Marcelo Arza Lobo da Costa

# Análise de duty-cycling para economia de energia na disseminação de código em rede de sensores

## DISSERTAÇÃO DE MESTRADO

DEPARTAMENTO DE INFORMÁTICA
Programa de Pós–Graduação em Informática

Rio de Janeiro
Setembro de 2015

Pontifícia Universidade Católica
do Rio de Janeiro

**Marcelo Arza Lobo da Costa**

# Análise de duty-cycling para economia de energia na disseminação de código em rede de sensores

**Dissertação de Mestrado**

Dissertação apresentada ao Programa de Pós–Graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio, como requisito parcial para obtenção do grau de Mestre em Informática.

Orientador: Profª. Noemi de La Rocque Rodriguez

Rio de Janeiro
Setembro de 2015

Pontifícia Universidade Católica
do Rio de Janeiro

**Marcelo Arza Lobo da Costa**

# Análise de duty-cycling para economia de energia na disseminação de código em rede de sensores

Dissertação apresentada ao Programa de Pós–Graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio como requisito parcial para obtenção do grau de Mestre em Informática. Aprovada pela Comissão Examinadora abaixo assinada.

**Profª. Noemi de La Rocque Rodriguez**
Orientador
Departamento de Informática — PUC–Rio

**Profª. Silvana Rossetto**
UFRJ

**Dr.. Francisco Figueiredo Goytacaz Sant'Anna**
Consultor Autônomo

**Prof. José Eugenio Leal**
Coordenador Setorial do Centro Técnico Científico — PUC–Rio

Rio de Janeiro, 08 de setembro de 2015

**Marcelo Arza Lobo da Costa**

Graduou–se em Ciência da Computação pela Universidade Federal do Rio de Janeiro (UFRJ). Trabalha há 9 anos na Petrobras como Administrador de Rede Windows e gerenciamento de serviços (Active Directory, DNS, DHCP, GPO, servidores de arquivo, replicação de dados, DFS, banco de dados SQL, WSUS, System Center e virtualização)



Ficha Catalográfica

Costa, Marcelo Arza Lobo da

Análise de duty-cycling para economia de energia na disseminação de código em rede de sensores / Marcelo Arza Lobo da Costa; orientador: Noemi de La Rocque Rodriguez. — 2015.

49 f. : il. (color.); 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Departamento de Informática, 2015.

Inclui bibliografia.

1. Informática – Teses. 2. RSSF. 3. Disseminação. 4. Economia de Energia. 5. Duty cycling. I. Rodriguez, Noemi de La Rocque. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Dedico este trabalho à minha esposa Aline e a nossa filha Giovana, que é uma benção nas nossas vidas e a pessoa que mais amo no mundo, a minha mãe Barbara e aos meus avós Gloria e Fernando *(in memoriam)* por tudo o que fizeram por mim durante toda a minha vida.

## Agradecimentos

Em primeiro lugar quero agradecer a Deus, pois sem ele nada existe.

Quero agradecer à minha orientadora Professora Noemi Rodriguez. Por ter acreditado no meu trabalho e ter aceitado me orientar mesmo trabalhando e podendo me dedicar somente uma parte do tempo. Pelo apoio e incentivo para a realização deste trabalho e que fez com que não desistisse do mesmo. Obrigado pela oportunidade! E obrigado pelos ensinamentos!

À minha esposa Aline, que nesses quase três anos, sempre me incentivou e me apoiou para que eu concluísse essa etapa da minha vida. Agradeço pela compreensão nos momentos de minha ausência e pelo amor e carinho que tem por mim.

À minha mãe Barbara e aos meus avós Gloria e Fernando *(in memoriam)* que sempre incentivaram meus estudos, que sempre me deram carinho, amor e afeto, e que sempre me apoiaram em todas as decisões que tomei na vida.

Ao pesquisador Adriano Branco, pelas inúmeras conversas e trocas de experiência e pelas palavras de apoio que sempre deram um ânimo para o desenvolvimento do trabalho.

À Petrobras e à PUC–Rio, pelos auxílios concedidos, sem os quais este trabalho não poderia ter sido realizado.

À todos os colegas, professores e funcionários do Departamento de Informática da PUC–Rio, pelo companheirismo, aprendizado e auxílio.

Aos colegas de trabalho da Petrobras que se sacrificaram um pouco mais na minha ausência para que eu conseguisse terminar o trabalho.

Aos membros da banca, pelas valiosas contribuições.

## Resumo

Costa, Marcelo Arza Lobo da; Rodriguez, Noemi de La Rocque. **Análise de duty-cycling para economia de energia na disseminação de código em rede de sensores**. Rio de Janeiro, 2015. 49p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Um dos principais desafios em redes de sensores sem fio (RSSF) é reduzir o consumo de energia dos nós sensores. Um método usado para economizar a bateria que alimenta os nós sensores é o *duty cycling* (DC) do rádio, onde o rádio fica desligado na maior parte do tempo e fica ligado por pouco tempo para verificar se existe alguma mensagem. O DC é usado com frequência em aplicações de monitoramento onde apenas uma mensagem é transmitida depois da leitura do sensor. Geralmente a leitura do sensor só volta a acontecer depois de minutos, logo poucas mensagens são transmitidas por unidade de tempo. Neste trabalho, analisamos o uso da técnica de DC em um contexto diferente, o da disseminação de código, onde várias mensagens são enviadas em um curto espaço de tempo, e que usa mensagens *broadcast*, ao contrário do monitoramento, que utiliza mensagens *unicast*. Analisamos dois algoritmos de disseminação específicos, um para um ambiente de máquinas virtuais executando nos motes, onde o código disseminado é um *script* com tamanho da ordem de *bytes*, e outro para disseminação de códigos da aplicação inteira, onde o tamanho é bem maior que no caso do *script*, da ordem de kbytes. O objetivo deste trabalho foi avaliar qual o impacto do DC na latência e quanto de energia foi economizado quando comparado a deixar o rádio ligado o tempo todo, que é como ambos algoritmos funcionam em sua forma original.



## Palavras–chave

RSSF; Disseminação; Economia de Energia; Duty cycling.

## Abstract

Costa, Marcelo Arza Lobo da; Rodriguez, Noemi de La Rocque (Advisor). **Analysis of duty-cycling for saving energy in code dissemination over sensor networks**. Rio de Janeiro, 2015. 49p. MSc. Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

One of the key challenges in wireless sensor networks (WSN) is to save energy at motes. One method to save battery is radio duty cycling (DC), which keeps the radio turned off in most of the time and turns the radio on for a short time to verify if any there are any messages. DC is frequently used in monitoring applications where only one message is transmitted after the mote reads its sensor. Usually the mote reads its sensor only once every few minutes, so few unicast messages are transmitted in the network per time unit. This work analyzes the use of the DC method in code dissemination. In this context, multiple broadcast messages are transmitted in a short time. We examined two specific dissemination algorithms, one of them proposed for a virtual machine environment, in which the disseminated code is a small script, and a second one originally proposed for disseminating the code of an entire application, typically much larger than a script. The objective of this study is to evaluate the impact of DC on latency and how much energy was saved when compared to leaving the radio on all the time, which is how both algorithms work in their original form.



## Keywords

WSN; Dissemination; Energy Saving; Duty Cycling.

# Sumário

# Lista de figuras

# Lista de tabelas

# 1
# Introdução

Um dos principais desafios em redes de sensores sem fio (RSSF) é reduzir o consumo de energia dos nós sensores (*motes*). Os motes são alimentados em geral por um par de pilhas definindo o tempo de vida útil de cada um deles. É importante reduzir o consumo de energia dos motes para aumentar o tempo de vida de toda a rede, uma vez que trocar a fonte de alimentação pode ser complicado devido ao tamanho da rede ou ao difícil acesso físico que cada mote foi instalado (Talzi et al., 2007).

Uma aplicação frequente em RSSF é a de monitoramento. Basicamente seu funcionamento ocorre com cada mote lendo a informação do sensor, e em seguida, enviando-a pela rede. Esse tipo de aplicação, até pelas características das RSSF, funciona com dezenas ou mesmo centenas de motes. Caso a aplicação que esteja rodando em cada um dos motes precise ser alterada, a forma mais interessante é através de um algoritmo de disseminação do código da aplicação (Kulik et al., 2002).

A desvantagem do uso desses algoritmos é o gasto de energia provocado pela disseminação que é feita pelo rádio, principal componente no consumo de energia (Raghunathan et al., 2002) dos motes. O rádio consome energia não apenas quando está transmitindo ou recebendo algum dado, mas também durante o tempo em que está apenas ligado sem tratar nenhuma mensagem destinada ao mote.

O tempo em que o rádio fica ligado sem transmitir ou receber qualquer informação é conhecido como *idle listening*. Uma forma de aumentar a vida útil do mote é reduzir seu tempo de idle listening. Um método usado para economizar a bateria que alimenta os motes é o *duty cycling* (DC) do rádio (Polastre et al., 2004). Nesta abordagem cada mote fica com o rádio ligado por apenas um curto espaço de tempo para verificar se existe alguma mensagem, desligando o rádio em seguida e mantendo-o desligado por um período de tempo bem mais longo que o que manteve o rádio ligado. Após o tempo que o rádio ficou desligado, o mote liga-o novamente, repetindo indefinidamente esse ciclo.

O DC é usado com frequência em aplicações de monitoramento onde apenas uma mensagem é transmitida depois da leitura do sensor (Lu et al., 2004). Geralmente, neste tipo de aplicação, a leitura do sensor só volta a acontecer depois de minutos, logo poucas mensagens são transmitidas por unidade de

tempo. Essa mesma característica de realizar uma tarefa e em seguida ficar somente aguardando o tempo de realizar a mesma tarefa novamente é a base do DC. Neste trabalho, analisamos o uso da técnica de DC em um contexto diferente, o da disseminação de código, onde várias mensagens são enviadas em um curto espaço de tempo, e que usa mensagens *broadcast*, ao contrário do monitoramento, que utiliza mensagens *unicast*.

O trabalho foi parcialmente motivado pelo sistema Terra (Branco et al., 2015) que implementa um ambiente de programação baseado em uma linguagem reativa de alto-nível com execução segura e utiliza uma máquina virtual que integra componentes customizados. Terra implementa uma versão da linguagem Céu (Sant'Anna et al., 2013) para a construção desses componentes e depende de um algoritmo de disseminação para distribuir o *script* contendo os componentes da máquina virtual para todos os motes da rede.

Neste trabalho analisamos dois algoritmos de disseminação específicos, o algoritmo utilizado em Terra, onde o código é um script com tamanho da ordem de *bytes*, e o Deluge (Hui e Culler, 2004), onde o código disseminado é a aplicação inteira existente no mote, cujo tamanho é bem maior que no caso do script, da ordem de kbytes. O objetivo deste trabalho foi avaliar qual o impacto do DC na latência e quanto de energia foi economizado quando comparado a deixar o rádio ligado o tempo todo, que é como ambos algoritmos funcionam em sua forma original.

O conhecimento mais preciso sobre esse *tradeoff* pode apoiar decisões como diminuir o tempo de disseminação em detrimento do consumo de energia, ou até mesmo sobre os parâmetros de DC apropriados para a densidade da rede e o tamanho do código que está sendo transmitido pela rede.

No capítulo 2 descrevemos os conceitos necessários para um melhor entendimento deste trabalho, como o duty cycling e os algoritmos de disseminação implementados nos experimentos realizados. No capítulo 3 indicamos quais tecnologias foram utilizadas no ambiente de execução dos experimentos. Em seguida, descrevemos os experimentos realizados e os resultados obtidos no capítulo 4. Por último, no capítulo 5 temos as conclusões desse trabalho.

# 2
# Conceitos

Neste capítulo descrevemos as características e os componentes que compõem uma RSSF. Na seção 2.1 apresentamos as características de uma RSSF e descrevemos sua definição e arquitetura, em seguida, na seção 2.2 explicamos o funcionamento do protocolo MAC que implementa o método de duty cycling do rádio e, finalmente, na seção 2.3 detalhamos o comportamento dos algoritmos de disseminação.

Figura 2.1: Arquitetura de uma RSSF.

Na figura 2.1 temos um desenho de uma RSSF típica. Um dos motes, conhecido como estação base, fica ligado a um computador convencional, geralmente pela porta serial ou USB. A estação base é o único mote onde não há necessidade de preocupação com o consumo de energia, uma vez que sua alimentação é provida pelo computador anexo à ela. É pela comunicação do computador com a estação base que se inicia a disseminação.

