Research, Society and Development, v.10, n. 1, e4010111453, 2021
(CC BY 4.0) | ISSN 2525-3409 | DOI: http://dx.doi.org/10.33448/rsd-v10i1.11453

Implementação e avaliação de desempenho de um detector de defeitos não-confiável utilizando o emulador CORE (COmmon Research Emulator)

Development and evaluate an unreliable asynchronous failure detector based on gossip using the CORE (COmmon Research Emulator)

Implementación y evaluación del rendimiento de un detector de defectos no confiable utilizando el emulador CORE

Recebido: 23/12/2020 | Revisado: 25/12/2020 | Aceito: 28/12/2020 | Publicado: 02/01/2021

Antônio Rodrigo Delepiane de Vit
ORCID: https://orcid.org/ 0000-0002-9452-0108
Universidade Federal de Santa Maria, Brasil
E-mail: rodrigodevit@inf.ufsm.br
Sidnei Renato Silveira
ORCID: https://orcid.org/ 0000-0002-4506-8522
Universidade Federal de Santa Maria, Brasil
E-mail: sidneirenato.silveira@gmail.com
Ricardo Tombesi Macedo
ORCID: https://orcid.org/ 0000-0001-7469-8446
Universidade Federal de Santa Maria, Brasil
E-mail: rmacedo1987@gmail.com

Resumo
Uma MANET (*Mobile Ad hoc NETwork*) é uma rede de nodos móveis e topologia ad hoc de fácil manutenção e alta robustez, onde os usuários podem ter acesso a uma infraestrutura básica de comunicação a qualquer hora e em qualquer lugar, sem necessitar de estações base. Detectores de Defeito são "oráculos" que, a partir de mensagens trocadas entre os nodos, são capazes de identificar nodos defeituosos. Como os detectores podem cometer enganos, suas suspeitas são utilizadas apenas para evitar que algoritmos não fiquem esperando indeterminadamente pela resposta de um nodo falho. *Gossip* é um algoritmo epidêmico baseado no fenômeno social chamado "fofoca", onde um nodo difunde (por *broadcast* ou *multicast)*, para seus vizinhos locais (dentro do seu alcance de transmissão), informações sobre um grupo de nodos vizinhos (locais ou remotos). Neste contexto, este artigo apresenta a implementação e avaliação de um detector de defeitos assíncrono não-confiável, baseado em *Gossip*, utilizando o emulador CORE (*COmmon Research Emulator*).
Palavras-chave: MANET; Detectores de defeito; Gossip; CORE.

Abstract
A Mobile Ad Hoc NETwork (MANET) is a network of mobile nodes and easy-to-maintain topology and high-strength where users can access a basic communication infrastructure anytime and anywhere without the need for base stations. Failure detectors are "oracles" that, from messages exchanged between nodes, are able to identify faulty nodes. Because detectors can make mistakes, their suspicions are only used to prevent algorithms from waiting indeterminately by a failed node response. Gossip is an epidemic-based algorithm builds on the social phenomenon called gossip, where a node announces (by broadcast or multicast) information to its local neighbors (within its transmission range) information about a group of neighboring (local or remote) nodes. This paper presents an implement and evaluate an unreliable asynchronous failure detector based on gossip using the CORE (COmmon Research Emulator).
Keywords: MANET; Failure detector; Gossip; CORE.

Resumen
Una MANET (Mobile Ad hoc NETwork) es una red de nodos móviles y topología ad hoc que es fácil de mantener y altamente robusta, donde los usuarios pueden tener acceso a una infraestructura de comunicación básica en cualquier momento y lugar, sin necesidad de estaciones base. Los detectores de defectos son "oráculos" que, a partir de mensajes intercambiados entre nodos, pueden identificar nodos defectuosos. Dado que los detectores pueden cometer errores, sus sospechas se utilizan solo para evitar que los algoritmos esperen indefinidamente a que responda un nodo fallido. Gossip es un algoritmo epidémico basado en el fenómeno social llamado "gossip", donde un nodo difunde (por difusión o multidifusión), a sus vecinos locales (dentro de su rango de transmisión), información sobre un grupo de nodos vecinos (locales o remotos). En este contexto, este artículo presenta la implementación y evaluación de un

detector de defectos asíncrono no confiable, basado en Gossip, utilizando el emulador CORE (COmmon Research Emulator).
Palabras clave: MANET; Detectores de defectos; Gossip; CORE.

1. Introdução

A disseminação de dispositivos móveis com comunicação sem fio, como telefones celulares, *smartphones* e *laptops* induz à construção de modelos de rede para áreas onde a arquitetura estruturada de rede convencional é inadequada (Kawamoto, *et al.,* 2013). Nestas áreas é necessária a implementação de redes de fácil manutenção e alta robustez, onde os usuários podem ter acesso a uma infraestrutura básica de comunicação a qualquer hora e em qualquer lugar, sem necessitar de estações base, o que pode ser alcançado por meio de uma MANET (*Mobile Ad hoc NETwork*) (Andrews, *et al.,* 2008; Miyao, *et al.,* 2009). Em uma MANET os usuários podem ser os próprios nodos da rede. Como resultado, seus dados podem ser transportados até uma área onde haja uma arquitetura de comunicação estruturada.

Uma MANET pode ser empregada em uma grande quantidade de ambientes de diversos tamanhos e topografias, mesmo frente a adversidades tecnológicas. Em áreas de desastres, por exemplo, é difícil fornecer acesso à rede devido a danos gerados em equipamentos de retransmissão de sinal. Um exemplo é a catástrofe que ocorreu no leste japonês em março de 2011 (G1, 2011), onde um terremoto e um subsequente tsunami afetaram drasticamente as infraestruturas de comunicação. Neste episódio, pessoas que não estavam cobertas por arquiteturas de comunicação tipo MANETs experimentaram o inconveniente da falta de informações devido à destruição de redes de comunicação convencionais.

Uma MANET apresenta as vantagens de ter nodos móveis e permitir autoconfiguração de sua topologia, pois não existe uma administração centralizada, o que a torna candidata para operar em ambientes sujeitos a falhas. Entretanto, justamente as vantagens de mobilidade e autoconfiguração fazem a rede ser sensível ao problema de diferenciar uma desconexão de um nodo por movimentação com perda de sinal ou de uma desconexão por falha do nodo, o que exige um algoritmo que permita analisar a mobilidade de um nodo, prevendo possíveis desconexões por movimentação ou falha.

Os algoritmos que coordenam esta tarefa são denominados detectores de defeitos (Chandra & Toueg, 1996), e são capazes de estimar se um dado nodo suspeito falhou, a partir de mensagens trocadas entre os nodos da rede. De acordo com Gracioli & Nunes (2007), para não depender de hierarquia ou topologia fixa, grande parte das estratégias de detecção de defeitos para redes móveis sem fio tem como base o algoritmo epidêmico do tipo *Gossip* (Renesse, *et al.,* 1998). O *Gossip* é um algoritmo epidêmico baseado no fenômeno social chamado "fofoca", onde um nó difunde (por *broadcast* ou *multicast*), para seus vizinhos locais (dentro do seu alcance de transmissão), informações sobre um grupo de nós vizinhos (locais ou remotos). Neste algoritmo, quando um nó detector de defeitos fica muito tempo sem receber notícias sobre um determinado nó (equivalente a um determinado *timeout*), o detector suspeita do nó. O ponto chave do algoritmo é que um nó conhece seus vizinhos ao longo do tempo por meio de mensagens que passam de vizinhos para vizinhos.

Como a estratégica epidêmica por si só não permite diferenciar nós defeituosos de nós móveis, Friedman & Tcharny (2009) procuraram representar a mobilidade permitindo que o *timeout* do algoritmo *Gossip* pudesse ser incrementado com múltiplos do atraso de rede a um *hop* de distância suportando, assim, lacunas no recebimento de mensagens de *heartbeat*.

Neste contexto, este artigo busca codificar e avaliar o desempenho do algoritmo de Friedman & Tcharny (2009) utilizando o *COmmon Research Emulator* (CORE) (Ahrenholz, *et al.,* 2008).

O restante deste trabalho está organizado como segue. A Seção 2 apresenta o modelo de sistema, discute detectores de defeitos e apresenta o trabalho que norteia este estudo. A Seção 3 apresenta a configuração utilizada e os resultados práticos obtidos a partir da simulação do algoritmo. Encerrando o artigo são apresentadas as considerações finais e as referências empregadas.

2. Modelo de Sistema e Detectores de Defeitos

Esta seção apresenta o modelo de sistema distribuído considerado e uma revisão sobre os detectores de defeitos.

