Revista Brasileira de Computação Aplicada, Abril, 2022

DOI: 10.5335/rbca.v14i1.12679
Vol. 14, Nº 1, pp. 94–105
Homepage: seer.upf.br/index.php/rbca/index

ISSN 2176-6649

TUTORIAL

Desenvolvimento, Simulação e Validação de Protocolos MAC para Redes de Sensores Sem Fios

Development, Simulation and Validation of MAC protocols for Wireless Sensor Networks

Rafael Cotrim[1], João M. L. P. Caldeira[1, 2, 3], Vasco N. G. J. Soares[1, 2, 3]

[1]Instituto Politécnico de Castelo Branco, [2]Instituto de Telecomunicações, [3]InspiringSci, Castelo Branco, Portugal

rafael.cotrim@ipcbcampus.pt; jcaldeira@ipcb.pt; *vasco.g.soares@ipcb.pt;

Recebido: 17/06/2021. **Revisado:** 26/10/2021. **Aceito:** 10/03/2022.

Resumo

O estudo e implementação de redes de sensores sem fios é um campo emergente da eletrônica que está interligado com a internet das coisas. Juntas essas duas tecnologias possibilitam a recolha e transmissão de dados em cenários ondes redes comuns não são adequadas. Entretanto, devido às limitações usuais que são impostas aos equipamentos usados, a conservação de energia se torna indispensável para a sua utilização. Por conta disso, foram criados diversos mecanismos para redução no consumo de energia, uma grande parte dos quais passa pela implementação de novos protocolos MAC. Este artigo explica como podem ser feitas implementações desse tipo de protocolo em um simulador, o que permite que se avalie o seu desempenho sem os custos associados com estudos no mundo real. Para tanto, foram comparados os simuladores mais comuns usados neste campo de estudo e o OMNeT++ foi escolhido como o mais adequado. Depois disso, foi feita uma demonstração prática de como implementar um protocolo MAC nesse simulador e os resultados da simulação foram analisados.

Palavras-Chave: Protocolos MAC; OMNeT++; RSSF; Simulação; Tutorial

Abstract

The study and implementation of wireless sensor networks is an emerging field of technology that is tightly coupled with the internet of things. Together these two technologies allow the collection and transfer of data in scenarios where conventional networks would not be adequate. However, due to the usual limitations imposed on the equipment used, the conservation of energy becomes indispensable when utilizing them. For this reason, there have been many mechanisms created to reduce energy consumption, a large fraction of which works by creating novel MAC protocols. This article explains how to implement this kind of protocol in a simulator, which allows them to be evaluated without the costs associated with the creation of a testbed. Many of the most common simulators used in the area were compared and OMNeT++ was chosen as the most adequate. Then a practical demonstration of how to implement a MAC protocol was done and the results of the demonstration analyzed.

Keywords: MAC Protocols; OMNeT++; WSNs; Simulation; Tutorial

1 Introdução

Redes de sensores sem fios (RSSFs) (Singh et al., 2018), também conhecidas como WSNs em inglês, são redes formadas por sensores espalhados em uma área ou presos a objetos de interesse com o objetivo de se fazer medições. Esses equipamentos se comunicam por meio de transmissão de rádio entre si, encaminhando os dados acumulados para um *gateway*, um nó que acumula os dados e os envia para o destino fora da RSSF. Os nós que constituem essas redes são feitos para serem baratos, compactos e compatíveis às condições nas quais eles serão instalados (Polavarapu and Panda, 2020). Esse tipo de rede tem diversas aplicações, incluindo monitoramento de pacientes em hospitais (Caldeira et al., 2013), gestão de redes elétricas (Lopez et al., 2019), medição dos níveis de poluição em rios (Kadir et al., 2019) entre outras (Begum and Dixit, 2016).

Apesar de serem muito úteis, existem diversos problemas que devem ser resolvidos para que uma RSSF funcione de forma efetiva (Ukhurebor et al., 2021), o maior dos quais é a limitada quantidade de energia disponível para cada nó. Por conta dos espaços onde nós são distribuídos, a rede raramente tem uma fonte estável de energia, forçando o uso de baterias. Por conta disso, a redução no consumo energético dos nós se torna uma prioridade em aplicações que exigem longos períodos de operação (Anastasi et al., 2009).

As operações dos rádios nessas redes representam algumas das maiores despesas em termos de energia, sendo ordens de magnitude mais caras do que a aquisição de dados ou processamento (Pottie and Kaiser, 2000). Esses fatores levaram a que uma grande parte da pesquisa nesta área fosse voltada à minimização do número e duração de transições feitas com um foco nas operações dos protocolos da camada MAC (Rafael Cotrim, Soares and Azzoug, 2021), os quais controlam o acesso ao meio e têm grande impacto nas operações do rádio.

Diversos protocolos MAC especializados para esse problema já foram desenvolvidos e métodos variados foram usados para avaliar a efetividade de cada um (Rafael Cotrim, Soares and Gaspar, 2021). Análises matemáticas podem ser usadas para estimar o consumo e outras propriedades de protocolos, mas esse tipo de avaliação usufrui de primícias idealizadas que raramente caracterizam as condições do mundo real. Testes com equipamentos físicos caem no lado oposto do espectro. Seus resultados representam o comportamento real do sistema, mas criar um ambiente de teste robusto para redes com centenas de sensores demanda muito tempo e dinheiro.

Em virtude disso, simulações são um ótimo passo intermediário para esse tipo de validação. Elas permitem que se faça a avaliação dos algoritmos desenvolvidos de forma mais fidedigna do que a análise matemática e mais barata do que a implementação em bancada de testes (Bakni et al., 2020). Elas ainda requerem validação para garantir que seus resultados são congruentes com o mundo real, mas simulações aceleram substancialmente o processo de prototipagem, permitindo que só se proceda quando há confiança nos algoritmos já desenvolvidos.

O objetivo deste artigo é proporcionar ao leitor uma perspectiva de como fazer a avaliação de protocolos MAC para RSSFs em simulações. Mais especificamente, mostrar o processo de se simular os protocolos T-MAC (Van Dam and Langendoen, 2003), B-MAC (Polastre et al., 2004) e RI-MAC (Sun et al., 2008) usando o simulador OMNeT++. A Seção 2 explora em mais detalhes o funcionamento dos protocolos citados. A Seção 3 apresenta alguns dos simuladores disponíveis, suas propriedades e as razões por trás da escolha do OMNeT++ (OpenSim Ltd, 2021b). A Seção 4 contém um exemplo de como se implementar o RI-MAC no OMNeT++. Na Seção 5, há uma avaliação dos resultados da simulação e, finalmente, na Seção 6, é apresentada a conclusão do artigo.

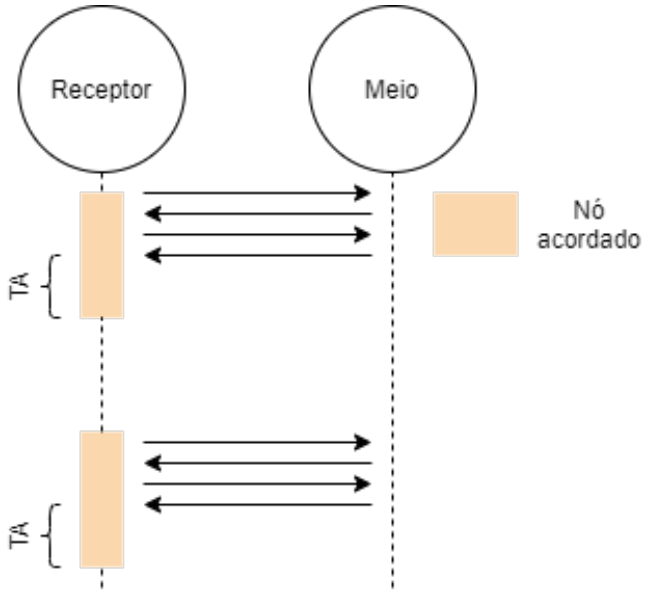

Figura 1: Exemplo das operações do T-MAC.