## 2.1
## Características de uma RSSF

Uma rede de sensores é um grupo de pequenos sistemas autônomos denominados de motes. Estes motes cooperam para resolver pelo menos um problema comum. Geralmente suas funções incluem algum tipo de percepção de parâmetros físicos.

Um mote é um sistema que tem capacidade de comunicação, computação, sensoriamento e armazenagem. Esses motes miniaturizados operam em restrições severas em termos de recursos disponíveis, como a energia da bateria, memória de programa, largura de banda disponível e poder de processamento. A figura 2.2 mostra um diagrama esquemático dos componentes de um mote. Cada mote é composto por um micro-controlador, fonte de alimentação, transceptor de Rádio Freqüência (RF), memória externa e sensores.

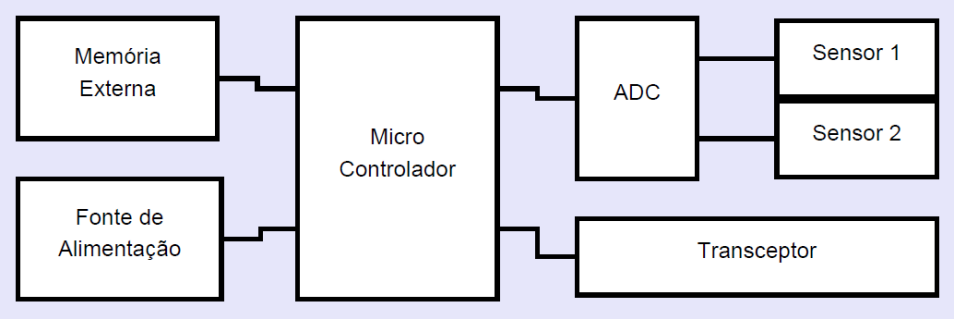


Figura 2.2: Módulos de um Mote

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

Devido à natureza inacessível desses nós sensores, essas redes devem ser simultaneamente autônomas e de longa duração. Os motes precisam sobreviver às duras condições ambientais e conservar o máximo de energia possível.

## 2.2 Duty Cycling

O DC pode ser síncrono (Ye et al., 2002) ou assíncrono (Buettner et al., 2006). Na abordagem síncrona, o momento de ligar o rádio dos motes que vão se comunicar um com o outro é sincronizado. A principal desvantagem em relação à abordagem síncrona é a sobrecarga adicional de energia consumida para transmitir as mensagens de controle que fazem o sincronismo entre os motes. Os motes precisam enviar alguma informação de sincronização periodicamente, mesmo quando não existe nenhuma informação para ser transmitida entre os motes, para que não percam o sincronismo com o passar do tempo. Nosso estudo está focado apenas em DC assíncrono.

O *Low power listening* (LPL) (Polastre et al., 2004) foi o primeiro método adotado, transmitindo um longo preâmbulo antes de transmitir a informação desejada, o *payload*. A duração do preâmbulo é pelo menos o período de tempo

em que o rádio dos motes fica desligado. Dessa forma, quando um mote liga o rádio e verifica o meio físico, é garantido que vai perceber o preâmbulo. Quando um mote detecta o preâmbulo, ele mantém o rádio ligado para receber o payload. O LPL é o mecanismo existente no B-MAC (Polastre et al., 2004).

O B-MAC foi criado em um tempo onde os rádios enviavam uma sequência de bits. Portanto, o preâmbulo longo era bem adequado para a verificação do meio físico no rádio quando o rádio era ligado. Atualmente, os novos modelos de rádios encapsulam a informação em pacotes antes de transmitir. Por conta disso, o uso do B-MAC fica restrito e não foi considerado neste trabalho.

O X-MAC (Buettner et al., 2006) substituiu o B-MAC para ser usado nos modelos de rádios mais novos, onde a informação vai encapsulada em pacotes. O preâmbulo longo foi substituído por uma série de preâmbulos curtos, cada um deles contendo o endereço de destino do mote receptor, permitindo que motes que não sejam destinatários daquela informação possam voltar a desligar o rádio sem ter que esperar todo o tempo de envio dos preâmbulos. A quantidade de preâmbulos curtos é determinada pela duração de tempo em que o nó sensor receptor fica com o rádio desligado.

O BoX-MAC (Moss e Levis, 2008) é uma variação do X-MAC que habilita a terminação precoce do envio dos preâmbulos curtos consecutivos deixando os intervalos entre eles maiores para que *acknowledgments* do mote receptor sejam enviados na comunicação ponto-a-ponto. Além disso, o BoX-MAC substituiu o preâmbulo curto pelo payload direto reduzindo a sobrecarga do envio de um preâmbulo. A figura 2.3 ilustra como o payload é transferido quando se usa utiliza o BoX-MAC na comunicação ponto-a-ponto.

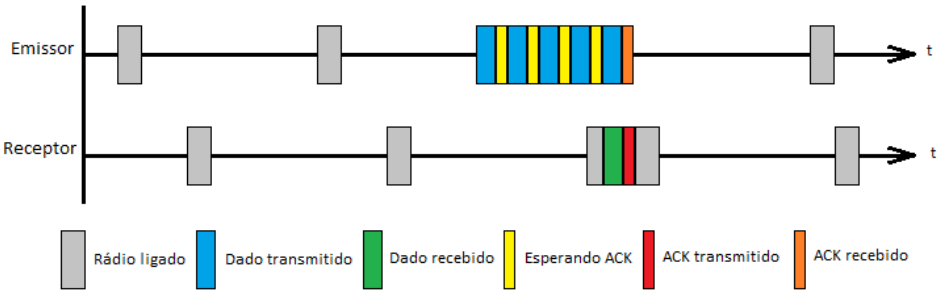


Figura 2.3: Implementação do BoX-MAC na comunicação ponto-a-ponto

Quando a comunicação ponto-multiponto é utilizada no BoX-MAC, o mote emissor da mensagem broadcast não recebe acknowledgments dos motes receptores e o payload é enviado pelo mesmo tempo que os motes ficam com o rádio desligado. Essa duração é configurável e esse valor é utilizado para garantir a entrega do payload uma vez que o emissor enviando o payload

durante todo o tempo que o mote fica com o rádio desligado, pelo menos uma mensagem será recebida.

O TinyOS 2.x (Levis et al., 2005) é um dos sistemas operacionais mais utilizados para a criação de aplicações de RSSF. O sistema introduziu uma abstração de hardware com uma arquitetura de três camadas. A camada inferior fornece acesso aos recursos básicos. A camada do meio tem interfaces em alto nível que fornecem abstrações de todos os recursos do hardware. Finalmente, a camada superior apresenta abstrações que são independentes de hardware e, portanto, permanece igual independentemente da plataforma. O TinyOS usa a linguagem NesC (Gay et al., 2003), que tem funcionalidades para suporte em plataformas de rede. Utilizamos como protocolo da camada MAC o mesmo escolhido pelo TinyOS, o BoX-MAC.

Vamos considerar para o cálculo do percentual do duty cycling (DC) o tempo que o rádio está ligado dividido pela soma dos tempos que o rádio está ligado e desligado. Por exemplo, obtemos um DC de 10% configurando o tempo de 11 ms para o rádio ligado e 99 ms para o tempo de rádio desligado. Para simplificar usaremos no texto o termo DC X% para indicar o duty cycling de X%.

## 2.3 Disseminação de Código

De acordo com Brown e Sreenan (Brown e Sreenan, 2013) a disseminação de código é apenas uma maneira para modificar a aplicação que está sendo executada nos motes. A disseminação de código pode ser vista como um inundação confiável dentro da RSSF, pois todos os motes precisam receber todas as mensagens enviadas da estação base contendo a nova aplicação. Na inundação convencional, que funciona essencialmente com todos os motes re-enviando as mensagens broadcasts recebidas, não existe garantia de recebimento por partes dos motes, ocorrendo entrega por melhor esforço. Como colisões e falhas no recebimento das mensagens acontecem dentro das RSSF, algoritmos de disseminação tratam essas falhas, tornando-os confiáveis em relação à inundação. Um algoritmo de disseminação também pode decidir quais mensagens recebidas devem ser encaminhadas para os vizinhos para evitar um aumento de latência e consumo de energia decorrente de retransmissões por colisões.

Nesse estudo escolhemos analisar dois protocolos de disseminação de código: o Deluge (Hui e Culler, 2004) e o existente dentro do sistema Terra, que vamos chamar a partir deste momento de DisCoTerra. Assim, como outros algoritmos de disseminação de código, os dois algoritmos analisados, Deluge e DisCoTerra usam o conceito de *advertisement*. O mote origem envia uma

mensagem broadcast, que chamaremos de ADV, para os motes vizinhos. Cada um desses motes envia uma mensagem ao mote origem solicitando o novo código. Esta solicitação, chamaremos de REQ, e pode ser broadcast ou unicast, dependendo do algoritmo de disseminação de código. Por fim, o mote origem envia uma ou mais mensagens broadcast com o novo código, que chamaremos de DATA.

O início de uma disseminação ocorre a partir de uma mensagem ADV da estação base. Essa mensagem é enviada depois que o novo código é completamente recebido pela conexão serial entre o computador e a estação base. Os motes que estão no alcance da estação base, ao receberam a mensagem ADV, ficam informados dessa nova versão. Cada mote que recebeu a mensagem ADV retransmite a mensagem para alcançar novos motes. Em seguida ao envio das mensagens ADV, cada mote envia uma mensagem REQ solicitando o novo código. Então a estação base se prepara para enviar um número x de mensagens DATA broadcast de forma consecutiva.

### 2.3.1 Deluge

O TinyOS disponibiliza como protocolo de disseminação de código o Deluge. Ele foi concebido com o intuito de propagar o código inteiro da aplicação a todos os motes da rede. Em geral o tamanho inteiro da aplicação é maior que a capacidade – bastante reduzida – das memórias RAM dos motes. Sendo assim, ao receber as mensagens DATA com as partes do novo código, um mote vai escrevendo-as na memória flash do mote para conseguir guardar toda a aplicação e, no momento que estiver completa, conseguir carregar o novo código no mote. O Deluge utiliza um sistema de *boot* que o TinyOS deixa disponível para carregar o novo código depois que o mote inicia.

Para economizar memória, o Deluge divide o tamanho inteiro da aplicação em páginas de tamanho fixo e faz o controle do que já foi recebido utilizando um mapa de bits. Cada página é dividida em um número fixo de pacotes, que é efetivamente o que será transferido. Sendo assim, o número de bits que precisa ser guardado em memória para verificação é o número de pacotes e não toda a aplicação. A mensagem ADV só é transmitida depois que uma nova página está disponível. A página só está disponível no Deluge se todos os bits foram marcados, ou seja, todos os pacotes daquela página foram recebidos corretamente. Além disso, todas as páginas anteriores à página atual também estarão disponíveis. O mote só envia aos vizinhos uma mensagem ADV contendo um número de página n como disponível se todas as páginas anteriores a n também já tiverem sido recebidas. Essa indicação é sempre crescente

conforme pode ser visto pelo diagrama de sequência da Figura 2.4.

No Deluge a mensagem ADV contém um número de identificação para o novo código a ser disseminado, a quantidade total de páginas desse novo código e o número de páginas já disponíveis para serem transmitidas. A mensagem ADV inicial é transmitida pela estação base depois que o novo código foi escrito inteiramente na memória Flash da estação base. A única diferença entre a mensagem ADV transmitida pela estação base e a re-enviada pelos motes vizinhos é no número de páginas já disponíveis, uma vez que os vizinhos da estação base não possuem nenhuma página quando transmitem ADV e, mandam a informação como zero. Dessa forma nenhum desses motes receberá mensagens REQ requisitando dados do novo código. Os motes vizinhos à estação base enviam um mensagem REQ unicast para a estação base e ficam aguardando o recebimento das x mensagens DATA que serão transmitidas pela estação base, onde x é o número de pacotes da página. No momento em que cada mote estiver com todos os pacotes referentes a primeira página recebidos, encaminha uma nova mensagem ADV broadcast trocando a informação de páginas disponíveis para um.

