Implementação de protocolos epidémicos

João Beleza
FEUP - MIEEC
Porto, Portugal
201402831@fe.up.pt

Júnio Parente
FEUP - MIEEC
Porto, Portugal
up201405307@fe.up.pt

Tiago Neves
FEUP - MIEEC
Porto, Portugal
up201406104@fe.up.pt

***Resumo*—Na ausência da possibilidade de comunicação em grupo via *multicast* é necessário utilizar protocolos que nos permitam difundir uma mensagem na rede através de comunicação ponto-a-ponto. Este documento descreve a implementação feita de 2 grupos de protocolos epidémicos, *Gossiping* e *Anti-entropy*.**

***Index Terms*—*Gossiping*, *Anti-entropy*, *Multicast*.**

***Repositório* -** https://github.com/nvsclub/difusion_com

I. Definição do problema

Este projeto visa a emulação de protocolos epidémicos, com o objetivo de estudar a difusão de mensagens dentro de diversas topologias de rede.

Os protocolos de comunicação epidémicos que serão abordados são o protocolo *Gossiping* e o *Anti-entropy* [3].

No caso do *Gossiping* a difusão é desencadeada pela receção de uma mensagem nova por um nó. Este irá iniciar o protocolo de forma a difundir a mensagem pelos nós da rede dos quais tem conhecimento.

Através do feedback dos nós vizinhos, o nó emissor irá decidir quando continuar ou parar de difundir a mensagem. O feedback poderá ser: **Positivo**, permitindo ao nó continuar a transmitir a sua mensagem para outro nó vizinho; **Negativo**, forçando o nó tomar a decisão entre continuar ou não a propagar a mensagem.

Já para o *Anti-entropy* o protocolo é disparado por via temporal sendo executado ciclicamente a uma cadência definida. Existem várias implementações do protocolo de *Anti-entropy*, nomeadamente *PUSH*, *PULL* e *PUSH-PULL*.

A variante *PUSH* difunde a mensagem para um nó aleatório que posteriormente decide se continua a difundir a mensagem ou não.

No caso do *PULL* este pergunta a um nó vizinho se este possui uma mensagem mais recente que a sua, e em caso afirmativo o nó vizinho deverá responder com sua mensagem. Este último modelo é interessante pois apenas os nós que pretendem atualizar o seu valor serão executados, ao contrario do modelo *PUSH* no qual a rede tem de permanecer em sintonia sendo necessário o esforço de todos os nós para que a mensagem se propague.

Por fim, o *PUSH-PULL* combina características de ambos os protocolos. Este irá enviar uma mensagem na tentativa de difundir a mesma, tal como o protocolo *PUSH*. Caso o nó destino contenha uma mensagem mais recente este irá enviar a mensagem de volta, tal como no protocolo *PULL*.

Na tabela seguinte é possível comparar as principais diferenças entre os algoritmos:

Tabela I
Diferenças entre Gossiping e Anti-entropy

	Gossiping	***Anti-entropy***
Trigger	*Event*	*Time*
Completo	Não	Sim
Carga na rede	Variável	Constante

II. Descrição da solução

A. Ferramentas utilizadas

De forma a resolver o problema foi utilizada a linguagem *Java* juntamente com o *API UDP* com o objetivo de emular as comunicações [2]. Utilizou-se a biblioteca *Graphstream* para gerar e visualizar grafos.

Os gráficos que se encontram neste documento foram gerados utilizando *Python* e a biblioteca *matplotlib*.

B. Nós

Os principais objetivos de teste são a rapidez de propagação da mensagem para toda a rede, a cobertura dos protocolos e a carga colocada na rede. Desta forma é necessário implementar uma estrutura de forma a que seja possível emular vários tipos de condições de rede de forma a testar os vários algoritmos.

Utilizando uma abordagem modular é possível gerar várias topologias e escalar o problema para estruturas mais complexas. Para tal foi implementada uma nó que ao ser gerado irá ter uma porta atribuída para ouvir mensagens e possuirá uma lista de portas das quais tem conhecimento (nós vizinhos).

O nó irá inicialmente criar um socket na porta indicada fazendo o papel de servidor. Esta abordagem tem a vantagem dos nós serem corridos em paralelo permitindo escalar a rede em tamanho e mudar a sua estrutura.