2 Protocolos MAC

Protocolos MAC podem afetar o consumo energético de nó de várias formas (Alfayez et al., 2015). Algoritmos feitos especificamente para RSSF minimizam o tempo que o rádio está ativo ao entrar num estado de sono quando ele não é necessário (Radha et al., 2019). Enquanto um nó dorme, ele é incapaz de enviar ou receber dados, mas seu consumo energético é substancialmente reduzido. Eles também reduzem o cumprimento e a frequência das transferências, permitindo que o nó durma durante mais tempo.

Existem diversas formas de se atingir reduzir o tempo que o rádio de um nó permanece ativo e cada protocolo usa sua própria estratégia. O Timeout MAC (T-MAC) (Van Dam and Langendoen, 2003) faz com que todos os nós acordem em intervalos regulares, o que permite que eles se comuniquem durante um curto período. Cada equipamento também otimiza suas próprias operações ao voltar a dormir caso não receba quaisquer mensagens em um certo período. Na Fig. 1, a qual exemplifica as operações do T-MAC, este período está denominado como TA. Enquanto efetivo, o T-MAC requer que seja implementado um mecanismo de sincronização entre os nós para garantir que não haja desfasamento entre seus períodos de atividade.

Em comparação, O Berkeley MAC (B-MAC) (Polastre

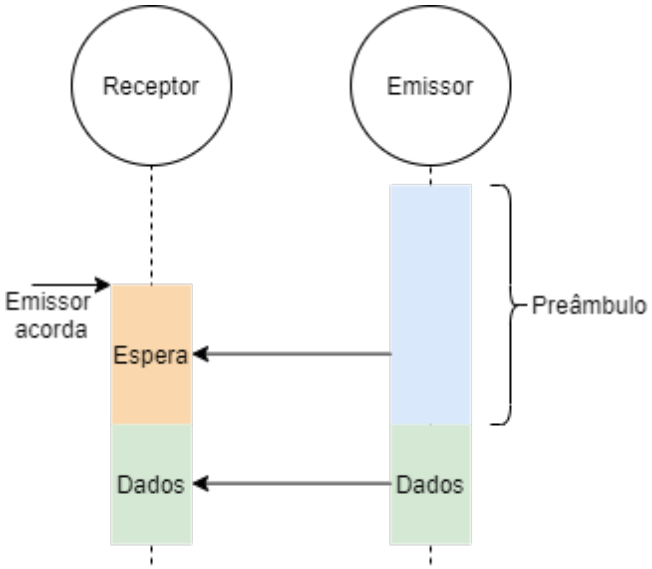

Figura 2: Exemplo das operações do B-MAC.

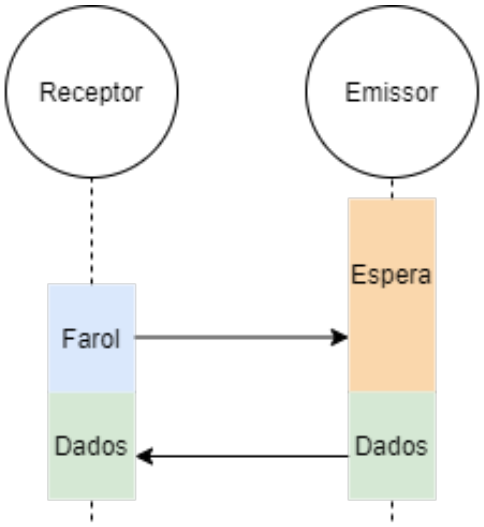

Figura 3: Exemplo das operações do RI-MAC.

et al., 2004) não requer que toda a rede acorde no mesmo momento e logo também não requer o mecanismo de sincronização. Em redes que usam esse algoritmo, nós com dados a serem enviados acordam e começam a enviar um preâmbulo que é ligeiramente mais longo que o período que os nós dormem. Quando outros nós acordam, eles verificam se há transmissões no meio e, se detectarem um preâmbulo, permanecem acordados até que ele termine. Uma vez que o preâmbulo termine, o emissor envia os dados e os receptores comparam o endereço MAC do pacote com o seu próprio para determinar se são o destinatário.

Finalmente, o Receiver Initiated MAC (RI-MAC) (Sun et al., 2008) inverte as operações normais do protocolo. Nós com dados a enviar permanecem acordados e ouvindo ao meio de comunicações. Quando um novo dispositivo acorda, ele envia um pacote especial denominado de *beacon* para todos os seus vizinhos. Ao receber esse pacote, emissores verificam se o nó que acabou de acordar é o destinatário dos dados que eles possuem. Se for, eles enviam os dados e voltam a dormir, caso contrário eles permanecem em estado de escuta a todo o tráfego da rede.

3 Ferramentas de simulação

Durante o processo de desenvolvimento e testes de um novo protocolo, é benéfico usufruir de diversas formas de avaliação. Explorações teóricas delimitam o comportamento geral do algoritmo, permitindo que se identifique possíveis falhas nas implementações das fases seguintes. Experimentos no mundo real garantem a efetividade dos protocolos desenvolvidos, mas requerem um bom planejamento e hardware suficiente para serem feitas. Por fim, simulações permitem que se possa estudar o comportamento do sistema em condições mais próximas da realidade com pouco investimento de recursos, o que permite a contínua prototipagem e garante a reprodutibilidade dos resultados. A Tabela 1 contém uma lista dos principais simuladores usados na avaliação de protocolos MAC para RSSFs.

O OMNeT++ (OpenSim Ltd, 2021b), acrônimo para Objective Modular Network Testbed in C++ (Stroustrup, 2000), é um simulador de eventos discretos especializado na avaliação de redes. O projeto é uma biblioteca e *framework* de simulação de código aberto e gratuito para usos acadêmicos e de exploração científica. Também existe uma versão expandida para uso comercial chamada de OMNEST (Cogitative Software FZE, 2021). Ambas as versões são construídas ao redor da ideia de módulos. A rede é o módulo no topo da hierarquia e todos os seus componentes, incluindo equipamentos sendo simulados, são submódulos. Esses submódulos também podem ter seus próprios submódulos que definem uma diversidade de comportamentos. Módulos comunicam entre si usando mensagem que são geridas pelo próprio *framework* para evitar que haja dependências explícitas entre componentes. A estrutura resultante permite que módulos sejam substituídos facilmente.

O simulador Castalia (Boulis and Pediaditakis, 2016) é baseado no OMNeT++ e se foca especificamente na simulação de de RSSFs. Tal como o OMNeT++, este *framework* é muito bem parametrizado e já incluem diversos protocolos comuns na área. Além disso, o Castalia não precisa de bibliotecas externas para se adequar a comunicações sem fio, pois boa parte das funcionalidades mais usuais já estão incluídas.

Com financiamento proveniente da Fundação Nacional da Ciência nos Estados Unidos, o NS-3 (NSNAM, 2021a) é o sucessor ao popular simulador de redes NS-2 (NSNAM, 2021b). Enquanto o NS-2 ainda é usado em diversos contextos, o seu desenvolvimento foi parado em 2011 e ele não é compatível com o NS-3. Tal como o OMNeT++, o NS-3 é um simulador de eventos discretos programado em C++, mas também permite o uso da linguagem Python (Van Rossum and Drake, 2009). O suporte a essa linguagem ainda é limitado, mas é algo que vem sendo melhorado nas últimas versões. O projeto NS-3 é voltado ao uso na educação e pesquisa, mas sua licença permite uso comercial desde que quaisquer alterações feitas ao simulador sejam dispo-

Tabela 1: Lista de simuladores para RSSFs.