O Deluge sempre funciona tratando as páginas disponíveis de forma crescente. Se um mote recebe uma mensagem ADV onde o número de páginas disponíveis é quatro, isso significa que todas as páginas até quatro estão escritas em sua memória flash e, portanto, disponíveis para serem transmitidas aos motes vizinhos. O Deluge prioriza as páginas de menor valor. Se o mote estiver enviando pacotes de uma página e chega uma mensagem REQ solicitando uma página menor, ele põe na frente do conjunto de mensagens a ser enviado esses pacotes da página de menor valor para somente depois continuar enviando os pacotes da página de maior valor que estava enviando.

Os motes primeiro tratam as mensagens REQ que chegam dos seus vizinhos, e assim, enviam todas as mensagens DATA correspondentes, para somente depois enviar uma mensagem REQ solicitando a próxima página. O controle de espera de mensagens REQ, envio de mensagens REQ e retransmissão de mensagens ADV dentro do mote é feito usando o algoritmo Trickle (Levis et al., 2004). O Deluge foi concebido de tal forma a fazer um *pipeline* na disseminação. Os motes a um salto da estação base podem estar transferindo uma página com um determinado identificador enquanto os motes a três saltos de distância da estação base podem estar transferindo uma página com um identificador menor.

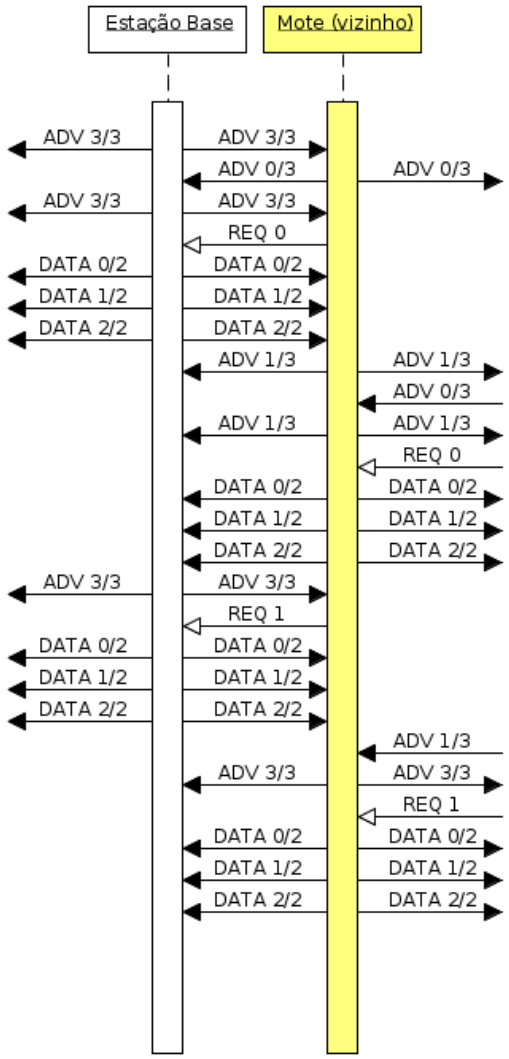


Figura 2.4: Diagrama de sequência do Deluge

### 2.3.2 DisCoTerra

O DisCoTerra é o algoritmo de disseminação existente no sistema Terra. No ambiente Terra as aplicações rodam sobre máquinas virtuais e a disseminação distribui apenas o bytecode do script da aplicação, que tem tamanho muito menor do que o código binário de um executável completo, que é o que o Deluge distribui. Sendo assim, o DisCoTerra não se preocupou com o tamanho do mapa de bits como no Deluge para controle do que já foi recebido. O DisCoTerra divide o tamanho total do script pelo tamanho da mensagem transmitida e o resultado dessa operação é o tamanho do mapa de bits.

Como todo script fica carregado em memória RAM para execução, o DisCoTerra não tem necessidade de escrever em memória Flash a nova informação que é recebida, gastando menos tempo que o Deluge na hora que

um novo pacote é recebido. A disseminação começa quando a estação base recebe do computador pela serial um novo script Céu, conforme o diagrama de sequência das mensagens da Figura 2.5. Então a estação base envia uma mensagem ADV broadcast indicando que existe um novo script e a quantidade de mensagens. Cada mote que recebeu a mensagem envia uma mensagem REQ broadcast. Como o único mote que possui essa informação nesse momento é a estação base, este é o único mote a enviar a primeira mensagem DATA broadcast. Todas as mensagens DATA restantes para terminar a disseminação são enviadas consecutivamente sem que nenhuma outra mensagem precise ser enviada para a estação base.

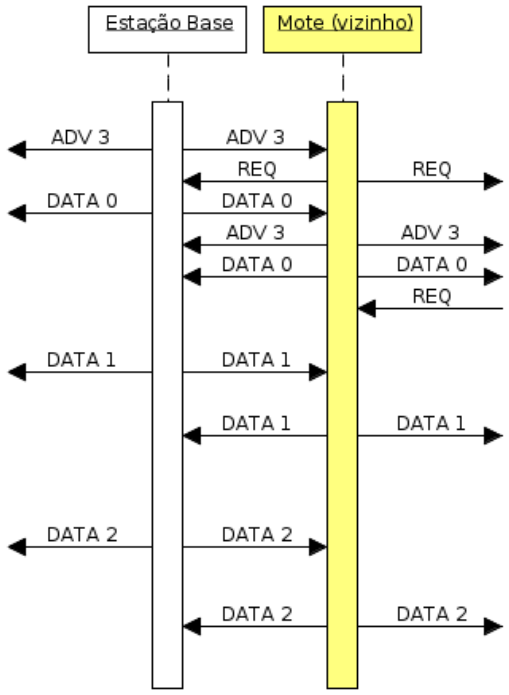


Figura 2.5: Diagrama de sequência do DisCoTerra

Após receber a primeira mensagem DATA da estação base, cada mote encaminha a mensagem ADV em broadcast exatamente da forma que recebeu da estação base, atingindo os motes subsequentes, indicando que existe um novo script. A mensagem só não é reenviada se já foi recebida anteriormente, ou seja, mensagens repetidas são descartadas. Todos os motes que recebem uma mensagem ADV pela primeira vez fazem o *reset* do mapa de bits que controlam quais mensagens já foram recebidas e enviam uma mensagem REQ broadcast e, somente depois de receberem a primeira mensagem DATA, encaminham a mensagem ADV, caso não seja repetida. Ao contrário do Deluge que envia várias mensagens ADV atualizando suas páginas, esta mensagem ADV do DisCoTerra somente é enviada uma vez por cada mote. E antes de cada mote

transmití-la, a mensagem REQ broadcast é enviada. Cada mote ao receber a primeira mensagem REQ envia as mensagens DATA consecutivamente. As mensagens REQ repetidas são descartadas. Ao receberem as mensagens DATA, os motes vão marcando os bits para terem um controle do que ainda não foi recebido e, quando a mensagem não estava marcada anteriormente, ela é transmitida para alcançar novos motes.

Os motes guardam a informação do id do mote que enviou a última mensagem DATA no momento que atualizam o bit para o seu controle, e caso esteja faltando alguma mensagem de número de sequência menor do que foi recebido, após enviar a mensagem DATA, envia uma nova mensagem REQ, dessa vez unicast apenas para o mote que enviou a última mensagem recebida solicitando a mensagem com número de sequência que faltou.

### 2.3.3 Considerações entre os protocolos

A diferença principal entre os dois protocolos é que o Deluge faz um controle maior na entrega das mensagens enquanto o DisCoTerra é mais otimista quanto a entregas das mensagens, uma vez que são aceitas mensagens recebidas antes mesmo de requisições terem sido feitas e também as recebidas de um mote diferente do que mandou a mensagem ADV.

O Deluge faz cálculos de CRC em cada uma das páginas e em cada um dos pacotes utiliza um preenchimento. Para disseminar 100 bytes de imagem, o Deluge cria um objeto com 528 bytes devido ao preenchimento e CRC. Este procedimento do Deluge causa uma sobrecarga muito grande em relação ao DisCoTerra. O DiscoTerra consegue disseminar 100 bytes com apenas 120 bytes, isto é, 5 mensagens de 24 bytes de payload. Essa sobrecarga não é tão significativa em relação aos tamanhos típicos de códigos disseminados pelo Deluge.

Tabela 2.1: Tamanho real a ser disseminado

| | **Deluge** | **DisCoTerra** |
|---|---|---|
| 100 bytes | 528 | 120 |
| 250 bytes | 792 | 264 |
| 500 bytes | 1056 | 504 |
| 1000 bytes | 1584 | 1008 |
| 2000 bytes | 2640 | 2016 |

O preenchimento e CRC no Deluge é estável, sempre aumentando o tamanho da imagem a ser transmitido em torno de 500 bytes, independentemente do tamanho da imagem. A tabela 2.1 mostra o tamanho do código e quanto efetivamente é disseminado para alguns casos.

# 3
# Ambiente de Execução de Testes

Neste estudo usamos o simulador Cooja (Eriksson et al., 2009) para verificar o comportamento dos dois algoritmos de disseminação com diferentes configurações. O Cooja é um simulador desenvolvido em Java e tem como principal diferencial ser modular e extensível. Permite que sejam realizadas alterações, inclusões ou substituições de plataformas de um mote, transceptores de rádio e modelos de transmissão de mensagem. Além disso, o Cooja permite simular ambientes heterogêneos, ou seja, ele pode comportar em uma mesma rede tanto motes de diferentes plataformas de hardware quanto motes com sistemas embarcados distintos. Essas características fazem do Cooja um simulador consideravelmente flexível.

Escolhemos o Cooja justamente por ser um simulador bastante flexível que pode ser usado tanto na parte gráfica, mostrando as mensagens sendo trocados entre os nós, o mapa de localização dos nós, o desenho do alcance do rádio e a linha do tempo de tudo que está acontecendo, quanto sendo executado por linha de comando, escondendo a parte gráfica mas continuando a produzir todas as informações necessárias durante a simulação para serem capturadas.

Nesse projeto adotamos o mote TelosB (Polastre et al., 2005). Quando o simulador é iniciado, a ordem e o momento em que os motes são iniciados é pseudo randômica e obedece a uma semente para a geração da sequência aleatória. Utilizamos sempre a mesma semente, portanto a ordem e o momento dos nós ligarem é sempre o mesmo. Não simulamos o MicaZ, além do TelosB, pois o rádio de ambos é o mesmo (CC2420) e no Cooja o desenvolvimento do MicaZ foi descontinuado, uma que seu tamanho de memória flash é bem inferior ao do TelosB.

Para que a simulação possa ser replicada no Cooja basta que a semente utilizada seja a mesma. Dessa forma, criamos dentro do Cooja o nó do mesmo tipo usado na compilação. Também escolhemos a quantidade desses nós e a posição de cada um deles. A flexibilidade do Cooja permite tanto simular diferentes aplicações no mesmo hardware como simular a mesma aplicação mas em diferentes tipos de motes na mesma rede. Neste trabalho simulamos os dois algoritmos de disseminação separadamente, ou seja, todos os motes da rede estavam com o mesma aplicação, e somente utilizamos como hardware o TelosB.

Existem alguns plugins embutidos no Cooja que interagem para a captura

de informações adicionais não fornecidas pelo simulador. Utilizamos o MSPSIM (Eriksson et al., 2009) para termos as informações sobre uso de CPU e uso do rádio dos motes TelosB a cada segundo e também usamos o Serial Socket. Este último permite que as informações da serial de cada mote sejam direcionadas para uma porta TCP da máquina onde está sendo executado o Cooja. Nós fizemos uso desse recurso através de um software já disponível dentro do TinyOS chamado PrintfClient para capturar as mensagens que imprimimos dentro da aplicação. Pela conexão na porta TCP que foi aberta pelo Cooja para cada mote, as informações da porta serial de cada um dos motes foi direcionada a um arquivo texto e tratada posteriormente por um script Lua.

Para gerar o relatório final com as informações de consumo de energia e tempo total da disseminação, criamos um script Lua que lê os arquivos gerados pelo Cooja e mostra de forma compacta todas as informações desejadas.

Para garantir a sincronização global dos eventos ocorridos em cada um dos motes, utilizamos o relógio do simulador como global e adicionamos em todos os tempos de cada mote o offset do tempo que ele demorou para ligar. Uma vez que o simulador repete a ordem e o momento que cada um dos nós será ligado para uma mesma semente, essa manobra nos permitiu garantir a sincronização de cada um dos eventos para a geração correta do relatório.

Criamos um teste automatizado que consiste em diversos scripts Shell para executar todos os passos citados anteriormente sem precisarmos intervir em nada.

