., Lorincz, K., Johnson, J., Lees, J. e Welsh, M. (2006). Fidelity and yield in a volcano monitoring sensor network. Em *Proceedings of the 7th symposium on Operating systems design and implementation*, páginas 381–396. USENIX Association.

Whitehouse, K., Sharp, C., Brewer, E. e Culler, D. (2004). Hood: A neighborhood abstraction for sensor networks. Em *Proceedings of the 2Nd International Conference on Mobile Systems, Applications, and Services*, MobiSys '04, páginas 99–110. ACM.

Wu, Y. e Li, Y. (2009). Distributed indexing and data dissemination in large scale wireless sensor networks. Em *Computer Communications and Networks, 2009. ICCCN 2009. Proceedings of 18th Internatonal Conference on*, páginas 1–6. IEEE.

Xie, L., Lu, S., Cao, Y. e Chen, D. (2014). Towards energy-efficient storage placement in large scale sensor networks. *Frontiers of Computer Science*, páginas 1–17.

Xu, J., Guo, S., Xiao, B. e He, J. (2015). Energy-efficient big data storage and retrieval for wireless sensor networks with nonuniform node distribution. *Concurrency and Computation: Practice and Experience*, 27(18):5765–5779.

Yang, K.-C., Yang, Y.-C., Lin, C.-L. e Wang, J.-S. (2010). Hierarchical data management for spatial-temporal information in wsns. Em *Sensor Technologies and Applications (SENSORCOMM), 2010 Fourth International Conference on*, páginas 435–440. IEEE.

Yick, J., Mukherjee, B. e Ghosal, D. (2008). Wireless sensor network survey. *Computer networks*, 52(12):2292–2330.

Yoon, S. e Shahabi, C. (2007). The clustered aggregation (cag) technique leveraging spatial and temporal correlations in wireless sensor networks. *ACM Transactions on Sensor Networks (TOSN)*, 3(1):3.

Yu, Z., Xiao, B. e Zhou, S. (2010). Achieving optimal data storage position in wireless sensor networks. *Computer Communications*, 33(1):92–102.