Simulator	Plataformas	Licença	Permite uso Comercial	Linguagens	Versão mais recente
Castalia	Windows, Linux, Mac OS e Docker	Academic Public License ou Castalia License	Não	C++	3.3 (2013)
NetSim	Windows	Licença Comercial	Sim	C	13 (2021)
NS-2	Windows, Linux e Mac OS	GPL	Sim	C++	2.35 (2011)
NS-3	Linux e Mac OS	GPLv2	Sim	C++ e Python	3.33 (2021)
OMNeT++	Windows, Linux, Mac OS e Docker	Academic Public License	Não	C++	5.6.2 (2020)
OMNEST	Windows, Linux, Mac OS e Docker	Licença Comercial	Sim	C++	5.6.2 (2020)

nibilizadas.

O NetSim (Tetcos, 2021) é um simulador de redes proprietário criado pela empresa Tetcos e pode ser licenciado para uso pessoal, comercial, ou em instituições de ensino. Suas maiores vantagens são o seu editor visual, suporte provido pela Tetcos, o sistema de integração com hardware real, e a integração com diversos outros programas tais como o Matlab (MATLAB, 2010), Eclipse SUMO (Lopez et al., 2018) e Wireshark (Sanders, 2017). Devido a sua flexibilidade, este simulador é usado tanto para aplicações militares quanto para modelação de sistemas de transporte inteligentes e outras aplicações civis.

Dentre os simuladores citados, o OMNeT++ foi escolhido por sua versatilidade e robustez. Ele torna simples a criação de novos modelos e permite a criação de cenários com diversos níveis de concordância com o mundo real. Ele já foi usado na avaliação de diversas novas tecnologias na área de redes (Ibrahim and Bayat, 2020, Nabi et al., 2010, Van Dam and Langendoen, 2003) e é constantemente atualizado. Além disso, ele é gratuito e pode ser usado em uma variedade de ambientes, incluindo Linux (Sobell, 2015), Windows (Bott and Stinson, 2019), macOS (Apple Inc., n.d.) e contêineres Docker (Merkel, 2014). Também foi utilizado *framework* INET (OpenSim Ltd, 2021a), o qual inclui modelos para comunicações sem fio, roteamento e assuntos correlatos.

```
1  module RIMac extends MacProtocolBase like IMacProtocol
2  {
3      parameters:
4          // MAC address
5          string address = default("auto");
6
7          // How long the node sleeps
8          double sleepInterval @unit(s) = default(1s);
9
10         // Time the node spends doing
11         // CCA (clear channel assertion)
12         double ccaInterval @unit(s) = default(0.01s);
13
14         // Maximum time a node waits
15         // for an ACK (acknowledgement)
16         double dwellInterval @unit(s) = default(0.03s);
17
18         // Size of the RI-MAC Header
19         int headerLength @unit(b) = default(10b);
20
21         double bitrate @unit(bps) = default(19200 bps);
22         int mtu @unit(B) = default(0B);
23         string radioModule = default("^.radio");
24
25         @class(RIMac);
26         @statistic[packetDropIncorrectlyReceived](
27             title="packet drop: incorrectly received";
28             source=packetDropReasonIsIncorrectlyReceived(
29                 packetDropped
30             );
31         @selfMessageKinds(inet::RIMacType);
32     submodules:
33         queue: <default("DropTailQueue")> like IPacketQueue {
34             parameters:
35                 packetCapacity = default(20);
36                 @display("p=100,100;q=l2queue");
37         }
38 }
```

Figura 4: Arquivo NED para o RI-MAC.

4 Implementação do RI-MAC

4.1 Classes base

Para avaliar um novo protocolo MAC no OMNeT++, é necessário criar um módulo que o implemente. O INET oferece a classe `MacProtocolBase` e a interface `IMacProtocol` como base para esse desenvolvimento. Primeiramente, será criado o arquivo NED visível na Fig. 4, o qual serve para descrever as propriedades de cada módulo e permitir a edição de parâmetros posteriormente. Neste arquivo devem ser colocados os parâmetros de operação do protocolo e algumas informações que vão auxiliar as as operações internas da biblioteca INET. É conveniente definir valores padrões para as propriedades, mas não é estritamente necessário. A configuração vista na figura inicializa as variáveis do protocolo com valores padrões, permite o monitoramentos de algumas estatísticas e limita o tamanho da fila de pacotes a 20.

Todos os módulos do OMNeT++ são acompanhados por classes C++ em que o seu comportamento é implementado. Por padrão, o *framework* associa módulos com a classes que tenham o mesmo nome, mas isso pode ser alterado com o decorador `@class` presente no arquivo NED do módulo. Será criada então a classe `RIMac` e ela herdará da classe `MacProtocolBase` e da interface `IMacProtocol`. Uma secção do cabeçalho da classe contendo os métodos mais importantes e suas funções podem ser vista na Fig. 5.

```
class INET_API RIMac : public MacProtocolBase,
    public IMacProtocol{
private:
    // Copy constructor is not allowed.
    RIMac(const RIMac&);
    // Assignment operator is not allowed.
    RIMac& operator=(const RIMac&);

  public:
    RIMac() {}
    virtual ~RIMac();

    // Number of initialization stages
    virtual int numInitStages() const override {
        return NUM_INIT_STAGES;
    }

    // Initialization of the module and some variables
    virtual void initialize(int) override;
    // Update changes to the display
    virtual void refreshDisplay() const override;
    // Delete all dynamically allocated objects of the module
    virtual void finish() override;
    // Handle messages from lower layer (radio)
    virtual void handleLowerPacket(Packet *packet) override;
    // Handle messages from upper layer (app)
    virtual void handleUpperPacket(Packet *packet) override;
    // Handle self messages (timers)
    virtual void handleSelfMessage(cMessage *) override;
    // Handle control messages from lower layer
    virtual void receiveSignal(cComponent *source,
        simsignal_t signalID,
        intval_t value,
        cObject *details) override;
    // Apply configuration to the network interface
    virtual void configureInterfaceEntry() override;

    /*...*/
};
```

Figura 5: Principais métodos de um protocolo MAC usando a biblioteca INET.

Em termos gerais, a seguinte sequência de eventos acontece quando um protocolo MAC é iniciado: O método `initialize` inicializa o protocolo com valores retirados de um arquivo de parâmetros e o método `configureInterfaceEntry` aplica as informações do protocolo na interface de rede. Depois disso um dos métodos `handleLowerPacket`, `handleUpperPacket` ou `handleSelfMessage` é executado quando uma mensagem é recebida a depender de sua origem. É nestas três funções que a maior parte do comportamento de um protocolo é definida. O método `receiveSignal` é chamado quando ocorrem eventos na camada inferior tais como o desligamento do rádio. Este método é particularmente útil para detectar quando a transmissão que um nó está efetuando foi concluída. Finalmente, a função `finish` é responsável por desalocar os objetos dinâmicos em memória.