2.1 Modelo de sistema

O sistema distribuído considerado neste trabalho é assíncrono e composto por um conjunto finito de nós móveis Π = {*p1*, ... , *pn*}, onde n > 1. Cada nó tem sua própria memória, unidade de processamento, relógio local e processos em execução. Não há relógio global. Os nós se comunicam por meio de troca de mensagens, usando difusão (*broadcast*) via rádio com alcance de transmissão finito e igual para todos os nós. A potência do sinal de recepção é variável dentro do alcance de transmissão. Uma mensagem *m* enviada por um nó *pi* somente é recebida por outro nó *pj* que esteja dentro do seu alcance de transmissão quando *m* for enviada e a potência do sinal de recepção da mensagem *m* puder ser identificada por *pj*. Todo nó móvel *pi* percorre, em um tempo finito, pelo menos uma trajetória *T* tal que ao longo de *T* o nó *pi* entra no alcance de transmissão de pelo menos um nó *pj* e recebe uma mensagem *m* de *pj*, e pelo menos um nó *pj* recebe uma mensagem de *pi*. A topologia de rede é dinâmica e os canais de comunicação são bidirecionais e confiáveis (não alteram, não criam e não perdem mensagens). Os nós são auto-organizáveis e adaptam-se de forma autônoma a falhas do tipo *crash*. Para controlar falhas, cada nó no sistema distribuído executa um serviço de detecção de defeitos.

2.2 Detector de defeitos

Dada a natureza assíncrona do sistema distribuído, o conceito de detectores de defeitos não-confiáveis foi introduzido por Chandra & Toueg (1996) para contornar o problema da impossibilidade de resolver o consenso em um sistema assíncrono sujeito a falhas (Fischer *et al*., 1985). A isto, dá-se o nome de *Impossibilidade FLP*. Segundo seus autores, Fischer, Lynch & Patterson (1985), não há como garantir o cumprimento das propriedades de um protocolo de consenso em um sistema distribuído assíncrono que esteja sujeito até mesmo a uma única falha por colapso. Isso se deve ao fato de que, na ausência de resposta de um dos processos, os demais participantes do consenso não têm como identificar se houve falha ou se o processo apenas está mais lento que os demais. Como em um ambiente puramente assíncrono não é possível impor nenhum limite para a recepção de mensagens, os participantes do consenso irão esperar indefinidamente pela resposta do processo falho.

Para resolver o consenso em ambientes puramente assíncronos, Chandra & Toueg (1996) propuseram um modelo de consenso auxiliado por detectores de defeitos. De fato, o mérito desta solução foi o de considerar as informações de componentes que podem cometer enganos, como no caso dos detectores de defeitos, aumentando o conhecimento sobre os demais elementos e estruturando os algoritmos de forma que as detecções incorretas não acarretem inconsistências entre os participantes do consenso. Dado isto, se tem que a ideia básica é encapsular o problema no detector de defeitos e fornecer uma noção de estado para a aplicação. O estado pode ser expresso por uma lista de nós suspeitos de estarem falhos (Chandra & Toueg 1996), uma lista de nós confiáveis (Aguilera, *et al*., 1998) ou uma lista de contadores de mensagens (Aguilera, *et al*., 1997). Normalmente, o estado que é informado à aplicação por um detector de defeitos corresponde à percepção do módulo de detecção local e um nó *pj* é dito *suspeito* quando um detector *pi* não recebe uma mensagem do nó *pj* dentro de um determinado intervalo de tempo.

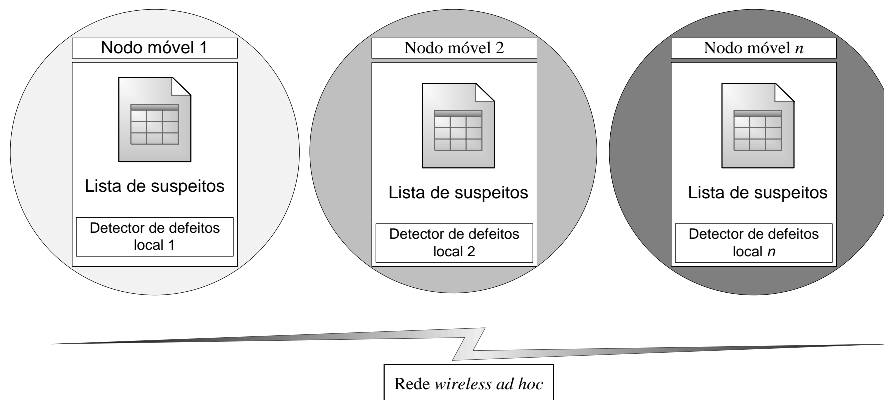
Há diferentes estratégias de comunicação para construir a noção de estado no detector (Felber, *et al*., 1999). Em redes cabeadas, as mais tradicionais são: i) *push*, estratégia baseada no envio periódico de mensagens de estado corrente (*I_am_alive!*) ou *heartbeat* (Aguilera, *et al*., 1997); e ii) *pull*, estratégia baseada no envio periódico de mensagens de solicitação de estado (*Are_you_alive?*) com consequente confirmação (*Yes_I_am!*). Em redes móveis sem fio, incluindo as MANETs, a mais tradicional é a estratégia baseada em "fofoca" (Friedman & Tcharny, 2009; Gracioli & Nunes, 2007), a qual

é baseada no envio periódico de mensagens *Gossip* que são repassadas de vizinhos locais para vizinhos remotos. Normalmente, uma mensagem *Gossip* carrega uma lista de identificadores e contadores de *heartbeat*.

A

Figura **1** apresenta uma abstração de detectores de defeitos locais em uma MANET com ***n*** nodos. Cada nodo da MANET executa, localmente, seu próprio detector de defeitos. Assim, cada "*i-ésimo* Nodo móvel" processa o "*i-ésimo* detector de defeitos local", que mantém uma lista de suspeitos que contém entradas para todos os outros ***n*** nodos do Sistema. Nestas listas, cada nodo "*i*" marca os "*j-ésimos*" restantes como suspeitos ou não suspeitos, para "$i \neq j$".

Figura 1. Abstração de detectores de defeitos em uma MANET.



Fonte: Autores (2020).

De acordo com Gracioli & Nunes (2007), para não depender de hierarquia ou topologia fixa, grande parte das estratégias de detecção de defeitos para redes móveis sem fio tem como base o algoritmo *Gossip* (Renesse, *et al.,* 1998). O *Gossip* é um algoritmo epidêmico baseado no fenômeno social chamado "fofoca", onde um nodo difunde (por *broadcast* ou *multicast*), para seus vizinhos locais (dentro do seu alcance de transmissão), informações sobre um grupo de nodos vizinhos (locais ou remotos). Neste algoritmo, quando um nodo detector de defeitos fica muito tempo sem receber notícias sobre um determinado nodo (equivalente o um temporizador "$T_{cleanup}$"), o detector suspeita do nodo.

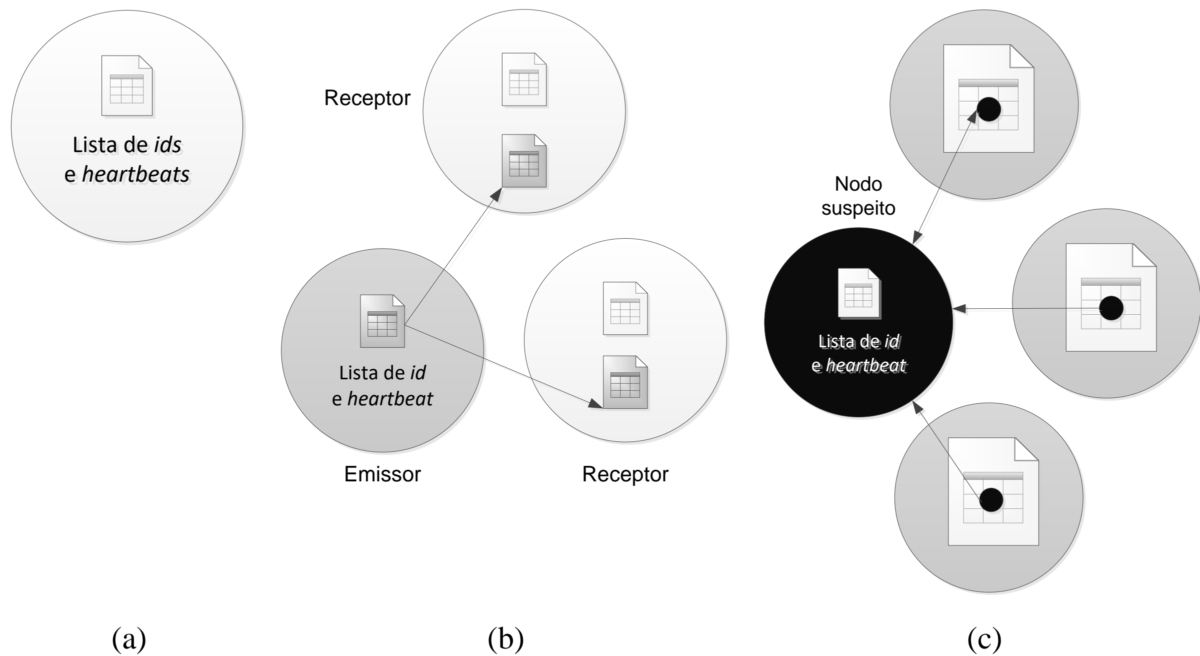
No algoritmo epidêmico do tipo *Gossip* (Renesse, *et al.,* 1998), cada nodo da rede é um monitor e mantém uma lista contendo o endereço e um contador de *heartbeat* (tipicamente um número inteiro) para cada outro nodo monitorado do sistema (

Figura **2**(a)). A cada intervalo T_{gossip}, um nodo escolhe aleatoriamente um ou mais vizinhos para enviar a sua lista (

Figura **2**(b)). Cada nodo mantém o instante do último incremento do contador de *heartbeats*. Se o contador não for incrementado em um intervalo de T_{fail} unidades de tempo então o nodo membro é considerado suspeito e colocado na lista de suspeitos após $T_{cleanup}$ unidades de tempo (

Figura **2**(c)). O ponto chave do algoritmo é que ele independe de topologia, pois um nodo conhece seus vizinhos ao longo do tempo por meio de mensagens que passam de vizinhos para vizinhos (Renesse, *et al.,* 1998). Por esta razão o algoritmo é capaz de suportar a mobilidade dos nodos em redes móveis, servindo como base para outros detectores de defeitos.