Todas as ações que os scripts fazem serão detalhadas a seguir:

- Verificar se os parâmetros passados são válidos.
- Compilar os arquivos da aplicação passando por parâmetros os valores de entrada.
- Copiar o executável para o local correspondente de acordo com o cenário.
- Alterar o script de execução automática do Cooja dentro do arquivo de configuração do simulador.
- Executar o simulador Cooja utilizando o arquivo de configuração já modificado.
- Verificar se as portas TCP de cada um dos nós está aberta.
- Conectar nas portas TCP dos respectivos nós e salvar as informações em um arquivo para cada nó.
- Somar o offset em cada um dos arquivos em cada uma das linhas para sincronizar o valor o tempo dos eventos.
- Certificar que a simulação já terminou antes de executar o relatório.

– Juntar todos os arquivos em um arquivo global que servirá de entrada para o script Lua.
– Executar o script Lua para gerar o relatório.

Para maiores detalhes mostramos o script Lua que gera o relatório no Apêndice A.

# 4
# Análise Experimental

Neste capítulo relatamos os resultados dos experimentos que realizamos. Definimos todos os parâmetros utilizados pelos experimentos como o tempo do rádio ligado, os duty cycling utilizados e o tamanho das mensagens contendo os dados que foram disseminados. Mostramos um comparativo entre os algoritmos de disseminação e descrevemos os dois cenários utilizados dentro dos experimentos.

Os experimentos foram executados dentro do ambiente de simulação Cooja e as aplicações com os algoritmos de disseminação foram compilados no TinyOS, uma vez que o DisCoTerra foi feito utilizando o TinyOS e o Deluge também é utilizado dentro do TinyOS.



## 4.1
## Definição do Tempo do rádio ligado

Em todos os experimentos utilizamos o mesmo valor de tempo do rádio ligado. Exceto quando o rádio fica ligado o tempo todo, que chamaremos a partir desse momento de DC 100%. Apesar do nome DC, a técnica de duty cycling não é utilizada nesse caso.

Após testes preliminares em uma topologia sequencial, variamos o tempo do rádio ligado entre 5 e 15 ms. Disseminando sempre 500 bytes e mantendo o DC em 10%.

A topologia sequencial (Figura 4.1) utilizada consiste de 10 motes lineares numerados de dois a onze mais a estação base (nó um) onde cada mote só tem como vizinho os motes imediatamente a frente e à trás dele.

Com 11 ms obtivemos o menor tempo da disseminação entre as simulações realizadas. Provavelmente esse resultado foi determinado em virtude de outros parâmetros padrões do algoritmo de disseminação utilizado.

## 4.2
## Valores de Duty Cycling (DC) escolhidos para os experimentos

As simulações usando DC com valores de 1% e 5% tiveram resultados piores que o DC 100%. No caso do DC de 1% o tempo de disseminação demorou minutos para transmitir centenas de bytes. Decidimos por não incluir sua análise nesse estudo uma vez que seu uso na prática é não trás nenhuma vantagem.

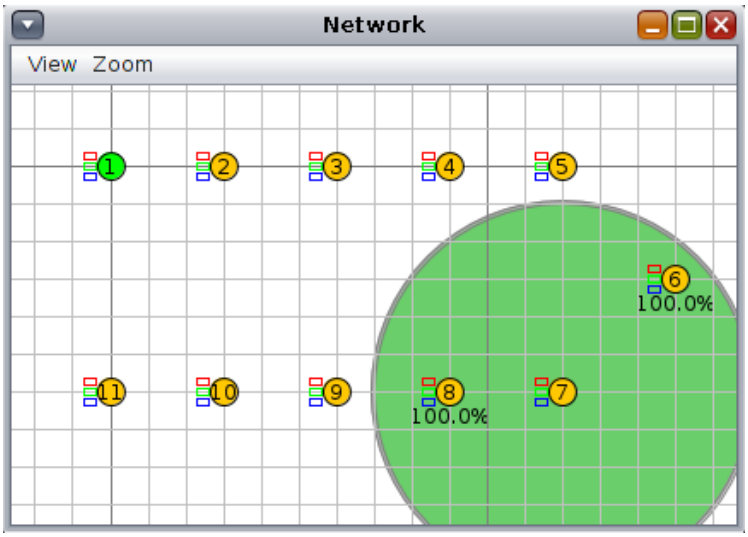


Figura 4.1: Topologia sequencial

Com a utilização do DC de 5% o aumento esperado da latência quando usamos duty cyling foi tão grande a ponto que o consumo de energia ter sido superior ao do DC 100%. Por essa razão, decidimos não mostrar qualquer análise desse DC. Para os nossos experimentos utilizamos os DC de 10%, 15%, 20% e 100% nos em todos os cenários.

## 4.3 Tamanho das mensagens

O protocolo de comunicação do rádio existente no TinyOS é o Active Message (AM). O tamanho padrão das mensagens no AM é 28 bytes. O payload do DisCoTerra é 24 bytes. A parte de controle usa 4 bytes. O Deluge utiliza 6 bytes para controle, sobrando 22 bytes para o payload. Portanto em um programa de 1000 bytes, por exemplo, o Deluge transmite 72 mensagens de dados. Esse número de mensagens é decorrente do tamanho escolhido para página, que foi de 264 bytes (12 pacotes por página). O DisCoTerra transmite apenas 42 mensagens de dados para os mesmos 1000 bytes. O número de mensagens transmitidas na disseminação será acrescentado das mensagens de advertisement e mensagens de solicitação dos dados.

Esse valor de 264 bytes para o tamanho da página foi o menor valor conseguido para diminuir o a quantidade de bytes do preenchimento usado no Deluge. Fizemos vários testes para conseguir diminuir o número de bytes desnecessários a ser disseminado sem prejudicar o funcionamento do Deluge no cálculo de CRC da página. O Deluge por padrão utiliza 1024 bytes para o tamanho de página, logo tem 47 pacotes por página. Se mantivéssemos esses valores, a sobrecarga do Deluge aumentaria ainda mais quantidade de bytes

para a disseminação de códigos de tamanhos até 2000 bytes.

## 4.4 Cálculo da energia

O nosso cálculo da energia consumida, reproduzida na Tabela 4.1, é baseado na amperagem obtida do datasheet do TelosB (Polastre et al., 2005). A voltagem é de 3 V no TelosB. Para calcularmos a energia em miliJoules (mJ) utilizamos a equação E = P x T (1), onde P é a potência (em mW) e T é o tempo (em s). E para calcularmos a potência em miliWatts (mW) utilizamos a equação P = V x A (2), onde V é a voltagem (em V) e A é a amperagem (em mA). Substituindo o P da equação 2 na equação 1 temos a equação do consumo de energia dada por E = V x A x T.

Tabela 4.1: Valores da amperagem (em mA) no TelosB

| | |
|---|---|
| CPU Ativa | 1.800 |
| CPU Standby | 0.0545 |
| Radio Idle | 0.365 |
| Radio RX | 20.0 |
| Radio TX | 17.7 |
| Radio Desligado | 0.020 |

## 4.5 Avaliação do uso do Deluge

Na Figura 4.2 temos o resultado das simulações dos algoritmos de disseminação Deluge e DisCoTerra na topologia sequencial para disseminar códigos de tamanho 100, 250, 500, 1000 e 2000 bytes.

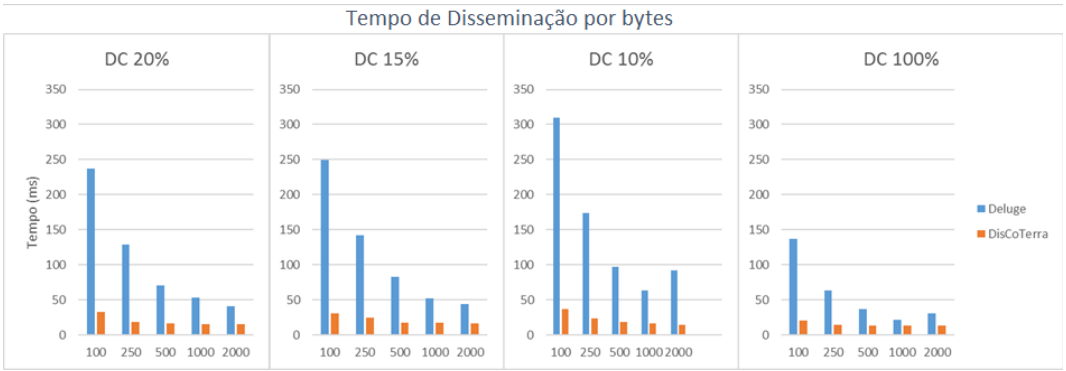


Figura 4.2: Tempos de disseminação por bytes do Deluge e do DisCoTerra.

O gráfico do tempo de disseminação por bytes se mantém homogêneo para todos os tamanhos de código no DisCoTerra enquanto que no Deluge o

valor do tempo da disseminação é muito alto para esses tamanhos de códigos menores e tende a diminuir com o aumento dos tamanhos de códigos. Isso ocorre por conta da sobrecarga do Deluge.

Em virtude do algoritmo Deluge ter obtido tempos de disseminação muito altos mesmo com a redução da sobrecarga padrão utilizada por seu algoritmo, não abordaremos as análises desse algoritmo. Os tempos de disseminação aumentaram em 70% com uso dos diversos duty cycling em relação a DC 100%. Além disso, para disseminar o mesmo tamanho do código em relação ao DisCoTerra, os tempos de disseminação foram entre 40% e 85% mais lentos. Portanto, esse algoritmo na prática não traz vantagens sobre o DisCoTerra. A partir desse momento todos os experimentos e análises descritos dizem respeito ao DisCoTerra.

## 4.6 Cenários dos experimentos

Realizamos dois cenários de experimentos. No cenário A temos a disseminação de códigos para cinco tamanhos de códigos distintos. Os tamanhos 100, 250, 500, 1000 e 2000 bytes. Esses tamanhos de códigos escolhidos foram motivados a partir de exemplos de aplicações para Terra (Branco, 2015).

Utilizamos quatro valores percentuais de DC. Fizemos experimentos com o DC 100% para usarmos de linha de base para comparação além dos DC de 10%, 15% e 20% do rádio. Realizamos os experimentos em três topologias: a topologia sequencial, a topologia aleatória (Figura 4.4) e a topologia grid (Figura 4.3).

Medimos no cenário A a soma da energia consumida por todos os motes da rede durante a disseminação, o tempo total da disseminação e o número total de mensagens transmitidas.

A topologia sequencial serviu para avaliar a disseminação sem colisões, uma vez que a comunicação entre vizinhos alcançáveis existente é somente entre o mote que transmite as mensagens de disseminação e mote que as recebe.

Na topologia grid os motes foram configurados no Cooja possuindo como vizinhos alcançáveis os motes acima, abaixo, à esquerda, à direita e nas suas quatro diagonais, totalizando 8 vizinhos. As exceções são os motes das bordas do grid que possuem cinco vizinhos, e os quatro motes das pontas do grid que possuem 3 vizinhos. Esta topologia representa uma RSSF densa.

Na topologia aleatória os motes foram posicionados de forma aleatória pelo Cooja. O número de vizinhos varia entre 1 e 6. A maioria dos motes desse cenário possui apenas 2 vizinhos. Esta topologia representa uma RSSF esparsa.