Uma das formas mais convenientes de seguir esse fluxo de controle ao implementar um protocolo deste gênero é usando uma máquina de estados. Desta forma, o algoritmo pode responder apropriadamente às mensagens que são recebidas e o estado do rádio pode ser mais facilmente controlado. Também é necessário definir todos os tipos de eventos internos que podem ocorrer, os quais são em sua maior parte temporizadores. Temporizadores são representados como mensagens enviadas para si mesmo no OMNeT++ e o acréscimo de um enumerador para indicar seu tipo é opcional, mas pode facilitar a implementação da máquina de estados e torna o processo de depuração mais simples. A Fig. 6 lista todos os estados e eventos que serão

```
enum RIMacState {
    // RADIO OFF/Sleeping
    INIT,                 // Initialize protocol
    SLEEP,                // Sleep

    // Radio receiving
    BEACON_CCA,           // Wait channel to be free to send beacon
    DATA_CCA,             // Wait channel to be free to send data

    // Radio transmitting
    WAIT_BEACON_OVER,     // Wait beacon transmission to be over
    WAIT_DATA_OVER,       // Wait data transmission to be over
    WAIT_ACK_OVER,        // Wait ACK transmission to be over
    WAIT_RESPONSE_OVER,   // Wait beacon-on-request transmission
                          // to be over

    // Radio receiving
    WAIT_DATA,            // Wait for data to arrive
    WAIT_ACK,             // Wait ACK beacon to arrive
    WAIT_BEACON           // Wait normal beacon to arrive
};

enum RIMacEvent {
    WAKEUP,               // Time to wake up
    CCA_TIMEOUT,          // CCA period ended
    ACK_TIMEOUT,          // ACK not received in time
    DWELL_TIMEOUT,        // The node has received a beacon,
                          // but the data has not arrived
                          // within the expected timeframe
    BEACON_TIMEOUT,       // Beacon not received in time
    TRANSMISSION_ENDED,   // Current transmission ended
    BEACON_RECEIVED,      // Beacon was received
    DATA_RECEIVED         // Data was received
};
```

Figura 6: Estados e eventos do RI-MAC.

usados e descreve seu significado geral.

Além disso, é necessário definir os cabeçalho que o protocolo usará. Isso é feito em um arquivo `MSG`, o qual serve para descrever mensagens customizadas no OMNeT++. Este sistema existe para evitar que programadores tenham de escrever o código repetitivo que normalmente é associado a esse tipo de dado. Quando o código for compilado, o *framework* irá transformar as informações presentes nesse arquivo em código C++ e dará continuidade ao processo de compilação. Serão gerados dois arquivos `{nome original}_m.h` e `{nome original}_m.cc`, que não devem ser alterados manualmente, uma vez que todas as alterações serão perdidas quando o código for compilado novamente.

Na Fig. 7, são definidos os tipos de informação que podem estar sendo enviados, sinais de *beacon* ou dados, e três tipos de cabeçalho. `RIMacHeaderBase` é apenas usado como base para os outros dois, enquanto `RIMacBeaconFrame` será usado em mensagens *beacon* e `RIMacDataFrame` será usado em mensagens contendo dados. A variável `networkProtocol` serve para identificar qual o protocolo da camada superior está sendo usado, esta informação é preenchida quando o encapsulamento da mensagem é feito.

4.2 Comportamento do protocolo

Com as classes base definidas, torna-se possível começar a implementação do comportamento do protocolo. Primeiramente, os dados presentes no arquivo NED devem ser extraídos pelo código durante o período de inicialização. Isso é feito no método `initialize`. Esse método é chamado diversas vezes para iniciar partes diferentes do protocolo e sempre é passado um enumerador, o qual está identificado

```
enum RIMacType {
    RIMAC_DATA = 1;
    RIMAC_BEACON = 2;
};

class RIMacHeaderBase extends FieldsChunk {
    MacAddress srcAddr;
    MacAddress destAddr;
    RIMacType  type;
}

class RIMacBeaconFrame extends RIMacHeaderBase { }

class RIMacDataFrame extends RIMacHeaderBase  {
    int networkProtocol = -1;
}
```

Figura 7: Definição do cabeçalhos do RI-MAC.

```
void RIMac::initialize(int stage) {
    MacProtocolBase::initialize(stage);
    if (stage == INITSTAGE_LOCAL) {

        bitrate = par("bitrate");
        ctrlFrameLength = b(par("headerLength"));
        sleepInterval = par("sleepInterval");
        ccaInterval = par("ccaInterval");
        dwellInterval = par("dwellInterval");

        stateDescr[INIT] = "INIT";
        stateDescr[SLEEP] = "SLEEP";
        /*Describe other states*/

        eventDescr[WAKEUP] = "WAKEUP";
        eventDescr[CCA_TIMEOUT] = "CCA_TIMEOUT";
        /*Describe other events*/

        // Initialize queue
        txQueue = check_and_cast<queueing::IPacketQueue*>(
            getSubmodule("queue")
        );

        macState = SLEEP;    // Current state of the protocol

        attempts = 0;        // Number of times the node has
                             // attempted to send a package

        maxAttempts = 5;     // Maximum number of attempts

    } else if (stage == INITSTAGE_LINK_LAYER) {
        /*...*/
    }
}
```

Figura 8: Inicialização das variáveis locais.

como `stage`, para indicar o que deve ser feito. Como mostra a Fig. 8, durante a fase de inicialização local as variáveis do protocolo são recuperadas através da função `par`. Além disso, são criados dicionários contendo descrições de cada um dos estados e eventos. Isso será usado para gerar mensagens de *log* descritivas, as quais facilitam o processo de depuração.

A próxima fase da inicialização, representada pela Fig. 9, foca-se no rádio e na criação dos eventos internos do protocolo. Esse módulo irá se inscrever para receber atualizações sobre o estado do rádio. Isso permitirá a detecção de quando uma transmissão foi concluída ou quando algo está sendo recebido para que se possa reagir de forma apropriada. Além disso, a macro `WATCH` é usada para permitir que se possa visualizar o estado do protocolo no ambiente gráfico durante uma simulação. Finalmente, a mensagem `wakeup` é agendada, fazendo com que o nó acorde depois de um período igual ao `sleepInterval` ter passado. O método `scheduleEvent` foi criado para simplificar a operação de agendamento e seu conteúdo pode ser visto na Fig. 10

```
void RIMac::initialize(int stage) {
    MacProtocolBase::initialize(stage);
    if (stage == INITSTAGE_LOCAL) {
        /*...*/
    } else if (stage == INITSTAGE_LINK_LAYER) {

        // Subscribe to radio events
        cModule *radioModule = getModuleFromPar<cModule>(
            par("radioModule"),this);

        radioModule->subscribe(
            IRadio::receptionStateChangedSignal, this);
        radioModule->subscribe(
            IRadio::transmissionStateChangedSignal, this);
        radio = check_and_cast<IRadio*>(radioModule);

        // Allow the MAC state to be observed in the graphical
        // environment during simulations
        WATCH(macState);

        // Create self-messages
        wakeup = new cMessage("Wakeup", WAKEUP);
        ccaTimeout = new cMessage("CCA Timeout", CCA_TIMEOUT);
        dwellTimeout = new cMessage("Dwell Time", DWELL_TIMEOUT);
        ackTimeout = new cMessage("Ack timeout", ACK_TIMEOUT);
        beaconTimeout = new cMessage(
            "Beacon Timeout",
            BEACON_TIMEOUT
        );

        scheduleEvent(sleepInterval, wakeup);
    }
}
```

Figura 9: Inicialização secundária.

```
void RIMac::scheduleEvent(double interval, cMessage *event) {
    scheduleAt(simTime() + interval, event);
}
```

Figura 10: Método de agendamento de eventos.

Para responder ao sinal de acordar e aos outros temporizadores, basta implementar o método `handleSelfMessage`. A Fig. 11 exemplifica o procedimento necessário. O sinal `WAKEUP` será um dos eventos mais comuns e deve ser repetido de forma periódica independentemente das outras operações do protocolo, então ele é reenviado sempre que é recebido. Seu retorno será agendado para entre 50% e 150% do período de sono médio a fim de evitar que os nós acidentalmente se sincronizem. Depois disso, o evento pode ser tratado dependendo do estado do protocolo. Um procedimento similar ao exemplificado na Fig. 11 acontece para todos os estados e as mensagens relevantes para cada um deles.

Uma vez que cada estado do protocolo tem apenas um estado do rádio associado, é conveniente criar uma função que automaticamente mude a configuração do rádio quando atualizarmos o estado do protocolo. O método `setState` presente na Fig. 12 tem esse objetivo. Toda vez que é usado para mudar o estado do protocolo, ele também envia uma mensagem de depuração contento informações sobre a transição e atualizará o modo de operação do rádio.