Figura 2. Exemplificação da execução parcial do algoritmo *Gossip*.



Fonte: Autores (2020).

Em MANETs, independente do uso de um algoritmo epidêmico, que dissemina a informação de nó para nó por meio de uma estratégia eficiente e escalável baseada em "fofoca" ou boatos, é preciso distinguir um comportamento defeituoso de uma movimentação do nó na rede. Uma das estratégias propostas para isto foi feita por Friedman & Tcharny (2009).

2.3 O detector de defeitos de Friedman & Tcharny (2009)

Friedman & Tcharny (2009) adaptaram para MANETs um detector de defeitos baseado no algoritmo *Gossip*. Os autores descrevem que o problema de *overhead* da comunicação, devido ao uso periódico de mensagens do tipo *heartbeats,* é superado por detectores de defeitos baseados *Gossip*. O algoritmo, ilustrado na

Figura **3**, assume um número conhecido de nodos e falhas do tipo *crash* e omissão de mensagens, além do conhecimento da mobilidade dos nodos. Um nodo periodicamente envia *heartbeats* a seus vizinhos. Um vetor é incluído em cada uma destas mensagens, de modo que cada entrada neste vetor corresponda ao mais alto *heartbeat* conhecido. A cada Δ unidades de tempo, cada nodo incrementa a entrada do vetor relativa a si e faz *broadcast* (*Gossip*) desta entrada para seus vizinhos. Ao receber uma mensagem deste tipo, um nodo atualiza seu vetor para o máximo de seu vetor local e o inclui na mensagem. Um nodo também associa um temporizador, iniciado por **Θ**, para os demais nodos. O valor de **Θ** considera que cada *heartbeat* é enviado a Δ unidades de tempo, adicionados de um *gap* que considera alterações na rota devido à mobilidade ou perda de mensagens. Se este *gap* for atingido, o nodo é considerado *suspeito*.

Figura 3. Algoritmo proposto por Friedman & Tcharny (2009).

Notation of node i:
$alive_i$ – array of heartbeat counters for each node
$suspected_i$ – bitmap array of suspected nodes

Initialization of node i:
$alive_i = [0, 0 \ldots 0]$
$suspects_i = [0, 0 \ldots 0]$
for all j do $suspect_timer_i[j].set(\beta(\gamma_0))$

Every α time units,
$alive_i[i] := alive_i[i] + 1$
broadcast(alive, $alive_i$)

Upon **receive**(alive, $alive_j$) from node j do
$\forall$ k, **if**($alive_i[k] < alive_j[k]$) **do**
$suspected_i[k] := 0$
$suspect_timer_i[k].set(\beta(\gamma_f))$
$alive_i := ArrayMax(alive_i, alive_j)$
done;

Upon $suspect_timer_i[j]$.**timeout** do
$suspected_i[j] := 1$
done;

Fonte: Friedman & Tcharny (2009)

Basicamente, o sucesso deste algoritmo se dá pelo cômputo utilizado para determinar os valores dos temporizadores. Tudo começa pelo cálculo de $\beta(\gamma_0)$, que é inicializado no início da execução do protocolo. Este valor determina o tempo máximo esperado para iniciar a audição do primeiro *heartbeat* do nó mais distante do sistema. Simplificando, denota-se por *D* o diâmetro esperado da rede, ou, em outras palavras, o número máximo de *hops* entre dois nós da rede. Então, γ_0 é *D* multiplicado pela latência esperada para cada *hop*. Assim, à medida que a densidade da rede aumenta, *D* se aproxima da máxima distância geográfica possível dividida pela faixa de transmissão *R*. Para fins ilustrativos e de simplificação da ideia de Friedman & Tcharny (2009), considera-se um conjunto de nodos dispostos em uma área quadrada de 300 metros x 300 metros e uma faixa de transmissão *R* de 50 metros. O diâmetro da rede se reduz à medida que o número de nós cresce. Intuitivamente, quanto maior é o número de nós na rede, o maior avanço médio pode ser conseguido por um único salto da origem até o destino. Isto é devido à maior probabilidade de haver pelo menos um nó dentro da faixa de transmissão de cada emissor em cada direção a partir deste emissor. No entanto, quando a densidade de nós já é alta, adicionar mais nós não aumenta significativamente a conectividade. Conforme o número de nodos aumenta, os pontos mínimos e máximos – usados no processo de cálculo – tornam-se próximos aos cantos – limites – da área, enquanto que *D* aproxima-se de 25 metros – metade de *R*. Desta forma, conforme aumenta o número de nodos, o diâmetro da rede aproxima-se de $[300\sqrt{2}/25] = 17$ nodos. Feitos os cálculos para uma rede com 50 nodos, se obtém $\gamma_0 = 24$ segundos. Os valores de cálculo dos temporizadores após a inicialização são determinado por γ_f e, basicamente é determinado pelos autores como algo entre 45 e 55 segundos.

Os resultados experimentais de Friedman & Tcharny (2009) exibem simulações para diferentes configurações da rede, em relação ao número de nodos e velocidade de movimentação dos nodos, entre outros aspectos. Os autores demonstram que, quanto maior o *timeout* para detecção de falhas, menor são os erros cometidos. Por outro lado, *timeouts* grandes aumentam o tempo de detecção de possíveis falhas. Além disso, se a rede apresentar melhor conectividade devido ao crescimento do número de nodos ou intervalo de transmissão, o número de falsas suspeitas diminui. Finalmente, os autores afirmam que, se um nodo se move rapidamente, o número de falsas suspeitas é menor, uma vez que este propaga mais uniformemente informações sobre o sistema.

3. Experimento

No experimento desenvolvido utilizou-se o emulador CORE (Ahrenholz, *et al.,* 2008) e a linguagem de programação Java para codificar, testar e avaliar o Detector de Defeitos proposto por Friedman & Tcharny (2009). Os detalhes para a implantação de todo este processo são descritos nas próximas subseções.

Quanto à natureza, esta pesquisa é um trabalho original. Para comprovar esta originalidade foi realizada uma revisão bibliográfica, apresentada na seção 2. Em relação a seus objetivos, tem-se uma pesquisa exploratória, já que buscou-se desenvolver um estudo comparativo de diferentes paradigmas. Acerca de seus procedimentos técnicos tem-se um trabalho experimental já que, a fim de comprovar que a proposta apresentada neste artigo é melhor e/ou equivalente as demais existentes, realizaram-se experimentos práticos que são, minuciosamente, descritos nas seções seguintes (Pereira, *et al.*, 2018).

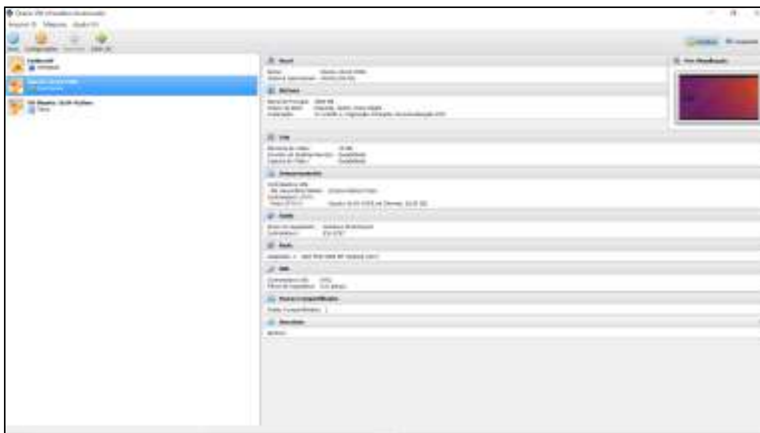
3.1 Configuração do experimento

O emulador CORE, disponível para *download* em https://www.nrl.navy.mil/itd/ncs/products/core, é disponibilizado em vários formatos, tais como pacotes de diretórios, código-fonte e imagens de máquinas virtuais. Para este trabalho, optou-se pelo pacote disponibilizado para o Sistema Operacional *Ubuntu*, versão 16.04.

A partir disto, criou-se uma máquina virtual usando o *software Virtual Box* 5.1.26 r117224 (Qt5.6.2), disponível em https://www.virtualbox.org/wiki/Downloads (

Figura **4**), com a seguinte configuração de *hardware*: 1 processador, 4 GB (*Giga Bytes)* de memória RAM (*Random Access Memory)*, HD (*Hard Disk*) de 20GB, Adaptador de Rede Intel PRO/1000MT, intercalado entre modos *NAT* e *Bridge*.