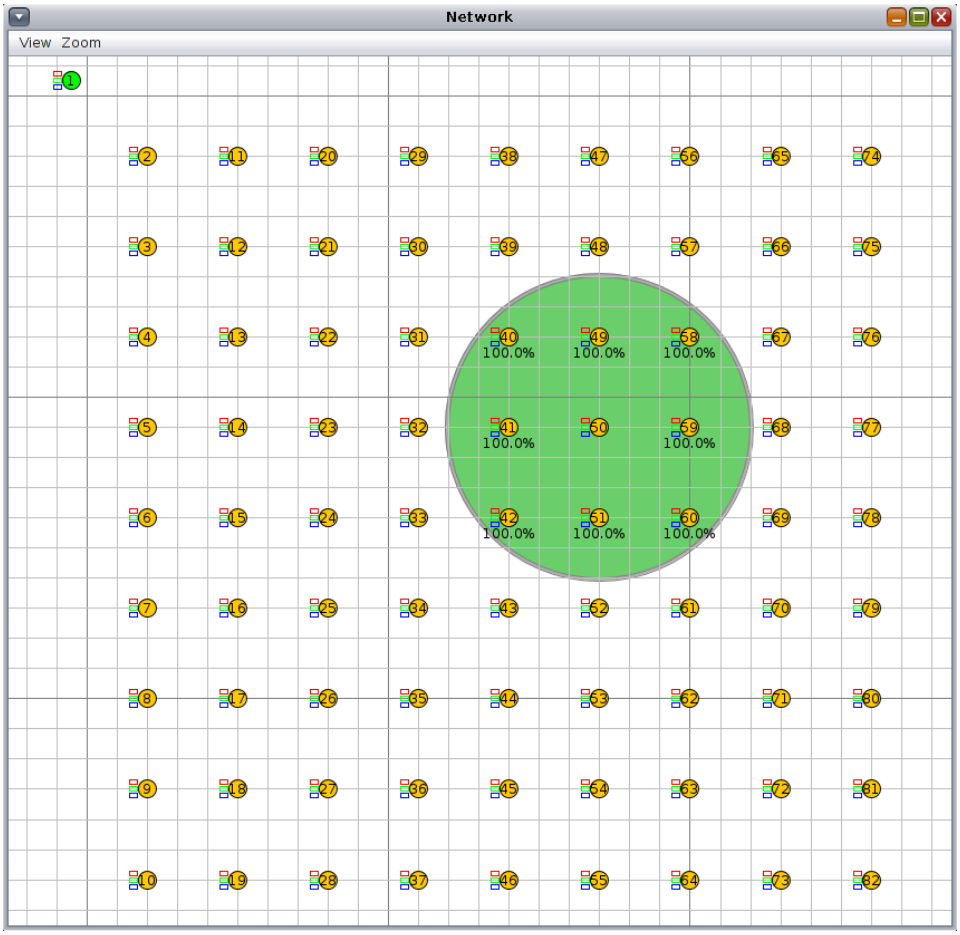


Figura 4.3: Topologia grid

Definimos como um salto o alcance do rádio de um mote aos seus vizinhos. Para uma mensagem ser transmitida a todos os nós da topologia sequencial, por exemplo, a mensagem vai dar dez saltos, uma vez que temos onze motes com apenas 2 vizinhos cada um. Analisando a quantidade de saltos nas topologias das Figuras 4.3 e 4.4 identificamos que a topologia aleatória ficou quatro vezes maior que o número de saltos da topologia grid.

Um segundo cenário de experimentos foi realizado, chamado de Cenário B. Nesse deixamos fixo o tamanho de código de 500 bytes que será disseminado. Novamente utilizamos os mesmos valores de DC do cenário A e as três topologias são grids variando os número de nós que são compostos por elas. Além do grid 9x9 do cenário A, utilizamos o grid 7x7 e o grid 5x5.

Escolhemos 500 bytes para o tamanho fixo do código da disseminação por se tratar de um valor médio do cenário A e ser um valor de tamanho comum em script Céu para o ambiente Terra.

Medimos as mesmas informações do cenário A e o valor médio de consumo de energia por cada nó no cenário B.

Todos os cenários de experimentos analisados no Capítulo **??** foram

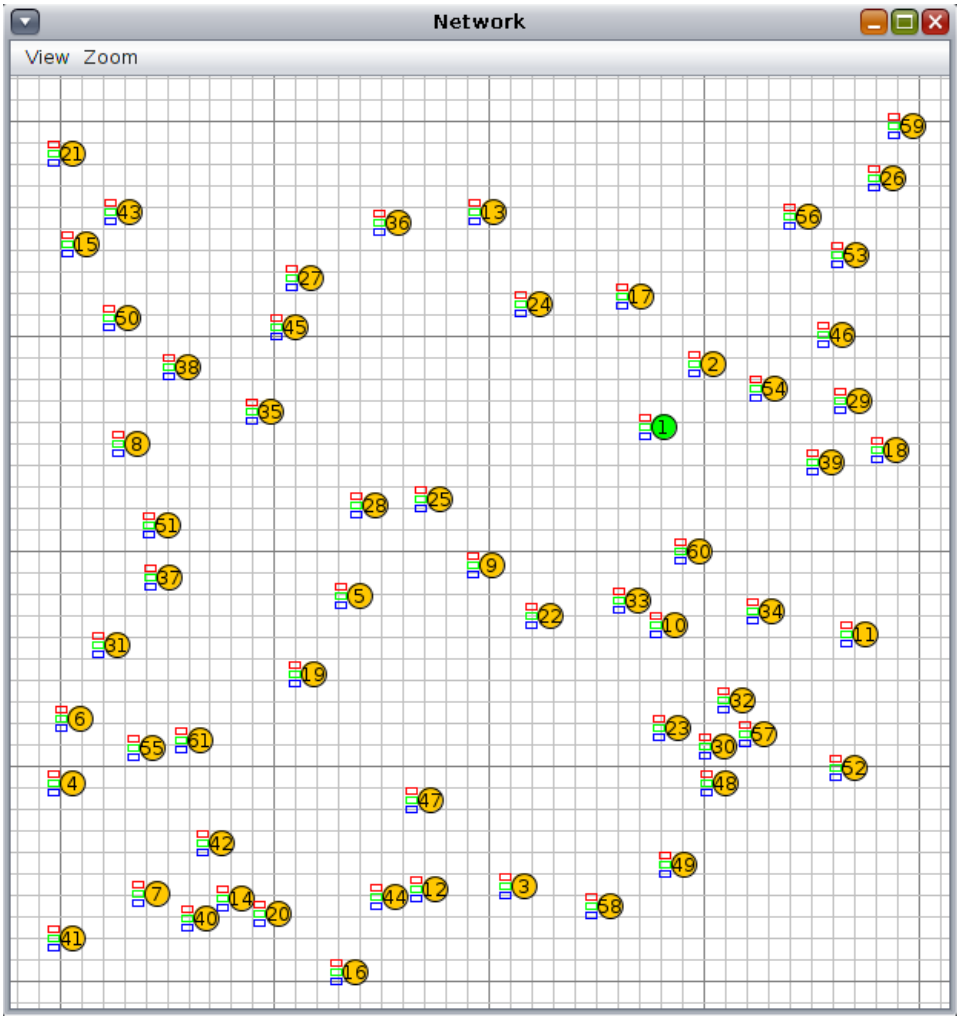


Figura 4.4: Topologia aleatória

executados cinco vezes e o resultado exibido foi a média desses valores ou a soma das médias.

## 4.7 Variações nos Resultados

As Figuras 4.5 e 4.6 mostram o tempo de disseminação de tamanhos de código de 500 e 1000 bytes, respectivamente, para as três topologias do cenário A utilizando os quatro valores de DC.

Nas duas figuras, os pontos verdes e vermelhos representam a média do tempo de disseminação das cinco simulações executadas e os traços pretos correspondem os valores máximo e mínimo das cinco simulações.

Podemos notar nesses dois exemplos que, apesar dos valores serem médias de cinco simulações, as variações em alguns casos ficaram grandes. Essa variação ocorreu mais na topologia aleatória e para os maiores tamanhos de códigos. Em todas as simulações para DC 100%, a média é praticamente igual

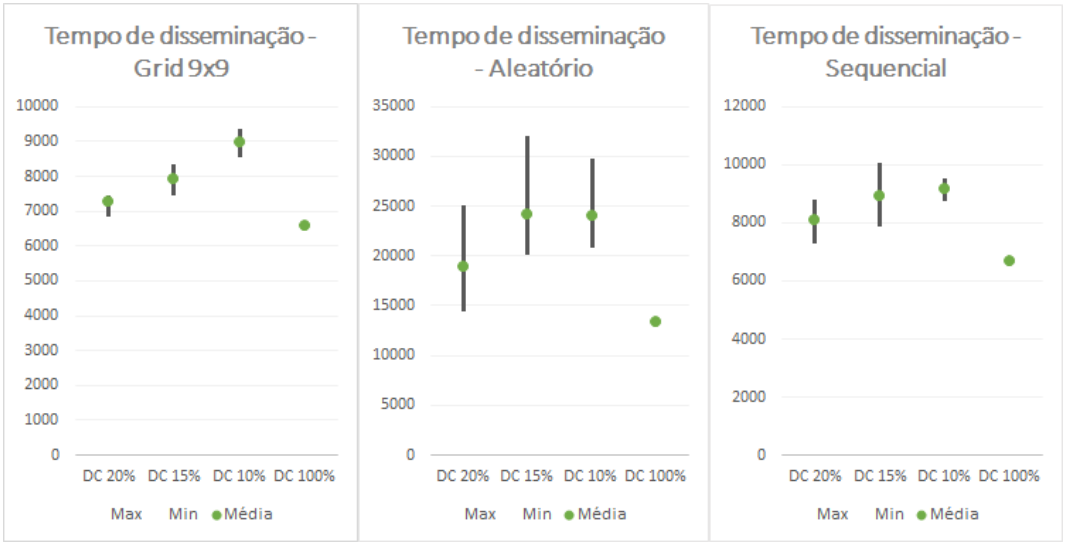


Figura 4.5: Tempo de disseminação de 500 bytes no cenário A

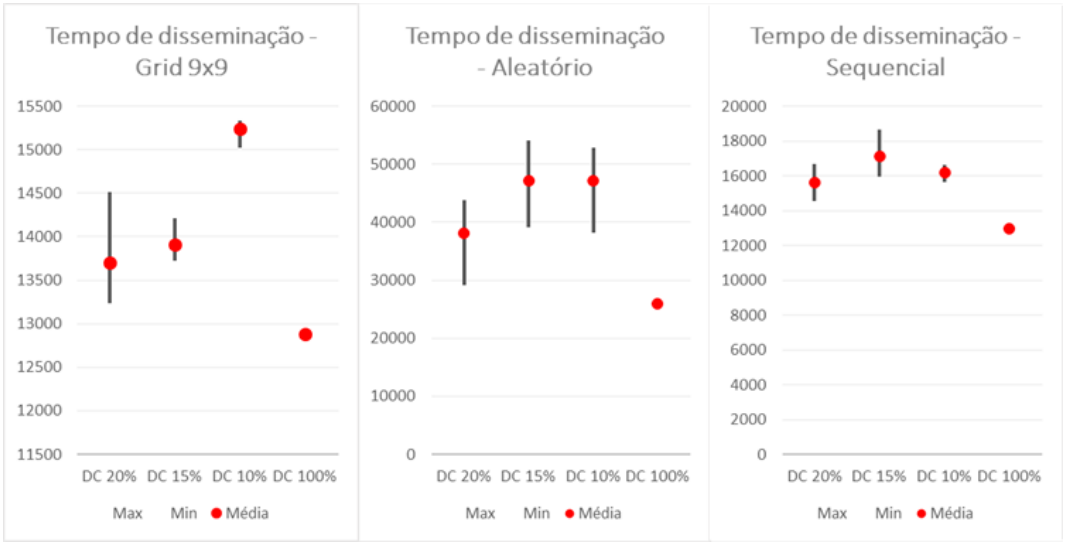


Figura 4.6: Tempo de disseminação de 1000 bytes no cenário A

aos valores máximos e mínimos.

O tempo total de disseminação na rede é diretamente proporcional à quantidade de saltos. Podemos verificar nos valores (máximos e mínimos) das Figuras 4.5 e 4.6 que a maior variação foi na topologia aleatória. Por ter o maior número de saltos na topologia aleatória, existe uma probabilidade maior de ocorrer mais falhas na transmissão durante a disseminação de código. Como eventuais falhas provocam retransmissões, o tempo de disseminação pode ser bastante alterado. Portanto, na topologia aleatória acontece uma maior variação em relação as outras topologias.

## 4.8 Resultados do cenário A

No cenário A foram coletados 60 valores correspondentes às 300 simulações para as três topologias, com cinco tamanhos de código utilizando quatro tipos de DC. Separamos em subseções cada uma das 3 medições deste cenário.

### 4.8.1 Energia consumida

O valor da energia em cada uma das colunas da Figura 4.7 consiste na soma dos valores de todos os motes da rede de cada uma das topologias. Lembrando que o valor da energia consumida por cada mote mostrado nos gráficos é o valor médio observado nas cinco simulações realizadas.