A recepção dos dados a serem enviados é feita na função `handleUpperPacket`, a qual é chamada sempre que as camadas superiores entregam um pacote ao protocolo. Quando isso ocorre, é necessário instanciar um novo cabeçalho,

```
1  void RIMac::handleSelfMessage(cMessage *msg) {
2      auto const event = static_cast<RIMacEvent>(msg->getKind());
3      if (event == WAKEUP) {
4          scheduleEvent(sleepInterval * (0.5 + dblrand()), wakeup);
5      }
6
7      switch (macState) {
8      case SLEEP:
9          if (event == WAKEUP) {
10             setState(BEACON_CCA, event);
11             scheduleEvent(ccaInterval * (1 + dblrand()),
12                 ccaTimeout);
13         }
14         break;
15     case BEACON_CCA:
16         if (event == CCA_TIMEOUT) {
17             if (!isChannelClear()) {
18                 cancelEvent(ccaTimeout);
19                 scheduleEvent(ccaInterval * (1 + dblrand()),
20                     ccaTimeout);
21                 return;
22             }
23
24             sendBeacon();
25             setState(WAIT_BEACON_OVER, event);
26         }
27         break;
28     /* ... */
29     }
30 }
```

Figura 11: Exemplo de recepção de temporizadores e outros sinais.

```
1  void RIMac::setState(RIMacState newState, RIMacEvent event) {
2      EV_DETAIL << "State " << stateDescr[macState] << ", message "
3          << eventDescr[event] << ", new state "
4          << stateDescr[newState] << endl;
5
6      switch (newState) {
7          case SLEEP:
8              radio->setRadioMode(IRadio::RADIO_MODE_SLEEP);
9              break;
10
11         case WAIT_BEACON_OVER:
12         case WAIT_DATA_OVER:
13         case WAIT_ACK_OVER:
14         case WAIT_RESPONSE_OVER:
15             radio->setRadioMode(IRadio::RADIO_MODE_TRANSMITTER);
16             break;
17
18         case INIT:
19         case BEACON_CCA:
20         case WAIT_DATA:
21         case WAIT_BEACON:
22         case WAIT_ACK:
23             radio->setRadioMode(IRadio::RADIO_MODE_RECEIVER);
24             break;
25     }
26     macState = newState;
27 }
```

Figura 12: Método de gestão de estados.

preenchê-lo com as informações relevantes e colocá-lo na parte frontal do pacote. Esse procedimento pode ser visto na Fig. 13. Depois dos dados da camada MAC serem preenchidos, o pacote é colocado na fila. Se a fila já estiver cheia, o pacote será descartado.

Quando chegar o momento de enviar um pacote de dados, o protocolo primeiramente varifica se há pacotes na fila usando a função `txQueue->isEmpty()`. Se houver, o protocolo irá entrar no estado `WAIT_DATA_OVER` e executará a função `popTxQueue`, a qual é herdada da classe `MacProtocolBase`. Depois disso, é necessário acrescentar o tempo de transmissão do pacote e enviá-lo para o rádio como é mostrado na Fig. 14. O rádio começará a transmitir dados assim que a sua transição de estados estiver concluída. O envio de outros tipos de pacotes, mensagens de *beacon* por exemplo, segue o mesmo molde. Em contrapartida, se não houver pacotes na fila, o protocolo RI-MAC define que o nó deve voltar a dormir até que o próximo sinal de acordar seja recebido.

No lado do receptor, a função `handleLowerPacket` coleta os dados recebidos, entregando o pacote ao protocolo. Primeiramente, é necessário verificar se houve erros ou colisões na transmissão. A biblioteca INET oferece diversos modelos de como detectar e tratar essas ocorrências. Em algumas aplicações, pode ser possível detectar quais os bits que foram afetados e usar uma metodologia de correção de erros para restaurar os pacotes. Para efeitos deste documento, não serão usadas essas funcionalidades. Se um erro for detectado, o pacote será descartado. Pacotes que sofreram interferência são marcados como errôneos em um processo que será explicado posteriormente, a existência dessa marcação pode ser verificada com o método `packet->hasBitError()`. Quando um pacote é descartado devido a de erros, também se deve executar a função emitir um sinal indicando esse acontecimento para que o simulador possa acumular estatísticas sobre essa ocorrência.

```
1  void RIMac::handleUpperPacket(Packet *packet) {
2      auto pkt = makeShared<RIMacDataFrame>();
3      pkt->setChunkLength(ctrlFrameLength);
4      pkt->setType(RIMAC_DATA);
5      auto dest = packet->getTag<MacAddressReq>()
6          ->getDestAddress();
7
8      delete packet->removeControlInfo();
9
10     pkt->setNetworkProtocol(
11         ProtocolGroup::ethertype.getProtocolNumber(
12             packet->getTag<PacketProtocolTag>()->getProtocol()
13         )
14     );
15     pkt->setDestAddr(dest);
16     pkt->setSrcAddr(interfaceEntry->getMacAddress());
17
18     //encapsulate the network packet
19     packet->insertAtFront(pkt);
20
21     EV_DETAIL << "pkt encapsulated\n";
22     packet->addTagIfAbsent<PacketProtocolTag>()
23         ->setProtocol(&Protocol::rimac);
24     txQueue->pushPacket(packet);
25     EV_DETAIL << "Max queue length: "
26     << txQueue->getMaxNumPackets()
27     << ", packet put in queue\n" << "  queue size: "
28     << txQueue->getNumPackets() << " macState: "
29     << macState << endl;
30 }
```

Figura 13: Método recepção de mensagens da camada anterior.

```
1 Packet *pkt = currentTxFrame->dup();
2 simtime_t duration = pkt->getBitLength() / bitrate;
3 pkt->setDuration(durationOf(pkt));
4 sendDown(pkt);
```

Figura 14: Envio do pacote atual.

```
if (packet->hasBitError()) {
        EV << "Received " << packet
        << " contains bit errors or collision, dropping it\n";
    PacketDropDetails details;
    details.setReason(INCORRECTLY_RECEIVED);
    emit(packetDroppedSignal, packet, &details);
    delete packet;
    return;
}
```

Figura 15: Emissão de sinal de que um pacote foi descartado.

```
const auto &hdr = packet->popAtFront<RIMacDataFrame>();

packet->addTagIfAbsent<MacAddressInd>()
    ->setSrcAddress(hdr->getSrcAddr());
packet->addTagIfAbsent<InterfaceInd>()
    ->setInterfaceId(interfaceEntry->getInterfaceId());
auto payloadProtocol = ProtocolGroup::ethertype
    .getProtocol(hdr->getNetworkProtocol());
packet->addTagIfAbsent<DispatchProtocolReq>()
    ->setProtocol(payloadProtocol);
packet->addTagIfAbsent<PacketProtocolTag>()
    ->setProtocol(payloadProtocol);

EV_DETAIL << " message decapsulated " << endl;

sendUp(packet);
```

Figura 16: Desencapsulamento de um pacote.

A Fig. 15 ilustra como isso pode ser feito. Vale ressaltar que a eliminação de pacotes recebidos deve ser feita com a palavra-chave `delete` independentemente de eles estarem sendo removidos por conta de erros ou pelo funcionamento normal do protocolo. Isso evita vazamentos de memória, os quais são particularmente graves quando um estudo de parâmetros é executado.

Após essa checagem, o pacote pode ser avaliado de acordo com o estado do protocolo. Se o pacote contiver dados, é necessário fazer o desencapsulamento como o exemplificado em Fig. 16. Depois disso, o pacote pode ser enviado para as camadas superiores com o método `sendUp`.