Figura 4. Tela com a configuração de *hardware* da máquina virtual utilizada.



Fonte: Autores (2020).

Tendo a máquina *Linux* pronta e configurada, seguiu-se uma série de comandos para instalar o CORE, apresentados na Figura 5: (1) e (2) para atualizar a distribuição *Ubuntu* 16.04; (3) para instalar o CORE; (4) para remover o módulo *Quagga* que, neste caso, deve ser substituído pelo módulo "*quagga-mr*" que é a implementação do protocolo *OSPF-MDR* para *Quagga OSPFv3*; (5) para baixar o *quagga-mr*; e (6), para instalar o *quagga-mr*. Feito isto, para executar o CORE são necessários os comandos (7) e (8) – utiliza-se *sudo* neste último porque pode ser necessário fazer alterações em */etc/hosts/*.

Figura 5. Comandos para instalar o CORE.

Comando	
sudo apt-get update	(1)
sudo apt-get upgrade	(2)
sudo apt-get install core-network	(3)
sudo apt-get remove quagga	(4)
sudo wget https://downloads.pf.itd.nrl.navy.mil/ospf-manet/quagga-0.99.21mr2.2/quagga-mr_0.99.21mr2.2_amd64.deb	(5)
sudo dpkg -i quagga-mr_0.99.21mr2.2_amd64.deb	(6)
sudo /etc/init.d/core-daemon start	(7)
sudo core-gui	(8)

Fonte: Autores (2020).

Uma das formas para implementar a mobilidade dos nodos de uma MANET no CORE se dá por meio do uso de *scripts* específicos para um simulador de redes chamado NS-2. Estes *scripts*, também chamados *traces*, são obtidos a partir do *software BonnMotion*, disponível no *link* <http://sys.cs.uos.de/bonnmotion>. O *BonnMotion* é um gerador de cenários para mobilidade. Para que tudo funcione, é necessário ter instalado e configurado o pacote JDK (*Java Development Kit*). Feito isto, deve-se baixar o arquivo disponível no *link* <http://sys.cs.uos.de/bonnmotion/src/bonnmotion-3.0.1.zip> e seguir as instruções de instalação/configuração disponíveis no manual da ferramenta. Para gerar um *trace*, usa-se o comando (9), como mostra a Figura 6, que vai gerar um arquivo intermediário denominado *sample*, usando o modelo de mobilidade *RandomWaypoint*, com 10 nodos, um quadrante de 300 metros (*x*) por 300 metros (*y*) com uma velocidade de deslocamento dos nodos de 5 metros por segundo; e o comando (10) Figura 6, que vai efetivamente gerar o *trace* em seu arquivo final.

Dado que o CORE enumera os nodos a partir de *n1* e os *traces* gerados pelo *BonnMotion* iniciam em *node_(0)*, é necessário editar o arquivo – usando um editor de textos qualquer – fazendo com que as informações de *node_(0)* sejam agora a do último nodo da rodada da simulação. Caso existam 10 nodos na simulação, *node_(0)* será substituído por *node_(10)*, por exemplo.

Figura 6. Comandos para gerar o *trace*.

Comando	
bm -f sample RandomWaypoint -n 10 -d 60 -x 300 -y 300 -h 5.0	(9)
bm NSFile -f sample	(10)

Fonte: Autores (2020).

Dado isto, codificou-se, em Java, a classe *FriedmanFD*, que implementa o algoritmo proposto por Friedman e Tcharny (2009). Neste código-fonte, a *thread doServer* implementa o *broadcast*; *doClient* implementa a recepção de uma mensagem; e *doSuspectTimer* faz o controle de *timeout* dos possíveis nodos suspeitos (

Figura **3**).

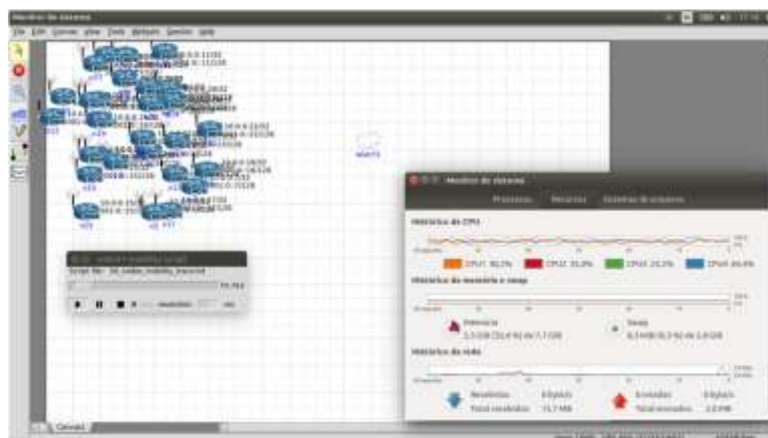
A fim de garantir a integridade dos resultados obtidos a partir das várias rodadas futuras, foram feitos alguns testes iniciais com uma MANET de características conhecidas e usadas para controle. Finalmente, dada a grande quantidade de tempo investida na correção de erros oriundos do código Java, registra-se que esta linguagem pode não ser a mais adequada para este tipo de aplicação, pois soluções de mais alto nível, como os *scripts Python*, podem resolver o problema sem apresentar este tipo de inconveniente.

3.2 Resultados do experimento

Na primeira rodada de testes se buscou observar o consumo de recursos computacionais utilizados pelo emulador CORE para a simulação de MANETs com diferentes quantidades de nodos. A medição se deu em dois ambientes diferentes: uma máquina física (*processador core i5; 8GB RAM; Ubuntu 16.04*) e uma máquina virtual (*Host: processador core i7, 16GB RAM e Windows 10; VM: processador de 4 núcleos, RAM de 12GB e Ubuntu 16.04*). A

Figura 7 estampa o monitor de recursos do sistema e também o monitor de consumo de recursos do emulador CORE; ambos foram utilizados nas medições.

Figura 7. Monitor de recursos do sistema.



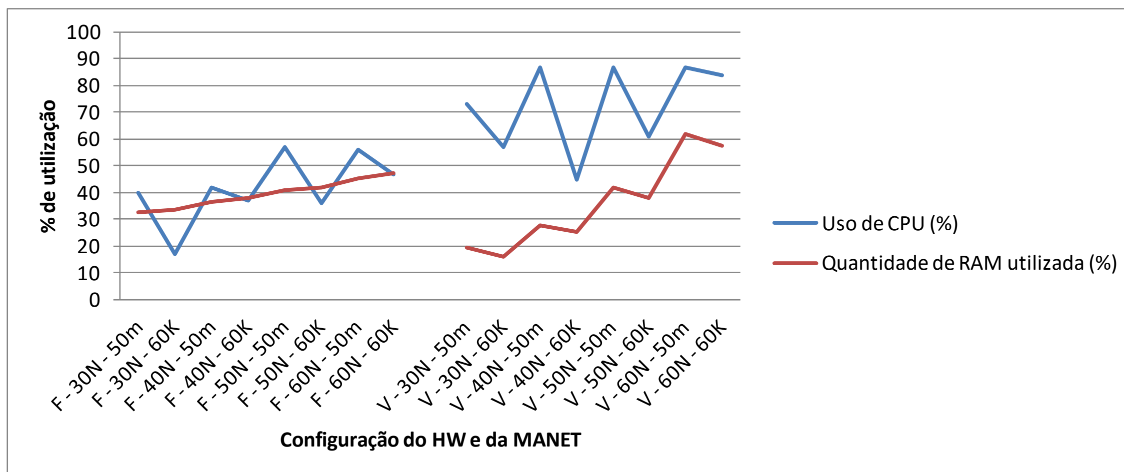
Fonte: Autores (2020).

Nos testes desta primeira rodada, expressos pela

Figura 8, foram feitas 16 medições. As mensurações feitas na máquina física são identificadas pela letra "*F*", enquanto que as executadas na máquina virtual, pela letra "*V*". Dois diferentes valores de "*refresh time*" (taxa de atualização da animação de movimentação dos nodos) foram utilizados: 50 milissegundos ("*50m*") e um minuto ("*60K*"). Cada medição foi coletada após 60 segundos de simulação. Os menores valores foram obtidos na máquina física: o uso mínimo do processador foi de 17% enquanto que o uso máximo foi de 57%; já na quantidade de memória RAM, o menor valor foi de 32,6% e o maior de 47%. Estes valores mostram que, mesmo com um *hardware* modesto, é possível executar emulações com um número significativo de nodos. Já na máquina virtual, os maiores valores de consumo de recursos foram obtidos: 87% de uso de processador e 61,9% de uso de memória RAM. Destaca-se aqui o uso do processador, que é bem maior do que o caso anterior. Isto retrata que as operações do emulador são desafiadoras para as aplicações de virtualização e se torna mais relevante quando as operações de maior atualização de vídeo ("*60K*") são aplicadas. Ademais, a gestão de memória também é um ponto fraco: nas duas baterias de teste ("*50m*" e "*60K*") foi necessário reinicializar a máquina virtual após a última rodada de testes, pois

foram esgotados os recursos de CPU (processador) e memória da máquina física. Para ambos os casos (máquinas física e virtual), enquanto o uso de CPU diminui para os testes com “60K” (menor atualização de vídeo), o mesmo só acontece, no quesito da memória RAM, na máquina virtual: por algum motivo desconhecido os testes com “60K” aumentaram, mesmo que minimamente, o consumo de RAM na máquina física.