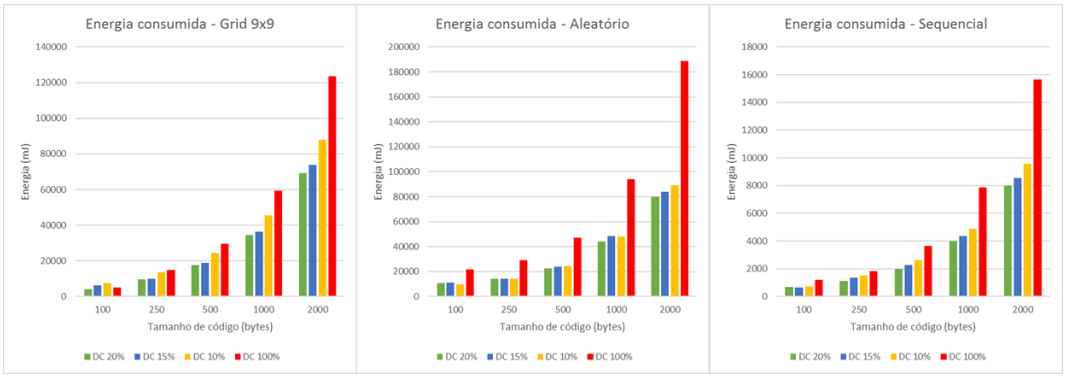


Figura 4.7: Energia consumida no cenário A

Podemos notar que a quantidade de energia consumida está diretamente ligada ao tamanho do código a ser disseminado. Isso ocorre em virtude de mais mensagens precisarem ser transmitidas para tamanhos maiores, e portanto, maior o tempo para terminar a disseminação.

Notamos que em todos as topologias o melhor resultado em relação a energia consumida por todos os nós da rede foi obtido pelo DC de 20% para todos os tamanhos de código. O uso de duty cycling só gerou mais consumo de energia que o DC 100% no caso da disseminação de 100 bytes na topologia grid para os DC de 15% e 20%. Acreditamos que a variação ocorrida nesse caso ocasionou esse resultado.

Os valores de energia consumida para DC de 10% e DC de 15% foram melhores para de DC de 100%, mas de forma geral foram piores que o DC de 20% em todas as topologias, para todos os tamanhos de códigos disseminados.

No DC de 20%, a redução de energia da rede em relação ao DC 100% teve como melhor resultado a redução de 58% da energia. Na topologia aleatória a redução foi pelo menos 50% para todos os tamanhos de código disseminados.

### 4.8.2 Tempo de disseminação

O valor do tempo de disseminação representado na Figura 4.8 é calculado seguindo a mesma explicação usada pelo consumo de energia.

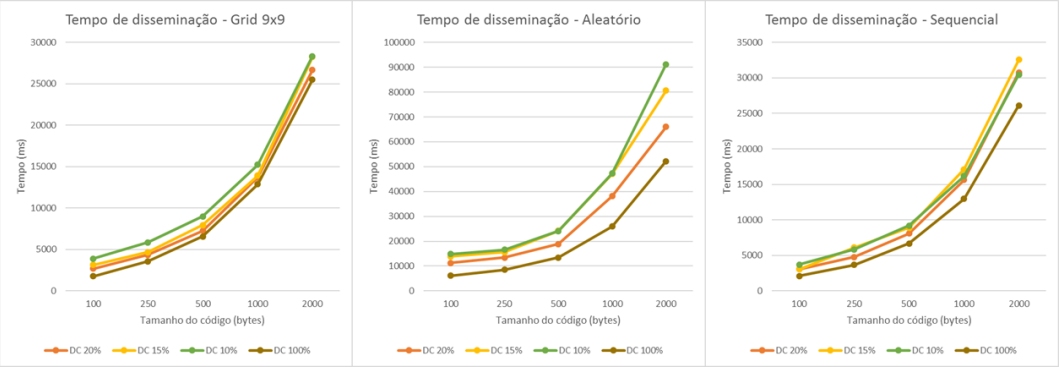


Figura 4.8: Tempo de disseminação no cenário A

Podemos notar claramente que como esperado, quanto maior o tamanho do código maior o tempo de disseminação na rede. A quantidade de saltos também está diretamente relacionada com o tempo de disseminação. Por isso que os tempos de disseminação na topologia aleatória foram os maiores e na topologia grid foram os menores.

Novamente, o DC de 20% foi o que resultou em menor aumento da latência em relação ao DC 100%. Os valores nas topologias grid e sequencial foram mais próximos para os valores de DC ficando um pouco mais afastados na topologia aleatória, mostrando mais claramente que os valores do DC de 20% foram sempre os mais próximos do DC 100%.

Quanto menor o tamanho do código maior foi o aumento da latência. Isso é provocado pelo funcionamento do DC que adiciona períodos de tempo de inatividade com o rádio desligado. Com poucas mensagens a ser transmitidas, os atrasos provocados pelo DC são mais facilmente percebidos. O menor aumento da latência do DC de 20% foi 5% em relação ao DC 100%.

### 4.8.3 Mensagens transmitidas

O número de mensagens transmitidas está mostrado na Figura 4.9. O uso da técnica de duty cycling para a disseminação de código aumenta muito

o número de mensagens transmitidas na rede em relação ao DC 100%. Como a disseminação de código utiliza mensagens broadcast, o acknowledgment que é enviado em mensagens unicast pelo mote depois que teve seu rádio ligado não existe. Por conta disso, a mesma mensagem é diversas vezes enviadas durante o tempo que o rádio está desligado.

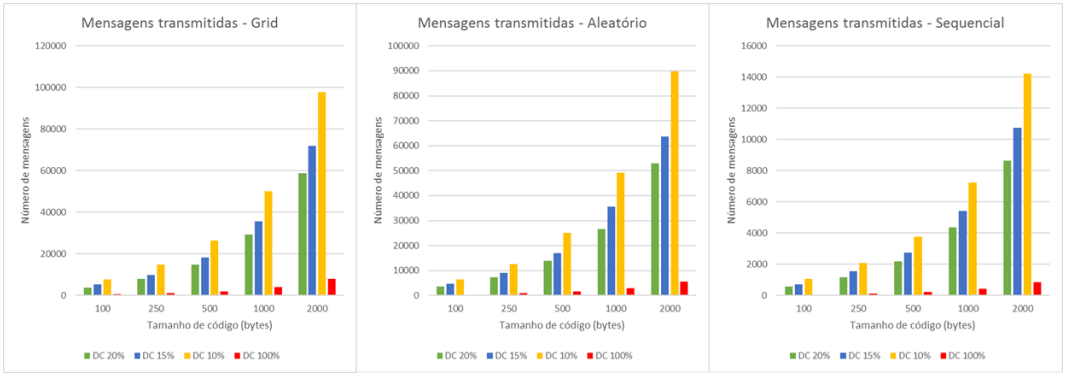


Figura 4.9: Mensagens transmitidas no cenário A

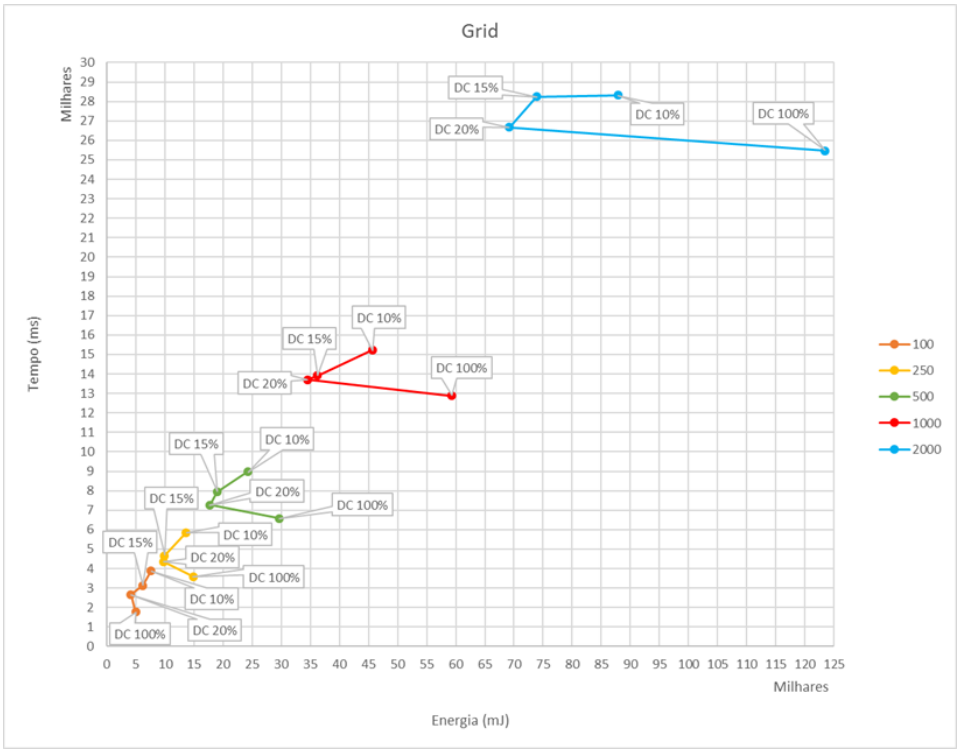


Figura 4.10: Relação do tempo de disseminação pelo consumo de energia na topologia grid

O DC de 20% teve o melhor desempenho novamente porque é o que possui menor valor absoluto de tempo de rádio desligado. O tempo de disseminação do DC de 20% foi em torno de 32,5% maior que o DC 100% no geral. Teve um

aumento em torno de 926% nas mensagens transmitidas na rede e a economia de energia na rede foi em média 44%.

Os testes comprovam que manter o rádio ligado é a principal causa do consumo de energia de um mote. Na Figura 4.10 temos o resumo do comportamento mostrado para a topologia grid. Cada cor representa um tamanho de código e os quatros pontos de cada uma dessas cores indicam os valores do tempo total de disseminação e energia consumida da rede para cada DC. No eixo horizontal temos a energia consumida por todos os motes da rede e no eixo vertical temos o tempo total de disseminação. O resultado obtido pelo DC 20% em todos os casos foi melhor.

## 4.9 Resultados do cenário B

O cenário B foi utilizado para ver como a variação do número de nós na mesma topologia afeta a disseminação do mesmo código. Neste cenário coletamos 12 valores correspondentes a 60 simulações na topologia grid com três quantidades de nós distintas (5x5, 7x7 e 9x9) e utilizando os quatro valores de DC disseminando 500 bytes de tamanho de código em todos os casos.

### Energia consumida

Na Figura 4.11 do lado esquerdo temos representado o valor da energia da rede. Em cada coluna representados o DC utilizado e cada cor indica qual topologia é correspondente.

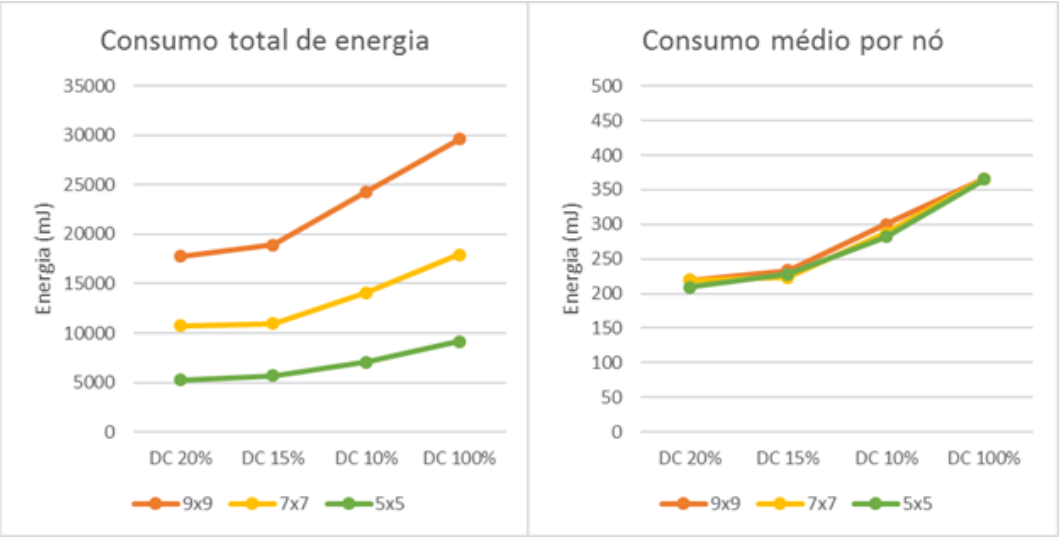


Figura 4.11: Energia consumida no cenário B

Conforme vimos no cenário A, quanto maior o número de saltos mais energia é consumida na rede durante a disseminação. A parte direita da Figura 4.11 representa o valor total pela quantidade de nós na rede. Percebemos que o consumo de energia médio é praticamente o mesmo para as três variações do número de nós na topologia grid.