Um emissor precisa detectar quando o rádio terminou de fazer a transmissão para que ele possa mudar para o estado apropriado. A detecção desse evento e diversos outros é feita no método `receiveSignal`. Essa função recebe os sinais para os quais o protocolo se inscreveu durante a fase de inicialização. O código na Fig. 17 percebe essas mudanças e executa certos métodos a depender do tipo de sinal. Por sua vez, esses métodos verificam o estado do protocolo e tomam as devidas ações.

4.3 Serializadores, dissecador e impressores

A última coisa a se fazer para implementar um protocolo MAC no OMNeT++ é criar as classes auxiliares que permitirem ao INET simular redes sem fio. Um serializador é uma classe que converte a representação em classe de um pacote na sequência de *bytes* que será transmitida. Essa classe também é responsável for desserializar o objeto quando este for recebido no nó de destino. No INET, serializadores precisam herdar da classe `FieldsChunkSerializer`, implementar os métodos apropriados e devem ser registrados

```
void RIMac::receiveSignal(cComponent *source,
    simsignal_t signalID, intval_t value, cObject *details) {

    Enter_Method_Silent();

    if (signalID == IRadio::receptionStartedSignal) {
        if (value == IRadio::RECEPTION_STATE_BUSY ||
        value == IRadio::RECEPTION_STATE_RECEIVING) {
            channelBusy();
        } else if (value == IRadio::RECEPTION_STATE_IDLE) {
            channelClear();
        }
    } else if (signalID == IRadio::transmissionStateChangedSignal
        && value == IRadio::TRANSMISSION_STATE_IDLE) {
        transmissionEnded();
        EV_DETAIL << "Transmission ended" << endl;
    }
}
```

Figura 17: Recepção de sinais do rádio.

como os serializadores para os tipos específicos de mensagem que estão sendo usados. Esse processo pode ser visto na Fig. 18. Vale ressaltar que a ordem de escrita e leitura dos campos tem de ser mantida a mesma em ambos os métodos. No exemplo dado, o serializador detecta que houve um erro na transmissão quando o tipo da mensagem recebida não existe. Quando isso acontece, o serializador gera um cabeçalho base, preenche-o com algumas informações e marca-o como contendo interferências.

Dissecadores permitem que se analise o que há dentro de um pacote independentemente de como ele está sendo representado, mesmo que esteja na forma de `BytesChunk` ou outros formatos binários. Impressores servem apenas para formatar o conteúdo de um pacote para algo legível, o que facilita a depuração.

Tal como os serializadores, tando dissecadores quanto impressores têm classes base apropriadas, `ProtocolDissector` e `ProtocolPrinter`, e precisam ser registrados. As Fig. 19 e Fig. 20 demonstram a implementação essas funcionalidades de maneira básica.

5 Avaliação de desempenho

5.1 Definição do cenário de estudo

Existem diversos cenários que podem ser usados para estudar um protocolo MAC. Esta seção provê um exemplo focado em mensurar os efeitos do aumento da densidade de uma rede que se encontra em um ambiente confinado. Entretanto, é possível usar o OMNeT++ para avaliar muitos outros casos, incluindo cenários com nós em movimento, sistemas de regeneração de energia, técnicas de redução de dados e diversos outros.

O cenário exemplo mostrado na Fig. 21 foi criado para avaliar a performance de diversos protocolos MAC para uso dentro de ambientes confinados como caminhões de entrega. Nesse caso, os recipientes usados no transporte dessas frutas têm sensores integrados que permitem ler o estado dos produtos em todos os momentos e eles formam uma rede de sensores sem fio no interior do veículo. Como o nível de preenchimento do caminhão, o formato e tamanho dos caixotes podem variar, a densidade da rede também é variável. As simulações propostas se focam em medir os impactos dessa variação.

```
Register_Serializer(RIMacHeaderBase, RIMacHeaderSerializer);
Register_Serializer(RIMacBeaconFrame, RIMacHeaderSerializer);
Register_Serializer(RIMacDataFrame, RIMacHeaderSerializer);

void RIMacHeaderSerializer::serialize(MemoryOutputStream& stream,
    const Ptr<const Chunk>& chunk) const
{
    b startPos = stream.getLength();
    auto& header = staticPtrCast<const RIMacHeaderBase>(chunk);
    stream.writeByte(header->getType());
    /*Write all header properties */

    auto remainderBits = b(header->getChunkLength() -
        (stream.getLength() - startPos)).get();
    if (remainderBits < 0)
        throw cRuntimeError("Header size smaller than required");
    if (remainderBits >= 8)
        stream.writeByteRepeatedly('?', remainderBits >> 3);
    if (remainderBits & 7)
        stream.writeBitRepeatedly(0, remainderBits & 7);
}

const Ptr<Chunk> RIMacHeaderSerializer::deserialize(
        MemoryInputStream &stream) const {
    b startPos = stream.getPosition();
    RIMacType type = static_cast<RIMacType>(stream.readByte());
    b length = b(stream.readUint16Be());
    MacAddress srcAddr = stream.readMacAddress();
    MacAddress destAddr = stream.readMacAddress();

    switch (type) {
    case RIMAC_DATA: {

        auto dataFrame = makeShared<RIMacDataFrame>();
        dataFrame->setType(type);
        /*Read all header properties */

        return dataFrame;
    }
    case RIMAC_BEACON:
        /*...*/

    default:
        auto unknownFrame = makeShared<RIMacHeaderBase>();
        unknownFrame->setType(type);
        unknownFrame->setChunkLength(length);
        unknownFrame->markIncorrect();
        return unknownFrame;
    }
}
```

Figura 18: Implementação do serializador.

```
Register_Protocol_Dissector(&Protocol::rimac,
    RIMacProtocolDissector);

void RIMacProtocolDissector::dissect(Packet *packet,
    const Protocol *protocol, ICallback& callback) const
{
    auto header = packet->popAtFront<RIMacHeaderBase>();
    callback.startProtocolDataUnit(&Protocol::rimac);
    callback.visitChunk(header, &Protocol::rimac);
    if(header->getType() == RIMAC_DATA)
    {
        const auto& dataHeader = dynamicPtrCast
            <const RIMacDataFrame>(header);
        auto payloadProtocol = ProtocolGroup::ethertype
        .findProtocol(dataHeader->getNetworkProtocol());
        callback.dissectPacket(packet, payloadProtocol);
    }
    ASSERT(packet->getDataLength() == B(0));
    callback.endProtocolDataUnit(&Protocol::rimac);
}
```

Figura 19: Implementação do dissecador.

```
Register_Protocol_Printer(&Protocol::rimac,RIMacProtocolPrinter);

void RIMacProtocolPrinter::print(const Ptr<const Chunk>& chunk,
    const Protocol *protocol,
    const cMessagePrinter::Options *options,
    Context& context) const
{
    if (auto header = dynamicPtrCast<RIMacHeaderBase>(chunk)) {
        context.sourceColumn << header->getSrcAddr();
        context.destinationColumn << header->getDestAddr();
        context.infoColumn << "(Acking MAC) " << chunk;
    }
    else
        context.infoColumn << "(Acking MAC) " << chunk;
}
```

Figura 20: Implementação do impressor.

```
network MacWireless {
    parameters:
        @display("bgb=18.75,2.6;bgg=100,1,grey95");
        @figure[title](type=label; pos=0,-1; anchor=sw;
            color=darkblue);

        int sensorCount = default(4);

        @figure[rcvdPkText](type=indicatorText; pos=380,20;
            anchor=w; font=,18;
            textFormat="packets received: %g"; initialValue=0);
        @statistic[packetReceived](source=gateway.app[0]
            .packetReceived; record=figure(count);
            targetFigure=rcvdPkText);

        @display("bgb=18.75,2.2");
    submodules:
        visualizer: <default("IntegratedVisualizer")>
            like IIntegratedVisualizer if hasVisualizer() {
            @display("p=580,125");
        }
        configurator: Ipv4NetworkConfigurator {
            @display("p=580,200");
        }
        radioMedium: <default("UnitDiskRadioMedium")>
            like IRadioMedium {
            @display("p=580,275");
        }
        sensors[sensorCount]: SensorNode {
            @display("i=misc/sensor2");
        }
        gateway: <default("WirelessHost")> like INetworkNode {
           @display("p=9.092353,0.8635196;i=misc/sensorgateway");
        }
}
```

Figura 21: Definição da rede.