Figura 8. Consumo de recursos computacionais em máquinas física e virtual.



Fonte: Autores (2020).

Em relação ao desempenho do código Java que implementa o algoritmo de Friedman e Tcharny (2009), denominado “*FriedmanFD*”, conclui-se que os resultados obtidos são similares àqueles apresentados em Friedman e Tcharny (2009). Para tratar os *logs* gerados a cada rodada – compostos de 30, 40, 50 e 60 arquivos do tipo “*txt*” (texto), contendo, respectivamente, média de 10.873, 15.324, 18.729 e 21.986 linhas – que permitiram medir o desempenho de *FriedmanFD,* foram codificados ferramentas específicas, para, por exemplo, tratar o número de mensagens recebidas (“*Hora: 15 Mensagem recebida: 18;2;[2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]*”), mensagens malformadas (“*MENSAGEM MALFORMADA RECEBIDA... DESCARTANDO... 11 -> 101*”), erros de recebimento (“*Hora: 1536 Mensagem recebida: 27;103;[103, 94, 92, 102, 94, 92, 94, 94, 101, 90, 102, 102, 102, 92, 100, 94, 102, 82, 94, 103, 95, 90, 102, 102, 102, 94, 102, 103, 102, 538888888]*”), suspeitas (“*Eu, 0, suspeitei de 2 no tempo 1795*”), recuperação de falsas suspeitas (“*Eu, 0, deixei de suspeitar de 12 no tempo 77*”) e várias outras de controle.

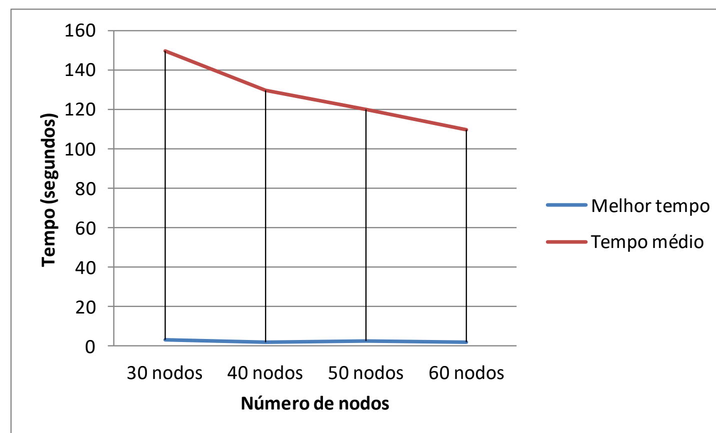
A emulação assume uma área de 300 metros quadrados. Inicialmente, todos os nodos são dispostos aleatoriamente nesta área. A mobilidade dos nodos utilizou o paradigma *Random Waypoint*. De acordo com este modelo, cada nó viaja para um ponto selecionado aleatoriamente dentro da área com uma velocidade selecionada aleatoriamente dentro do intervalo definido pelas velocidades mínimas e máximas permitidas. Então, um nó pode optar por ficar nesse ponto por um período selecionado aleatoriamente entre 0s e 90s. O alcance de transmissão dos nós (*Range*) foi de 50 metros. Foram utilizados diferentes valores para as velocidades máximas de movimentação (de 2 a 8 metros por segundo – m/s) e número de nós (de 30 a 60 nodos). Cada rodada de simulação durou 30 minutos.

Os resultados apresentados em Friedman & Tcharny (2009) foram obtidos pela média de dez rodadas diferentes com dez diferentes topologias de rede e padrões de movimento. Dadas as limitações de tempo existentes, oriundas do grande tempo

investido na implementação de *FriedmanFD*, os resultados ora apresentados foram obtidos a partir de uma única rodada de testes para as diferentes configurações de MANETs.

O primeiro experimento avalia o desempenho do *FriedmanFD,* ou seja, mede-se a quantidade de tempo gasto para detectar uma falha real. Aqui, se objetiva analisar o impacto do número de nós; se usam cenários contendo de 30 a 60 nós onde todos os nós estão configurados para se mover a uma velocidade de 8 m/s. Nesta etapa, em um determinado tempo de simulação conhecido, um nó é escolhido aleatoriamente para falhar. A Figura 9 apresenta os resultados obtidos nestas condições. O pior tempo corresponde ao tempo médio mais longo que *FriedmanFD* gastou para suspeitar definitivamente do nó defeituoso; e o tempo médio corresponde à qualidade de serviço apresentada pelo detector.

Figura 9. Tempo para detectar falhas (pior e médio prazos) de acordo com o número de nodos.

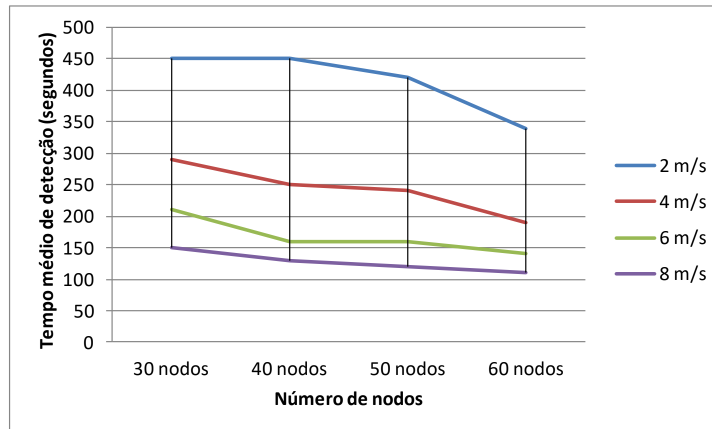


Fonte: Autores (2020).

Ainda na Figura 9 observa-se que o tempo de detecção diminui quando a quantidade de nós aumenta. Isso ocorre porque *FriedmanFD* adapta-se às características dinâmicas da rede. Em redes esparsas (30 nós), cada nó possui um número relativamente pequeno de conexões, enquanto em redes densas (60 nós) cada nó se conecta a uma quantidade significativa de nós de rede. Uma rede melhor conectada permite, à estratégia *Gossip*, utilizada pelo *FriedmanFD*, estabilizar-se melhor em cenários com falhas reais.

Ainda mantendo o interesse no tempo de detecção de falhas, se avaliou o desempenho de *FriedmanFD* considerando as variações de velocidade dos nodos. Assim, para uma rede de 30 a 60 nodos, aferiu-se o *FriedmanFD* a velocidades de 2, 4, 6 e 8 m/s. A Figura 10 mostra os resultados quantificados. Constata-se que o tempo de detecção diminui rapidamente com o aumento da velocidade do nodo. Essa redução ocorre porque o aumento da velocidade do nodo reduz o tempo que este nodo necessita para realizar uma determinada rota. Uma vez que o detector de defeitos estipula corretamente seus temporizadores (γ_f) com os intervalos esperados para classificar um nó como defeituoso, proporcionalmente melhora seus resultados para melhores tempos de detecção de falhas.

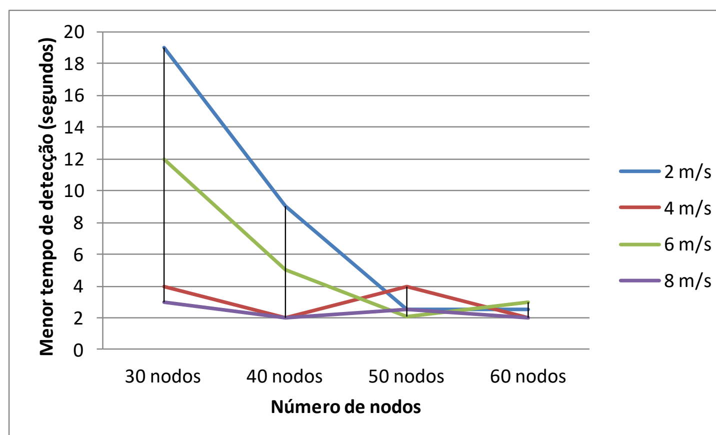
Figura 10. Tempo para detectar falhas de acordo com a velocidade e o número de nodos.



Fonte: Autores (2020).

Para verificar a rapidez com que o detector de defeitos pode reagir a uma falha, se repetiu o experimento anterior, mas, desta vez, considerando o tempo mínimo de detecção de falhas, presumindo um cenário de 30 a 60 nós e com velocidades variando de 2 a 8 m/s. A Figura 11 ilustra este resultado. É mostrado que o melhor tempo de detecção apresenta dependência relativa da velocidade e da quantidade de nodos quando a rede é menor do que 50 nodos; para valores superiores, há uma independência relativa. Esse comportamento ocorre porque o algoritmo não diferencia um nodo móvel de um falho. Quando ocorre uma falha em um nó que permanece na mesma região da faixa de transmissão por mais de duas rodadas de verificação (γ_f), o Detector reage de forma semelhante à de uma rede fixa.