Novamente o DC de 20% obteve o menor consumo de energia, da mesma forma que já havia acontecido no cenário A.

### 4.9.1 Tempo de disseminação

O valor do tempo de disseminação está representado do lado esquerdo da Figura 4.12. Os maiores valores são indicados pela topologia com maior número de saltos da mesma forma visto pelo cenário A.

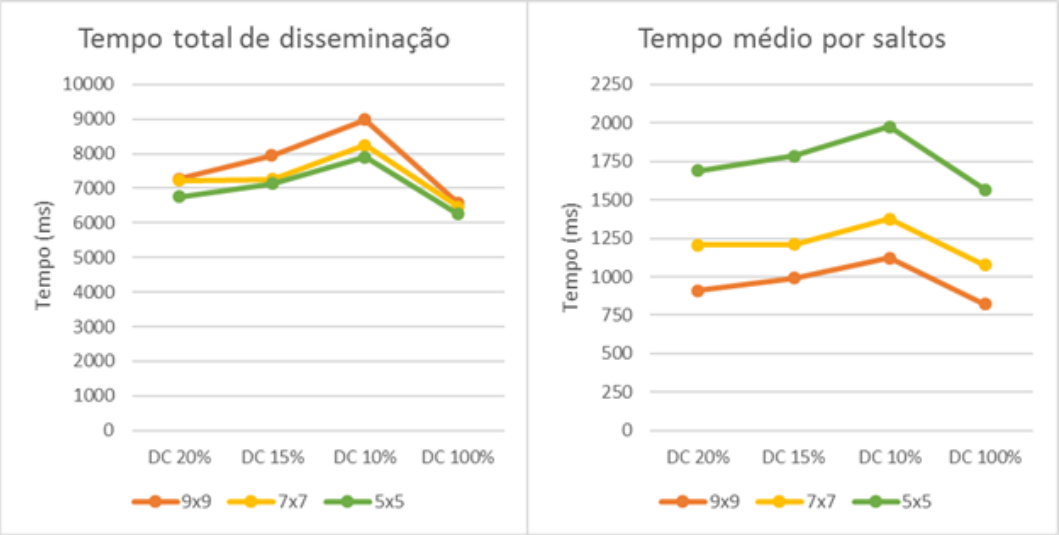


Figura 4.12: Tempo de disseminação no cenário B

No lado direito da Figura 4.12 está indicada a razão do tempo de disseminação pela quantidade de saltos representando o valor médio de disseminação por saltos. E os valores ficaram invertidos, ou seja, a topologia com maior número de saltos teve o menor valor de disseminação. Acreditamos que isso aconteça pois em uma rede densa com mais vizinhos, a chance de um nó vizinho iniciar a disseminação dentro de um intervalo de tempo é maior que nas redes onde o número de vizinhos é menor.

O DC de 20% foi mais rápido para todas as topologias em relação ao outros.

### 4.9.2 Mensagens transmitidas

O número de mensagens transmitidas está mostrado do lado esquerdo da Figura 4.13. No cenário A já fizemos a análise e o resultado no cenário B se mostrou o mesmo. Assim como aconteceu no cenário A o número de mensagens transmitidas teve seus menores valores no DC de 20% quando foi utilizado a técnica de duty-cycling.

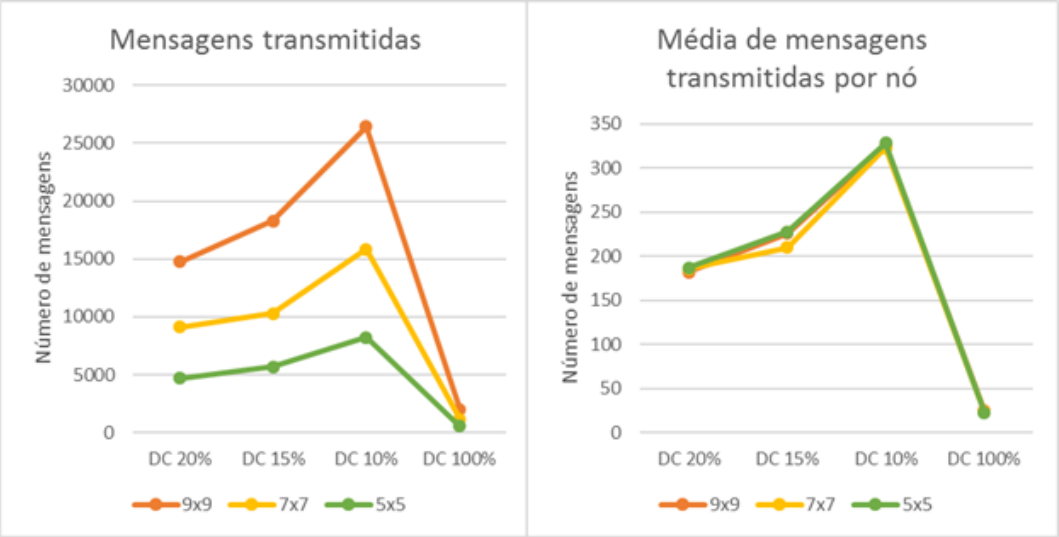


Figura 4.13: Mensagens transmitidas no cenário B

O lado direito da Figura 4.13 é analisado como o consumo de energia. Independente do tamanho da rede, o número médio de mensagens transmitidas é o mesmo. Novamente o uso do DC de 20% obteve os menores valores de mensagens transmitidas.

### 4.9.3 Valor médio de consumo por nó

Na figura 4.14 representamos o gradiente de consumo dos nós para os DC em todas as topologias. Os motes mais vermelhos representam os que consumiram mais energia e os motes mais verdes representam os que consumiram menos energia.

Podemos chegar a conclusão que os nós mais próximos da estação base são os que mais consomem energia em virtude da maior possibilidade de pertencerem ao caminho das mensagens de disseminação até os nós nós mais distantes. Os padrões de consumo para os grid 7x7 e 9x9 são muito parecidos para cada um dos DC. No grid 5x5 o tamanho da rede não ficou grande o

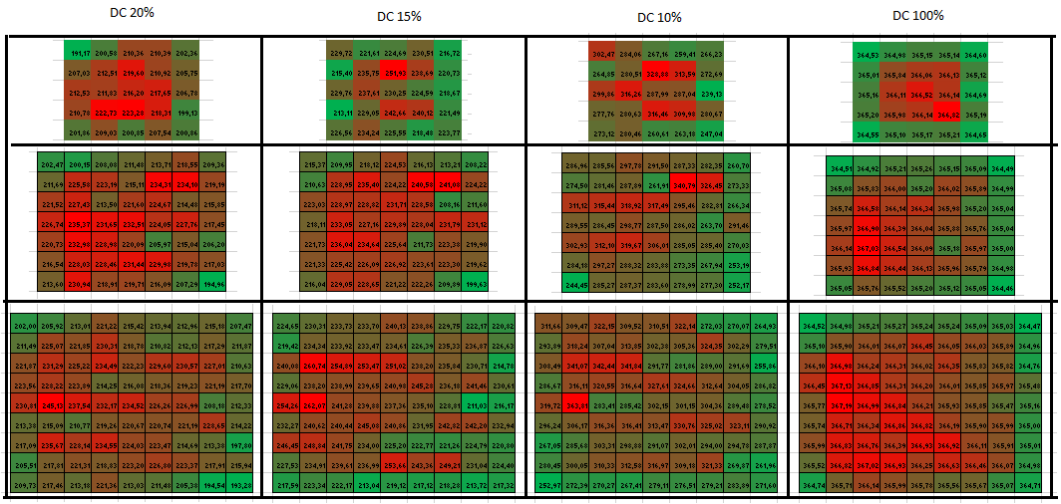


Figura 4.14: Gradiente de consumo de energia do cenário B

suficiente para diferenciar nós com muitos vizinhos, localizados no meio do grid, com nós poucos vizinhos, encontrados nas pontas do grid.

O padrão do consumo de energia do DC 100% se mostra diferente dos demais que usam duty cycling. Como o rádio se mantém ligado permanentemente, sem períodos de inatividade, o consumo nesse caso é bastante próximo entre todos os motes. Apesar das intensidades das cores estar diferente como nos outros casos, a diferença nos valores de consumo entre o nó que consumiu mais e o que consumiu menos é de 3 mJ. Enquanto que nos outros casos essa diferença é superior a 30 mJ, chegando em alguns casos a mais de 100 mJ.

# 5
# Conclusão

A principal contribuição deste trabalho foi entender melhor o tradeoff entre o tempo de disseminação e a energia gasta pelo motes. Em todos os cenários simulados houve um aumento no tempo de vida da rede para o uso de DC de 20% de forma unânime. O Tempo de disseminação aumentou apenas 10% e a energia consumida pelos motes teve uma redução de 40% nas três quantidades de nós avaliadas da topologia grid. E considerando todas as topologias avaliadas o aumento do tempo de disseminação foi menos de 30% e a redução da energia consumida em torno de 45%.

O trabalho conseguiu avaliar o impacto do Duty Cycling na latência da RSSF de maneira que a economia em toda a rede foi sempre superior a 35% em relação ao DC 100%. Os desenvolvedores de aplicações podem se valer desse levantamento para a tomada de decisões. Caso possa deixar que o atraso para disseminar o código possa ser um pouco maior, o tempo de vida da rede será aumentado utilizando o DC de 20%. O trabalho conseguiu fazer uma implementação funcional para a plataforma TelosB. Acredito que este estudo possa permitir o desenvolvimento do ambiente Terra para construção de códigos em RSSF contendo economia de energia permitindo que o ambiente se torne mais sofisticado que o funcionamento atual.

A principal conclusão entre configurar o rádio ligado o tempo todo e usar duty cycle é usar o DC de 20% como melhor opção de custo benefício para a relação entre tempo gasto para disseminar um código e energia consumida pelos motes.

Nas nossas análises, quanto maior o tamanho de código disseminado mais energia economizamos e menos o tempo de disseminação aumentou em relação ao DC 100%. Isso beneficia aplicações mais complexas do Terra onde o tamanho do script fica entre 1000 e 2000 bytes.

# 6
# Referências Bibliográficas


BRANCO, A. **Terra: Flexibility and safety in Wireless Sensor Networks**. Tese (Doutorado) — Pontifícia Universidade Católica do Rio de Janeiro – PUC-RIO, sep 2015. 4.6

BRANCO, A. et al. Terra: Flexibility and safety in wireless sensor networks. **ACM Transactions on Sensor Networks**, ACM, New York, NY, USA, 2015. ISSN 1550-4859. In press. 1

BROWN, S.; SREENAN, C. J. Software updating in wireless sensor networks: A survey and lacunae. **Journal of Sensor and Actuator Networks**, v. 2, n. 4, p. 717, 2013. ISSN 2224-2708. 2.3

BUETTNER, M. et al. X-mac: A short preamble MAC protocol for duty-cycled wireless sensor networks. In: **Proceedings of the 4th International Conference on Embedded Networked Sensor Systems**. New York, NY, USA: ACM, 2006. (SenSys '06), p. 307–320. ISBN 1-59593-343-3. 2.2

ERIKSSON, J. et al. Cooja/mspsim: Interoperability testing for wireless sensor networks. In: **Proceedings of the 2Nd International Conference on Simulation Tools and Techniques**. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009. (Simutools '09), p. 27:1–27:7. ISBN 978-963-9799-45-5. 3

GAY, D. et al. The nesc language: A holistic approach to networked embedded systems. **SIGPLAN Notices**, ACM, New York, NY, USA, v. 38, n. 5, p. 1–11, maio 2003. ISSN 0362-1340. 2.2

HUI, J. W.; CULLER, D. The dynamic behavior of a data dissemination protocol for network programming at scale. In: **Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems**. New York, NY, USA: ACM, 2004. (SenSys '04), p. 81–94. ISBN 1-58113-879-2. 1, 2.3

KULIK, J.; HEINZELMAN, W.; BALAKRISHNAN, H. Negotiation-based protocols for disseminating information in wireless sensor networks. **Wireless Networks**, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 8, n. 2/3, p. 169–185, mar. 2002. ISSN 1022-0038. 1