O cenário aqui definido cria uma área de 18,75 m por 2,6 metros que representa o caminhão. O nó *gateway*, também chamado de *gateway*, foi colocado no centro do espaço. O número de sensores distribuídos é modificável e controlado pela variável `sensorCount`. Em cada simulação, os outros sensores são dispostos de forma aleatória no ambiente. Propriedades adicionais do cenário tais como valores sobre o consumo de energia, valores de operação do rádio e taxa de amostragem podem ser colocados em um arquivo INI, o qual serve como arquivo de configurações iniciais para um conjunto de simulações. A parte desse arquivo que define as propriedades do RI-MAC e taxa de amostragem pode ser vista na Fig. 22.

5.2 Resultados

Os resultados foram obtidos executando 3 conjuntos de simulações, uma para cada um dos protocolos explorados na Seção 2. Foi usado um número de nós entre 5 e 100 com incrementos de 5. Para cada densidade, foram feitas 20 execuções. Os resultados poderiam ser analisados no editor integrado do OMNeT++, uma vez que ele tem capacidades gráficas, mas ferramentas como a linguagem

```
[Config SensorCountRIMac]

# Basic config
sim-time-limit = 10h
repeat = 20
*.sensorCount = 20
**.vector-recording = false

# Data generation
*.sensors[*].app[0].sendInterval = 100s
*.sensors[*].app[0].startTime = exponential(1s)

# MAC config
**.wlan[*].mac.typename = "RIMac"
**.wlan[*].mac.headerLength = 6B
**.wlan[*].mac.ccaInterval =  0.02s

# Define variable for study
**.wlan[*].mac.sleepInterval =${slotDuration=0.05..1 step 0.05}s

#...#
```

Figura 22: Definição da rede.

de programação Python oferecem maior flexibilidade. Por isso, os resultados foram exportados e se utilizou o módulo matplotlib (Hunter, 2007) para fazer as visualizações seguintes.

A Fig. 23 mostra o número total de pacotes que foram entregues ao *gateway* e os compara ao caso ideal em que todos os pacotes são entregues. Como se pode ver, o RI-MAC tem a maior flexibilidade entre os protocolos estudados. O T-MAC tem um funcionamento relativamente estável, mas sua capacidade de transferência é drasticamente reduzida depois de atingir um certo ponto. O B-MAC começa com um bom índice de sucesso, mas ele não consegue se adaptar ao crescimento do número de nós na rede, fazendo com que o número de pacotes que consegue entregar fique estagnado.

Em termos de consumo energético, o B-MAC e o RI-MAC tem um crescimento gradual. O B-MAC tem um consumo menor em termos absolutos, mas o fato de o RI-MAC conseguir entregar mais pacotes explica essa discrepância. O T-MAC, em contrapartida, tem um salto substancial no consumo em redes com mais de 50 sensores como pode ser visto na Fig. 24. Isso ocorre por conta do mecanismo de extensão do período que um nó permanece acordado. À medida que o número de transmissões aumenta, nós permanecem acordados durante mais tempo até o ponto que a rede fica saturada. Quando isso acontece, os sensores passam a gastar todo o seu tempo ativos, o que maximiza o consumo energético.

Considerando as características desejadas para o cenário apresentado, o RI-MAC é o melhor dos protocolos estudados. Mesmo que o cenário fosse diferente, ainda seria simples fazer uma avaliação. Além do INET guardar estatísticas sobre diversas facetas dos protocolos, também é possível mensurar outras propriedades dentro do código do protocolo. Por exemplo, poderia se guardar as quantidades de sinais de *beacon* que cada nó envia ou o tempo que cada pacote leva para chegar ao *gateway*. As possibilidades são tão variadas quanto as aplicações para RSSFs.

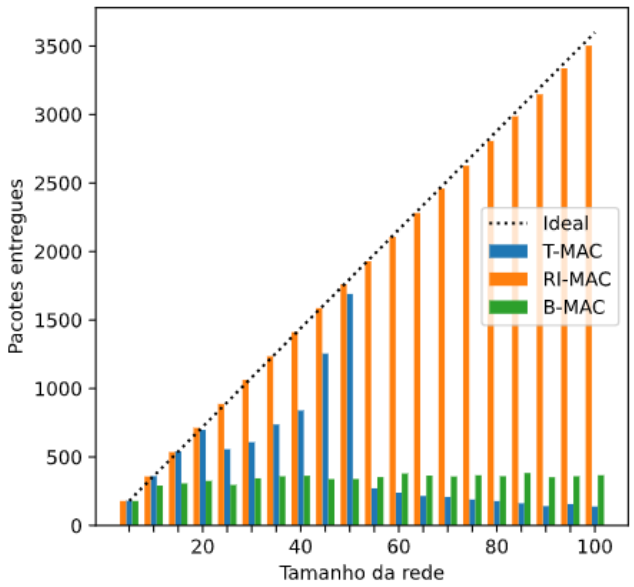

Figura 23: Número de pacotes entregues à medida que o tamanho da rede aumenta.

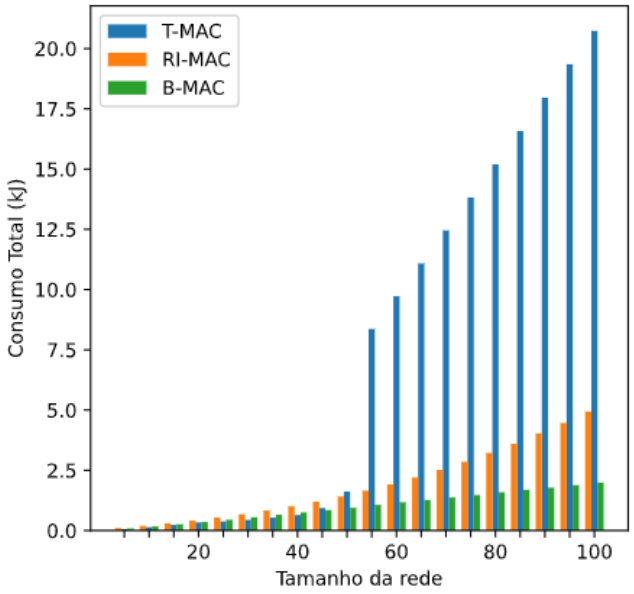

Figura 24: Consumo total de energia dependendo do número de nós na rede.

6 Conclusão

Redes de sensores sem fio estão sendo aplicadas a contextos cada vez mais variados. Entretanto, os limites fundamentais dessa tecnologia se mantiveram relativamente estáveis durantes os últimos anos. A falta de uma fonte confiável de energia e o tamanho reduzido dos nós limitam seu tempo de vida, forçando a sua otimização para cada aplicação. Existem diversas formas de testar novas estra-

tégias para a redução do consumo, mas análises teóricas podem não refletir as condições no mundo real e testes usando equipamentos reais são caros e difíceis de replicar.

Este artigo explorou a terceira opção: simulações. Essa estratégia permite maior confiança nos resultados adquiridos com custos muito menores e demandando menos tempo para pesquisadores. Primeiramente foram definidos os protocolos que se queria comparar e os simuladores disponíveis foram comparados. Depois disso, foram mostrados os pré-requisitos e o procedimentos para implementar esses protocolos no simulador OMNeT++ juntamente com a biblioteca INET. Por fim, os resultados foram avaliados usando a linguagem de programação Python.