Figura 11. Relação entre o menor tempo de detecção e a quantidade e velocidade dos nodos.

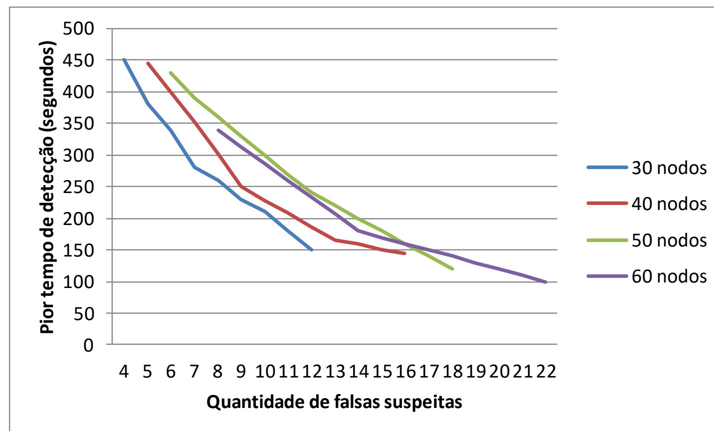


Fonte: Autores (2020).

O último experimento examina o comportamento do pior desempenho do detector considerando a variabilidade da quantidade de falsas suspeitas. Os resultados estão representados na Figura 12. É mostrado que, independente do número de nós, o tempo de detecção diminui proporcionalmente ao aumento do número de falsas suspeitas. Dado que o valor do temporizador (γ_f) é o mesmo para todos os nodos vizinhos, independente destes estarem mais perto ou mais distantes do nodo observador, este tende a suspeitar mais vezes dos nó

s vizinhos – mais distantes, que não foram emitiram mensagens no último intervalo (γ_f). Quando há menos nós na rede, há também menor troca de mensagens, o que tende a aumentar o valor de (γ_f).

Figura 12. Relação entre o maior tempo de detecção e a quantidade de falsas suspeitas.

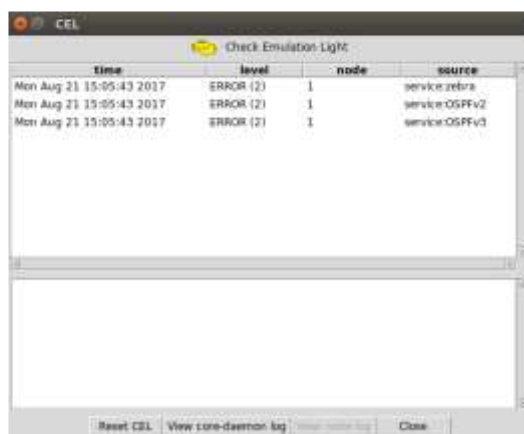


Fonte: Autores (2020).

4. Discussão do Experimento

Para o início dos trabalhos, optou-se por instalar a última versão disponível do Sistema Operacional *Ubuntu*, na data de realização do experimento, a 17.04. Ao dar início aos primeiros testes com o CORE, vários problemas foram detectados (**Erro! Fonte de referência não encontrada.**). Testamos as recomendações do manual do CORE, disponível no *link* https://downloads.pf.itd.nrl.navy.mil/docs/core/core_manual.pdf e também a *string* "*apt-get install core-network wireshark bridge-utils ebtables iproute libev-dev python tcl8.5 tk8.5 libtk-img xterm mgen traceroute quagga snmpd snmp-mibs-downloader snmptrapd mgen-doc autoconf automake make pkg-config python-dev libreadline-dev imagemagick help2man*", sem sucesso. Tentou-se também buscar maiores informações sobre a instalação/*configuração* do *Quagga*. Após contato com outros pesquisadores, foi recomendado o uso do *Ubuntu* 16.04. Feito isto, resolveram-se estes problemas.

Figura 13. Problemas detectados.



Fonte: Autores (2020).

A atividade de implementação do algoritmo de Friedman & Tcharny (2009) consumiu aproximadamente 240 horas, entre codificação e correção de *bugs*. A linguagem escolhida foi Java. A primeira versão do código continha duas classes – cliente e servidor – que implementavam *sockets* UDP (*User Datagram Protocol*); enquanto o servidor “ouvia” na porta 10.000, o cliente enviava datagramas para este local; nesta versão, o tamanho das mensagens tratadas poderia ser de até 256 *bytes*.

Adicionou-se a esta versão a escrita em arquivo das mensagens trocadas entre cliente e servidor. As mensagens trocadas se davam por meio de *broadcast* para o endereço "*255.255.255.255*".

Na segunda versão, unificou-se o código do cliente e servidor em uma única classe, criando-se duas *threads*: "*doServer*" e "*doClient*". Com isto, uma mesma classe faria o papel de cliente/servidor. Nesta versão verificou-se a necessidade da codificação de uma classe externa que finalizasse as *threads* criadas, algo que não é muito trivial em Java. Caso isto não fosse feito e a aplicação sofresse uma parada abrupta, os dados a serem escritos nos arquivos se perdiam e, com isto, toda a possibilidade de fazer o devido tratamento das mensagens trocadas pela rede. Para contornar este problema, se implementou a classe "*killAll*", que envia uma mensagem de parada, via *broadcast*, para todas os nodos da rede que estejam "*ouvindo*" na porta 10.000; todo o nodo que receber uma mensagem deste tipo, ecoada pela rede, deve gravar todos os seus dados no arquivo de *log* e finalizar sua execução.

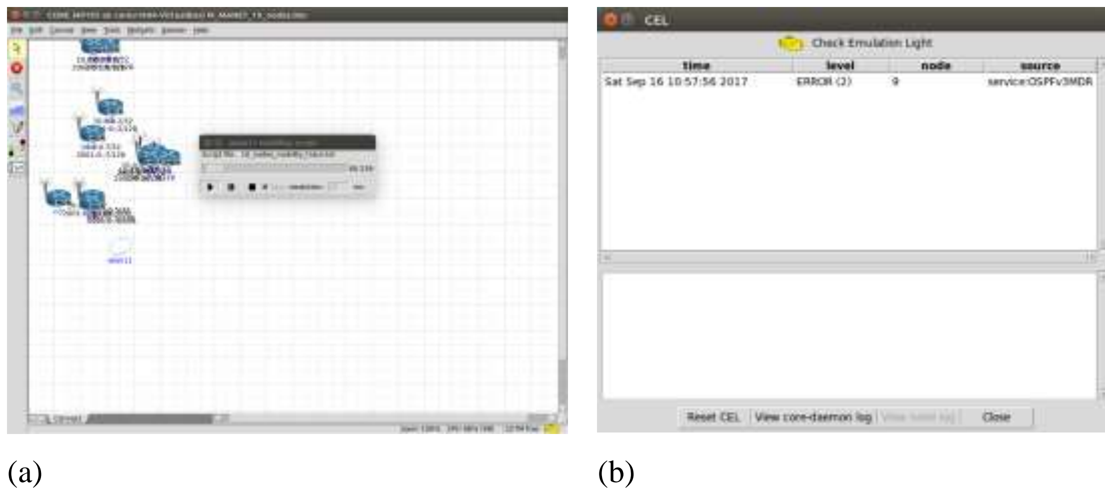
Na terceira versão do código, novos problemas foram contornados. O algoritmo de Friedman & Tcharny (2009) envia uma lista contendo informações sobre todos os outros nodos da rede. Quando aumenta o número de nodos, aumenta também o tamanho da mensagem a ser trocada. Dado isto, teve-se que tratar de forma dinâmica o tamanho das mensagens que, na versão inicial, era de 256 *bytes*. Ademais, além da lista de estado dos outros nodos, a mensagem que é recebida contém a identificação – *ID* – do nodo emissor e o valor atual de seu contador "*heartbeat*". Para que tudo funcionasse, foi necessário implementar um "*parser*" responsável por separar adequadamente todos os *tokens* da mensagem; para fins de ilustração, neste caso, esta mensagem – "*Hora: 45 Mensagem recebida: 9;4;[4, 3, 4, 3, 4, 3, 3, 3, 3, 4]*" – informa que, aos 45 segundos, o nodo observador recebeu uma mensagem do nodo cuja *ID* é *9* e cujo maior valor de *heartbeat* é *4*, além de toda sua percepção acerca dos demais nodos da rede, delimitados por "*[]*". Esta versão ainda tratou de restos de *strings* que eram recebidos pelo canal, além de incluir uma nova *thread*, "*doSuspectTimer*", responsável por fazer uma atualização da lista de nodos vizinhos suspeitos de acordo com um tempo determinado. Finalmente, o último entrave desta versão foi contornado com a implementação de um "*broadcast endereçado*": por padrão, o CORE não permite o broadcast quando em modo MANET. O endereço de *broadcast* disponível é "*Bcast:0.0.0.0*"; após contato com um dos mantenedores da lista de discussões, se obteve a informação de que "*In CORE, wired networks default to a /24 netmask and broadcast is set appropriately. For wireless networks, nodes default to a /32 netmask for MANET protocol purposes. MANET routing protocols form individual multi-hop network routes, so there really is no concept of a broadcast address or indeed a "subnet" as you typically think of in classical wired networking. You should be able to use multicast for any related purposes, or 255.255.255.255 can be used as broadcast for some things (e.g. DHCP).*". Do mesmo contato, se recebeu a seguinte informação: "*There was this issue that has been fixed if you grab the latest code from github: https://github.com/coreemu/core/pull/115*"; dado que estas soluções não funcionaram na prática, resolveu-se implementar um sistema que emite uma mensagem UDP endereçada a cada um dos membros que constitui a MANET; isto efetivamente resolveu o problema. Para garantir que todas as instâncias do detector de defeitos fossem finalizadas automaticamente quando do término da simulação e que os dados a serem gravados nos arquivos fossem preservados, implementou-se a classe "*killMySelf*" que, diferente da anterior, detecta a ID do nodo onde foi invocada e envia as mensagens de término para o seu nodo local.