LEVIS, P. et al. Tinyos: An operating system for sensor networks. In: **Ambient Intelligence**. [S.l.]: Springer, 2005. p. 115–148. ISBN 978-3-540-27139-0. 2.2

LEVIS, P. et al. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In: **Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1**. Berkeley, CA, USA: USENIX Association, 2004. (NSDI'04), p. 2–2. 2.3.1

LU, G.; KRISHNAMACHARI, B.; RAGHAVENDRA, C. An adaptive energy-efficient and low-latency mac for data gathering in wireless sensor networks. In: **Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International**. [S.l.: s.n.], 2004. p. 224–. 1

MOSS, D.; LEVIS, P. **BoX-MACs: exploiting physical and link layer boundaries in low-power networking**. [S.l.], 2008. 2.2

POLASTRE, J.; HILL, J.; CULLER, D. Versatile low power media access for wireless sensor networks. In: **Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems**. New York, NY, USA: ACM, 2004. (SenSys '04), p. 95–107. ISBN 1-58113-879-2. 1, 2.2

POLASTRE, J.; SZEWCZYK, R.; CULLER, D. Telos: enabling ultra-low power wireless research. In: **Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on**. [S.l.: s.n.], 2005. p. 364–369. 3, 4.4

RAGHUNATHAN, V. et al. Energy-aware wireless microsensor networks. **Signal Processing Magazine, IEEE**, v. 19, n. 2, p. 40–50, Mar 2002. ISSN 1053-5888. 1

SANT'ANNA, F. et al. Safe system-level concurrency on resource-constrained nodes. In: **Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems**. New York, NY, USA: ACM, 2013. (SenSys '13), p. 11:1–11:14. ISBN 978-1-4503-2027-6. 1

TALZI, I. et al. Permasense: Investigating permafrost with a wsn in the swiss alps. In: **Proceedings of the 4th Workshop on Embedded Networked Sensors**. New York, NY, USA: ACM, 2007. (EmNets '07), p. 8–12. ISBN 978-1-59593-694-3. 1

YE, W.; HEIDEMANN, J.; ESTRIN, D. An energy-efficient mac protocol for wireless sensor networks. In: **INFOCOM 2002. Twenty-First Annual Joint Confe-**

**rence of the IEEE Computer and Communications Societies. Proceedings. IEEE**. [S.l.: s.n.], 2002. v. 3, p. 1567–1576 vol.3. ISSN 0743-166X. 2.2

# A
# Relatório

O relatório obtido após cada execução de um experimento no Cooja é gerado por um script Lua. Abaixo temos o código fonte desse script.

Listing A.1: Script Lua que gera o relatório

```
1  if(#arg ~= 3) then
2      print("Usage: lua " .. arg[0] ..
3              " <case:1..12> <nodes> <blocks>")
4      os.exit(1)
5  end
6
7  file1 = io.open("../tests/T" .. arg[1] ..
8          "/results/data_log_final.txt", "r")
9
10 record = {}
11 energy = {}
12 Tfunc = {}
13 stats = {}
14 maxNodes = tonumber(arg[2])
15 maxBlocks = tonumber(arg[3])
16 volts = 3
17 refVI = {}
18 refVI.cpuActive = volts * 1.800
19 refVI.cpuIdle   = volts * 0.0545
20 refVI.radioIdle = volts * 0.365
21 refVI.radioRX   = volts * 20.0
22 refVI.radioTX   = volts * 17.7
23 refVI.radioOff  = volts * 0.020
24
25 stats = {}
26 stats.first = 0
27 stats.last  = 0
28 for j = 1, maxNodes do
29 stats[j] = {}
30 stats[j].start  = 0
31 stats[j].stop   = 0
32 stats[j].pkgsS  = 0
33 stats[j].pkgsR  = 0
34 stats[j].energy = 0
35 stats[j].complete = "N"
36 stats[j].energCpuActive = 0
37 stats[j].energCpuIdle   = 0
38 stats[j].energRadioIdle = 0
```

```
39 stats[j].energRadioRX   = 0
40 stats[j].energRadioTX   = 0
41 stats[j].energRadioOff  = 0
42 stats[j].cpuActive = 0
43 stats[j].cpuIdle   = 0
44 stats[j].radioIdle = 0
45 stats[j].radioRX   = 0
46 stats[j].radioTX   = 0
47 stats[j].radioOff  = 0
48 end
49
50 Tfunc2 = {
51     BOOT = function(data)
52     end,
53     INIT = function(data)
54         if (data.time >= stats.first and
55                 data.time <= stats.last) then
56             stats[data.id].pkgsS =
57                 stats[data.id].pkgsS + 1
58         end
59     end,
60     NOTF = function(data)
61         if (data.time >= stats.first and
62                 data.time <= stats.last) then
63             stats[data.id].pkgsR =
64                 stats[data.id].pkgsR + 1
65         end
66     end,
67     STOP = function(data)
68     end,
69     SEND = function(data)
70         if (data.time >= stats.first and
71                 data.time <= stats.last) then
72             stats[data.id].pkgsS =
73                 stats[data.id].pkgsS + 1
74         end
75     end,
76     RECV = function(data)
77         if (data.time >= stats.first and
78                 data.time <= stats.last) then
79             stats[data.id].pkgsR =
80                 stats[data.id].pkgsR + 1
81         end
82     end,
83     REQT = function(data)
84         if (data.time >= stats.first and
85                 data.time <= stats.last) then
86             stats[data.id].pkgsS =
87                 stats[data.id].pkgsS + 1
```

```
88          end
89      end,
90      REPL = function(data)
91          if (data.time >= stats.first and
92                  data.time <= stats.last) then
93              stats[data.id].pkgsR =
94                      stats[data.id].pkgsR + 1
95          end
96      end,
97  }
98
99  Tfunc = {
100     BOOT = function(data)
101     end,
102     INIT = function(data)
103         if (stats[data.id].start == 0) then
104             stats[data.id].start = data.time
105         elseif (data.time < stats[data.id].start) then
106             stats[data.id].start = data.time
107         end
108         if (stats.first == 0) then
109             stats.first = data.time
110         elseif (data.time < stats.first) then
111             stats.first = data.time
112         end
113     end,
114     NOTF = function(data)
115         if (stats[data.id].start == 0) then
116             stats[data.id].start = data.time
117         elseif (data.time < stats[data.id].start) then
118             stats[data.id].start = data.time
119         end
120     end,
121     STOP = function(data)
122         if (stats[data.id].stop == 0) then
123             stats[data.id].stop = data.time
124         elseif (data.time > stats[data.id].stop) then
125             stats[data.id].stop = data.time
126         end
127         if (stats.last == 0) then
128             stats.last = data.time
129         elseif (data.time > stats.last) then
130             stats.last = data.time
131         end
132         stats[data.id].complete = "S"
133     end,
134     SEND = function(data)
135     end,
136     RECV = function(data)
```

```
137     end,
138     REQT = function(data)
139     end,
140     REPL = function(data)
141     end,
142 }
143
144 function split(s, delimiter)
145     result = {};
146     for match in (s..delimiter):gmatch(
147             "(.-)"..delimiter) do
148         table.insert(result, match);
149     end
150     return result;
151 end
152
153 function printReport()
154     print(arg[1], maxBlocks)
155     local total = 0
156
157     total = stats.last - stats.first
158
159     print(total)
160     print("Node\tOK?\tTime\t#SEND\t#RECV\tIDLE" ..
161             "\tTX\tRX\tOFF\tCPU\tSTANDBY\tENERGY")
162     for j = 1, maxNodes do
163
164         time = stats[j].stop - stats[j].start
165         if (j == 1) then
166             time = "-"
167             stats[j].complete = "-"
168         end
169
170         stats[j].energCpuActive =
171                 stats[j].cpuActive * refVI.cpuActive
172         stats[j].energCpuIdle   =
173                 stats[j].cpuIdle   * refVI.cpuIdle
174         stats[j].energRadioIdle =
175                 stats[j].radioIdle * refVI.radioIdle
176         stats[j].energRadioRX   =
177                 stats[j].radioRX   * refVI.radioRX
178         stats[j].energRadioTX   =
179                 stats[j].radioTX   * refVI.radioTX
180         stats[j].energRadioOff  =
181                 stats[j].radioOff  * refVI.radioOff
182         stats[j].energy = stats[j].energCpuActive +
183                 stats[j].energCpuIdle +
184                 stats[j].energRadioIdle +
185                 stats[j].energRadioRX +
```

```
186                 stats[j].energRadioTX +
187                 stats[j].energRadioOff
188
189         print(j, stats[j].complete, time,
190             stats[j].pkgsS, stats[j].pkgsR,
191         string.format("%.2f", stats[j].energRadioIdle),
192         string.format("%.2f", stats[j].energRadioTX),
193         string.format("%.2f", stats[j].energRadioRX),
194         string.format("%.2f", stats[j].energRadioOff),
195         string.format("%.2f", stats[j].energCpuActive),
196         string.format("%.2f", stats[j].energCpuIdle),
197         string.format("%.2f", stats[j].energy))
198     end
199 end
200
201 io.input(file1)
202 while true do
203     local line = io.read()
204     if (line == nil) then
205         break
206     end
207     record = split(line, " ")
208     record.id     = tonumber(record[1])
209     record.time   = tonumber(record[2])
210     record.obj    = tonumber(record[3])
211     record.page   = tonumber(record[4])
212     record.pkg    = tonumber(record[5])
213     record.round  = tonumber(record[6])
214     record.action = record[7]
215     Tfunc[record.action](record)
216 end
217 io.close(file1)
218
219 file1 = io.open("../tests/T" .. arg[1] ..
220 "/results/data_log_final.txt", "r")
221
222 io.input(file1)
223 while true do
224     local line = io.read()
225     if (line == nil) then
226         break
227     end
228     record = split(line, " ")
229     record.id     = tonumber(record[1])
230     record.time   = tonumber(record[2])
231     record.obj    = tonumber(record[3])
232     record.page   = tonumber(record[4])
233     record.pkg    = tonumber(record[5])
234     record.round  = tonumber(record[6])
```

```
235     record.action = record[7]
236     Tfunc2[record.action](record)
237 end
238 io.close(file1)
239
240 for k = 2, maxNodes, 1 do
241 power = {}
242 simTime = nil
243 energy[k] = {}
244 energy[k].cpuActive = 0
245 energy[k].cpuIdle   = 0
246 energy[k].radioIdle = 0
247 energy[k].radioRX   = 0
248 energy[k].radioTX   = 0
249 energy[k].radioOff  = 0
250 if (k < 10) then
251     file2 = io.open("../tests/T" .. arg[1] ..
252             "/results/mspcli0" .. k .. ".log", "r")
253 else
254     file2 = io.open("../tests/T" .. arg[1] ..
255             "/results/mspcli" .. k .. ".log", "r")
256 end
257 io.input(file2)
258
259 while true do
260     line = io.read()
261     if (line == nil) then
262         break
263     end
264     if (#line > 0) then
265         power = split(line, " ")
266         if (power[1] ~= "Emulated") then
267             energy[k].cpuActive =
268                     tonumber(power[1]) / 100
269             energy[k].cpuIdle   =
270                     tonumber(power[3]) / 100
271             energy[k].radioIdle =
272                     tonumber(power[7]) / 100
273             energy[k].radioRX   =
274                     tonumber(power[8]) / 100
275             energy[k].radioTX   =
276                     tonumber(power[9]) / 100
277             energy[k].radioOff  =
278                     tonumber(power[10]) / 100
279         else
280             local time = split(power[4], "%(")
281             simTime = tonumber(time[1])
282             if (simTime >= stats.first and
283                     simTime <= stats.last) then
```

```
284                     stats[k].cpuActive =
285                             energy[k].cpuActive +
286                             stats[k].cpuActive
287                     stats[k].cpuIdle   =
288                             energy[k].cpuIdle   +
289                             stats[k].cpuIdle
290                     stats[k].radioIdle =
291                             energy[k].radioIdle +
292                             stats[k].radioIdle
293                     stats[k].radioRX   =
294                             energy[k].radioRX   +
295                             stats[k].radioRX
296                     stats[k].radioTX   =
297                             energy[k].radioTX   +
298                             stats[k].radioTX
299                     stats[k].radioOff  =
300                             energy[k].radioOff  +
301                             stats[k].radioOff
302                 end
303             end
304         end
305 end
306         io.close(file2)
307 end
308
309 printReport()
```