Referências

Alfayez, F., Hammoudeh, M. and Abuarqoub, A. (2015). A survey on mac protocols for duty-cycled wireless sensor networks, *Procedia Computer Science* **73**: 482–489. https://doi.org/10.1016/j.procs.2015.12.034.

Anastasi, G., Conti, M., Di Francesco, M. and Passarella, A. (2009). Energy conservation in wireless sensor networks: A survey, *Ad Hoc Networks* **7**(3): 537–568. http://dx.doi.org/10.1016/j.adhoc.2008.06.003.

Apple Inc. (n.d.). macos mojave. Disponível em https://www.apple.com/macos/mojave/.

Bakni, M., Manuel, L., Chacón, M., Cardinale, Y., Terrasson, G. and Curea, O. (2020). WSN Simulators Evaluation: An Approach Focusing on Energy awareness, *International journal of wireless mobile networks (IJWMN)* **11**(6): 1–20. https://dx.doi.org/10.5121/ijwmn.2019.11601.

Begum, K. and Dixit, S. (2016). Industrial WSN using IoT: A survey, *International Conference on Electrical, Electronics, and Optimization Techniques, ICEEOT 2016* pp. 499–504. http://dx.doi.org/10.1109/ICEEOT.2016.7755660.

Bott, E. and Stinson, C. (2019). *Windows 10 inside out*, Microsoft Press.

Boulis, T. and Pediaditakis, D. (2016). Castalia. Disponível em https://github.com/boulis/Castalia.

Caldeira, J. M. L. P., Rodrigues, J. J. P. C. and Lorenz, P. (2013). Intra-Mobility Support Solutions for Healthcare Wireless Sensor Networks–Handover Issues, *IEEE Sensors Journal* **13**(11): 4339–4348. http://dx.doi.org/10.1109/JSEN.2013.2267729.

Cogitative Software FZE (2021). OMNEST - High-Performance Simulation for All Kinds of Networks. Disponível em https://omnest.com/.

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment, *Computing in Science & Engineering* **9**(3): 90–95. http://dx.doi.org/10.1109/MCSE.2007.55.

Ibrahim, A. A. and Bayat, O. (2020). Medium Access Control Protocol-based Energy and Quality of Service routing scheme for WBAN, *HORA 2020 - 2nd International Congress on Human-Computer Interaction, Optimization and Robotic Applications, Proceedings* pp. 9–14. http://dx.doi.org/10.1109/HORA49412.2020.9152849.

Kadir, E. A., Siswanto, A., Rosa, S. L., Syukur, A., Irie, H. and Othman, M. (2019). Smart sensor node of wsns for river water pollution monitoring system, *2019 International Conference on Advanced Communication Technologies and Networking (CommNet)*, pp. 1–5. http://dx.doi.org/10.1109/COMMNET.2019.8742371.

Lopez, E., Monteiro, J., Carrasco, P., Saenz, J., Pinto, N. and Blazquez, G. (2019). Development, implementation and evaluation of a wireless sensor network and a web-based platform for the monitoring and management of a microgrid with renewable energy sources, *SEST 2019 - 2nd International Conference on Smart Energy Systems and Technologies* . http://dx.doi.org/10.1109/SEST.2019.8849016.

Lopez, P. A., Behrisch, M., Bieker-Walz, L., Erdmann, J., Flötteröd, Y.-P., Hilbrich, R., Lücken, L., Rummel, J., Wagner, P. and Wießner, E. (2018). Microscopic traffic simulation using sumo, *The 21st IEEE International Conference on Intelligent Transportation Systems*, IEEE. https://doi.org/10.1109/ITSC.2018.8569938.

MATLAB (2010). *version 7.10.0 (R2010a)*, The MathWorks Inc., Natick, Massachusetts.

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment, *Linux journal* **2014**(239): 2.

Nabi, M., Blagojevic, M., Geilen, M., Basten, T. and Hendriks, T. (2010). MCMAC: An optimized medium access control protocol for mobile clusters in wireless sensor networks, *SECON 2010 - 2010 7th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks* . http://dx.doi.org/10.1109/SECON.2010.5508200.

NSNAM (2021a). ns-3 | a discrete-event network simulator for internet systems. Disponível em https://www.nsnam.org/.

NSNAM (2021b). The Network Simulator - ns-2. Disponível em https://www.isi.edu/nsnam/ns/.

OpenSim Ltd (2021a). INET Framework. Disponível em https://inet.omnetpp.org/.

OpenSim Ltd (2021b). OMNeT++ Discrete Event Simulator. Disponível em https://omnetpp.org/.

Polastre, J., Hill, J. and Culler, D. (2004). Versatile low power media access for wireless sensor networks, *Proceedings of the 2nd international conference on Embedded networked sensor systems - SenSys '04*, ACM Press, p. 95. http://dx.doi.org/10.1145/1031495.1031508.

Polavarapu, S. C. and Panda, S. K. (2020). A survey on industrial applications using mems and wsn, *2020 Fourth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, pp. 982–986. http://dx.doi.org/10.1109/I-SMAC49090.2020.9243514.

Pottie, G. J. and Kaiser, W. J. (2000). Wireless integrated network sensors, *Communications of the ACM* **43**(5): 51–58. http://dx.doi.org/10.1145/332833.332838.

Radha, S., Bala, G. J. and Nagabushanam, P. (2019). Multilayer mac with adaptive listening for wsn, *2019 Third International Conference on Inventive Systems and Control (ICISC)*, pp. 15–21. https://dx.doi.org/10.1109/ICISC44355.2019.9036423.

Rafael Cotrim, J. C., Soares, V. and Azzoug, Y. (2021). Power saving MAC protocols in wireless sensor networks: a survey, **19**(6): 1778. http://dx.doi.org/10.12928/telkomnika.v19i6.19148.

Rafael Cotrim, J. C., Soares, V. and Gaspar, P. (2021). Power saving MAC protocols in wireless sensor networks: A performance assessment analysis, **6**(4): 341–347. http://dx.doi.org/10.25046/aj060438.

Sanders, C. (2017). *Practical packet analysis : using Wireshark to solve real-world network problems*, No Starch Press, San Francisco.

Singh, M. K., Amin, S. I., Imam, S. A., Sachan, V. K. and Choudhary, A. (2018). A survey of wireless sensor network and its types, *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, pp. 326–330. http://dx.doi.org/10.1109/ICACCCN.2018.8748710.

Sobell, M. G. (2015). *A practical guide to Ubuntu Linux*, Pearson Education.

Stroustrup, B. (2000). *The C++ programming language*, Pearson Education India.

Sun, Y., Gurewitz, O. and Johnson, D. B. (2008). RI-MAC: A Receiver-Initiated Asynchronous Duty Cycle MAC Protocol for Dynamic Traffic Loads in Wireless Sensor Networks, *Proceedings of the 6th ACM conference on Embedded network sensor systems - SenSys '08*, Vol. 81 LNICST, ACM Press, p. 1. http://dx.doi.org/10.1145/1460412.1460414.

Tetcos (2021). NetSim-Network Simulator & Emulator. Disponível em https://www.tetcos.com/.

Ukhurebor, K. E., Odesanya, I., Tyokighir, S. S., Kerry, R. G., Olayinka, A. S. and Bobadoye, A. O. (2021). Wireless sensor networks: Applications and challenges, *in* S. S. Yellampalli (ed.), *Wireless Sensor Networks*, IntechOpen, Rijeka, chapter 2. https://doi.org/10.5772/intechopen.93660.

Van Dam, T. and Langendoen, K. (2003). An adaptive energy-efficient MAC protocol for wireless sensor networks, *SenSys'03: Proceedings of the First International Conference on Embedded Networked Sensor Systems* pp. 171–180. http://dx.doi.org/10.1145/958491.958512.

Van Rossum, G. and Drake, F. L. (2009). *Python 3 Reference Manual*, CreateSpace, Scotts Valley, CA.