A fim de garantir a integridade dos resultados obtidos a partir das várias rodadas futuras, foram feitos alguns testes iniciais com uma MANET de características conhecidas e usadas para controle.

A Figura 14 (a) ilustra uma rodada de emulação de uma MANET com 10 nodos. Percebe-se, no canto inferior direito, um ícone amarelo, denominado "*Check Emulation Light*". Ao clicar neste ícone, abre-se uma caixa (

Figura **1** (b)) que mostra um erro no serviço OSPFv3MDR. A análise dos *logs*, obtidos a partir do botão "*View core-daemon log*", mostra vários erros do tipo "*Error sending reply data: [Errno 32] Broken pipe*". Dado que o manual do CORE não apresenta detalhes acerca disto, cogitou-se ser um erro de inicialização devido à mobilidade inicial dos nodos.

Figura 14. Aviso de problemas em uma rodada da emulação.



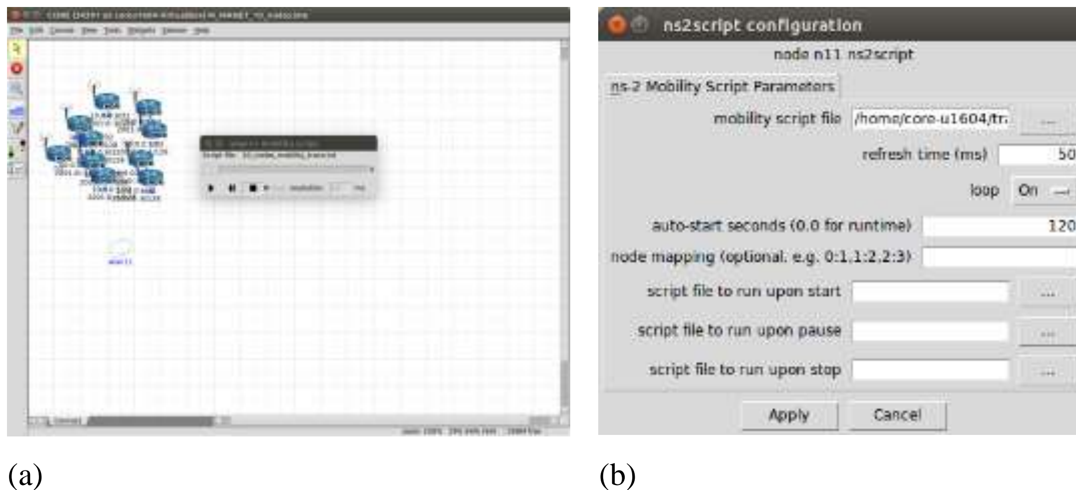
(a) (b)

Fonte: Autores (2020).

A fim de validar esta hipótese, utilizou-se a opção de reagrupar os nodos (
Figura **1** (a)) e iniciar o período de mobilidade dos nodos somente após 120 segundos de simulação (
Figura **1** (b)), aliado à opção “*Reset CEL*” (primeiro botão -

Figura **1** (b)). Isto resolveu o problema, pois a mensagem de atenção (ícone em amarelo) não mais foi percebida nesta nova rodada. Esta solução, entretanto, gerou falsas mensagens de suspeitas entre os nodos nos dois primeiros minutos de simulação e deveria ser tratada posteriormente pelo detector de defeitos. Para evitar a codificação de um "*patch*" específico para isto, se fez uma nova rodada, desta vez sem alterar o período de início de mobilidade e somente usando a opção "*Reset CEL*"; com isto, o problema se resolveu efetivamente.

Figura 15. Alteração do período inicial de mobilidade dos nodos.

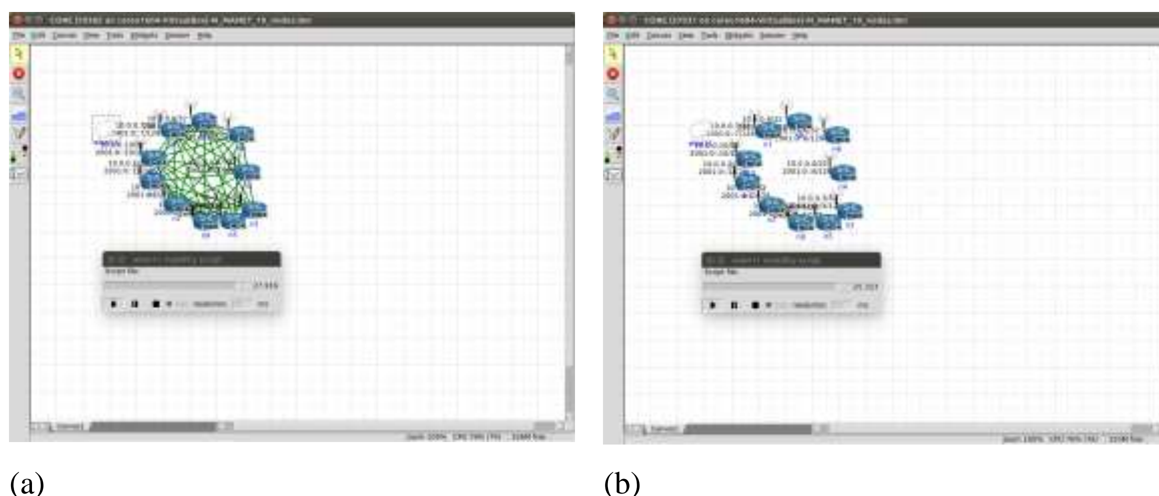


(a) (b)

Fonte: Autores (2020).

O último teste de integridade, desta vez referente aos dados reportados pelo "*FriedmanFD*", se deram em dois instantes: (1) em uma rodada sem mobilidade, que durou 600 segundos, com o "*Range*" de comunicação igual a 573.5294117647059 unidades, com todos os nodos totalmente conectados (
Figura **1** (a)), e, como era de se esperar, cada nodo observador recebeu, de todos os demais nodos observados, a mensagem "*Hora: 602 Mensagem recebida:* ID_j*;41;[40, 41, 40, 40, 40, 40, 41, 40, 40, 40]*"; (2) em uma rodada sem mobilidade, que durou 600 segundos, com o "*Range*" de comunicação igual a zero, com todos os nodos desconectados (
Figura **1** (b)), o nodo observador emitiu, para todos os demais nodos, a mensagem "*Eu, i, suspeitei de j no tempo XXX*".

Figura 16. Teste de integridade dos resultados controlados de "*FriedmanFD*".



(a) (b)

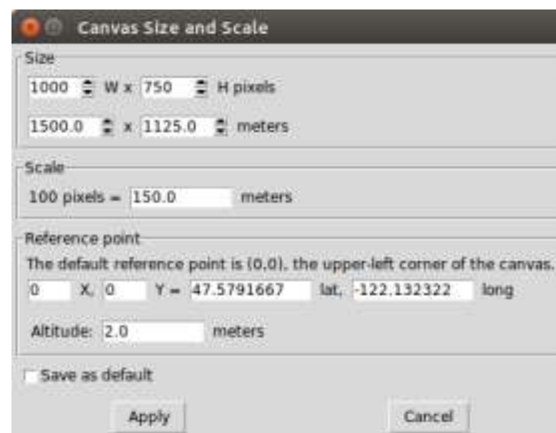
Fonte: Autores (2020).

As primeiras rodadas de teste mostraram alguns resultados inusitados. Ao verificar as configurações de rede, se percebeu que o "*Range*" do sinal, ou seja, a distância a que o sinal de transmissão/recepção alcança, é ditado de uma forma muito peculiar: "*The default configuration of the WLAN is set to use the basic range model, using the Basic tab in the WLAN con-figuration dialog. Having this model selected causes core-daemon to calculate the distance between nodes based on screen pixels. A numeric range in screen pixels is set for the wireless network using the Range slider. When two wireless nodes are within range of each other, a green line is drawn between them and they are linked. Two wireless nodes that are farther than the range pixels apart are not linked. During Execute mode, users may move wireless nodes around by clicking and dragging them, and wireless links will be dynamically made or broken.*". Nossa avaliação é baseada em uma distância de "*Range*" calculada em metros; logo, este cálculo certamente vai afetar nossos resultados; assim sendo, se optou por usar a medida padrão de "*275 pixels de distância para o Range*".

Após novas rodadas de teste, se percebeu que alguns nodos deixavam de receber mensagens após algum tempo. A análise dos *logs* mostrou a necessidade de dar tratamento de exceção na conversão dos números que eram recebidos do canal de comunicação ("*at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)*").

Dado que a manipulação para correção deste tipo de erro está fora do escopo deste trabalho, optou-se por descartar a mensagem incorreta, porém, registrando no arquivo de *logs* a sua existência. Também se passou a registrar os falhas de envio ("*IOException on Client sending: java.io.IOException: A rede está fora de alcance (sendto failed)*"), dado que, antes da delimitação do valor padrão para o "*Range*", alguns nodos passavam todo o tempo de teste sem emitir uma única mensagem. Quando os testes com MANETs com mais de 10 nodos foram efetuados, a máquina virtual parou de funcionar. Dado isto, se aumentou a quantidade de processadores para quatro e a quantidade de RAM para 12GB. O monitoramento dos recursos de *hardware* apontou que o gargalo se dá no uso de CPU, que chega a 100%. Ao se perceber que os nodos monitores não mais "*desconfiavam*" dos demais, buscou-se no manual do CORE, referências à palavra "*meters*"; com isto, se chegou ao *menu* de configuração de tamanho e escala (
Figura **1**) e, com isto, achar uma escala para conversão entre "metros e pixels"; aplicada uma regra de três, se determinou o valor de "*Range*" como 34 pixels, equivalente a 50 metros.

Figura 17. Menu Canvas do emulador CORE.



Fonte: Autores (2020).

Ao verificar que as emulações estavam ficando muito lentas, buscou-se reduzir o tempo de atualização da animação – movimentação dos nodos na tela. Com isto, modificou-se o valor de "*resolution*" ("*The resolution text box contains the number of milliseconds between each timer event; lower values cause the mobility to appear smoother but consumes greater CPU time.*"), que era de 50 milissegundos para 60.000 milissegundos (1 minuto). É importante o registro de que o arquivo "*/etc/hosts*" deve ser atualizado sempre que o número de nodos da rede é incrementado.

Corrigidos os problemas anteriores, foram realizados novos testes com MANETs contendo 30, 40, 50 e 60 nodos – a mesma quantidade utilizada no artigo de Friedman & Tcharny (2009). Percebeu-se que os nodos paravam de receber mensagem após alguns segundos de simulação. Este foi um dos erros mais difíceis de tratar: o problema se dava quando uma mensagem malformada era recebida pelo canal; esta mensagem acabava por fechar o *socket* que "*ouvia*" na porta 10.000 e também finalizava a *thread* que implementava "*doServer*"; para contornar este problema, se fez a *re-inicialização* do servidor de *sockets* na porta 10.000 e se iniciou uma nova thread "*doServer*". Este procedimento consumiu aproximadamente 30 horas (entre a depuração para encontrar o problema e a efetiva implementação do código final). O último problema observado foi a alteração de dados recebidos do canal ("*Hora: 1490 Mensagem recebida: 22;100;[97, 93, 97, 100, 93, 97, 100, 100, 93, 93, 93, 93, 100, 97, 97, 100, 100, 97, 85, 99, 97, 99, 100, 85, 100, 99, 97, 99, 97, 97, 99, 100, 93, 99, 97, 99, 97, 99, 97, 99, 97, 93, 97, 99, 99, 99, 97, 99, 93, 99, 85, 97, 99, 93, 99, 99, 85, 97, 97, 95555550]*"): o valor "*95555550*" é um erro, pois está fora da faixa de valores corretos; a fim de evitar mais horas de implementação para resolver o problema, optou-se por desconsiderar este tipo de valor.

Finalmente, dado toda a grande quantidade de tempo investida na correção de erros oriundos do código Java, se registra que esta linguagem pode não ser a mais adequada para este tipo de aplicação, pois soluções de mais alto nível, como os *scripts Python*, podem resolver o problema sem apresentar este tipo de inconveniente.

5. Conclusões

Este artigo buscou avaliar o algoritmo de Friedman & Tcharny (2009), implementado na linguagem de programação Java e executado por meio do emulador CORE. Os resultados obtidos são condizentes com o trabalho original, mostrando que este emulador apresenta-se como uma solução profícua para outras implementações de detectores de defeito em ambientes de MANETs.

Os resultados apresentados foram positivos. Diferentes de outros simuladores de redes, que são desenvolvidos em uma linguagem de programação específica (geralmente C++ ou Java), o CORE permite ao desenvolvedor escolher a linguagem de programação que melhor se adapte à resolução de seu problema; mais do que isto, permite que uma mesma solução seja codificada em diferentes linguagens de programação e, com isto, se alcancem vários benefícios associados, como a verificação de desempenho de um mesmo algoritmo em diferentes implementações. Ainda por se tratar de um emulador, sabe-se ser possível a medição de experimentos utilizando-se *hardware* real, o que acrescenta um grau de realismo ao experimento bem superior àquele disponibilizado pelos simuladores.

Em relação ao seu desempenho, o CORE mostrou-se "econômico", ou seja, permite a execução de testes com robustas configurações – redes com um número grande de nodos – sem exigir um *hardware* muito sofisticado, já que os testes aqui realizados foram feitos em *laptops* com configurações domésticas.

Finalmente, este trabalho buscou codificar e avaliar um detector de defeitos assíncrono não-confiável baseado em *Gossip* utilizando o emulador CORE. Os detectores de defeitos são mecanismos importantes para implementar redes móveis eficientes, onde o próprio dinamismo da rede, a escassez de recursos, a transmissão de mensagens sem fio e a dificuldade em

discernir entre um nó defeituoso e um nó móvel são fatores importantes. Como trabalhos futuros propõe-se a realização deste experimento utilizando outras linguagens de programação, tais como *Python*.

Referências

Aguilera, M., Chen, W., & Toueg, S. (1996) Randomization and Failure Detection: A Hybrid Approach to Solve Consensus. Distributed Algorithms: *International Workshop, WDAG '96* Bologna, Italy, October 9–11, Proceedings, Springer: Berlin, Heidelberg, 29-39.

Aguilera, M., Chen, W., & Toueg, S. (1997) Heartbeat: A timeout-free failure detector for quiescent reliable communication. *Distributed Algorithms*. 1320, 126-140. *Lecture Notes in Computer Science*.

Andrews, J., Shakkottai, S., Heath, R., Jindal, N., Haenggi, M., Berry, R., Guo, D., Neely, M., Weber, S., Jafar, S., & Yener, A. (2008) Rethinking information theory for mobile *ad hoc* networks. *IEEE Communications Magazine*, 46(12), 94-101.

Chandra, T., Toueg, S. (1996) Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2), 225-267.

Ahrenholz, J., Danilov, C., Henderson, T. R., and Kim, & J. H. (2008) CORE: A real-time network emulator. *MILCOM 2008 - 2008 IEEE Military Communications Conference*, San Diego, CA.

Felber, P., Defago, X., Guerraoui, R., & Oser, P. (1999) Failure detectors as first class objects. *Proceedings of the International Symposium on Distributed Objects and Applications,* 132-141.

Fischer, M., Lynch, N., & Paterson, M. (1985) Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2), 374-282.

Friedman, R., & Tcharny, G. (2009) Evaluating failure detection in mobile ad-hoc networks. *International Journal of Pervasive Computing and Communications*, 5(4), 476-496.

Gracioli, G., & Nunes, R. (2007) Detecção de defeitos em redes móveis sem fio: uma avaliação entre as estratégias e seus algoritmos. *Anais do Workshop de Testes e Tolerância a Falhas (SBRC/WTF)*, 159-172.

Johnson, D. B., & Maltz, D. A. (1996), Dynamic source routing in ad hoc wireless networks. In: Imielinski, T., & Korth, H. (Eds), *Mobile Computing*, Vol. 353, Kluwer Academic Publishers, Dordrecht.

Kawamoto, Y., Nishiyama, H., & Kato, N. (2013) Toward terminal-to-terminal communication networks: A hybrid MANET and DTN approach. *IEEE International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD),* 228-232.

Miyao, K., Nakayama, H., Ansari, N., & Kato, N. (2009) LTRT: An Efficient and Reliable Topology Control Algorithm for Ad-Hoc Networks. *IEEE Transactions on Wireless Communications*, 8(12), 6050-6058.

Pereira A. S., Shitsuka, D. M., Parreira, F. J., & Shitsuka, R. (2018). Metodologia da pesquisa científica. [e-book]. Santa Maria: UAB/NTE/UFSM. em: https://repositorio.ufsm.br/bitstream/handle/1/15824/Lic_Computacao_Metodologia-Pesquisa-Cientifica.pdf?sequence=1.

Renesse, R., Minsky, Y., & Hayden, M. (1998) A Gossip-style failure detection service. *Proceedings of the IFIP International Conference on Distributed Systems and Platforms and Open Distributed Processing (Middleware)*.