C. Geração de topologias e Graphstream

De forma a visualizar o trabalho efetuado recorreu-se à biblioteca *graphstream* [1] que permite não só visualizar grafos como também gerá-los. A biblioteca fornece um conjunto de classes geradores de grafos dos quais os mais relevantes são o *Lobster*, o *Dorogovtsev-Mendes* e o *RandomEuclidean*.

A topologia *Lobster* emula uma rede onde existe uma comunicação mais linear entre os vários nós que posteriormente servem um subconjunto de nós.

Já a topologia *Dorogovtsev-Mendes* fornece-nos um grafo que corresponde a uma topologia de *peer-to-peer*.

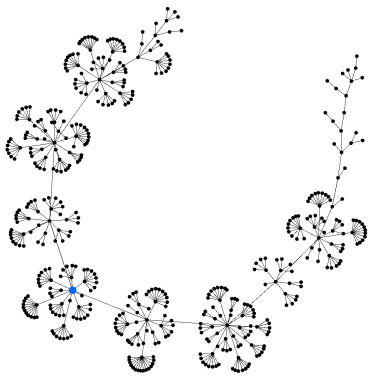

Figura 1. Topologia *Lobster*

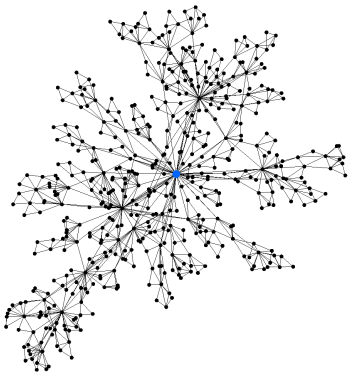

Figura 2. Topologia *Dorogovtsev-Mendes*

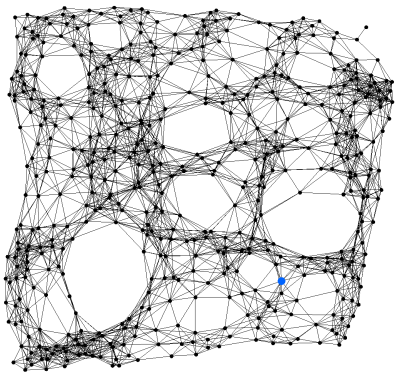

Figura 3. Topologia *RandomEuclidean*

Por fim, o *RandomEuclidean* fornece-nos um grafo que é análogo à distribuição da rede elétrica dentro de uma cidade com clusters mais ou menos densos. Desta forma será possível emular os protocolos em cenários variados.

É importante referir que estes grafos são gerados a partir da seed ”0”de forma a possibilitar a comparação entre protocolos garantindo, desta forma, grafos iguais entre testes. De forma a simplificar a implementação foi atribuído a cada nó uma porta com valor igual ao ID com um offset definido de forma a não utilizar portar reservadas.

Após a geração de grafos, é feita uma configuração da interface gráfica de forma a demonstrar em tempo real o comportamento da rede. A interface gráfica é relativamente fácil de utilizar sendo apenas necessário implementar procedimentos para alterar o aspetos individuais dos nós e das conexões de acordo aquando receção/envio da mensagem.

De forma a processar informação dos vários nós *(threads)* implementou-se junto à interface gráfica um servidor UDP que recebe informação dos vários nós e permite modificar o aspeto do grafo na interface gráfica permitindo visualizar a rede em tempo real. Este servidor irá recolher dados sobre o comportamento dos nós de forma a possibilitar a análise do comportamento da rede.

D. Mensagens

Aproveitando o facto de termos uma interface gráfica disponível, resolveu-se utilizar como mensagem o formato *DATA_ID_R,G,B*. O campo *DATA* representa um identificador do tipo de mensagem a ser enviado, o *ID* funciona como uma variante do relógio de Lamport, isto é, quanto maior o *ID* mais recente é a mensagem. Esta estrutura funciona neste caso pois apenas nos interessa a ordem das mensagens. Os campos *R,G,B* são a mensagem que se pretende difundir. Quando esta mensagem é recebida o nó atualiza a sua cor na interface gráfica.

III. Implementação

Todos os nós seguem uma implementação semelhante em termos estruturais onde apenas varia o protocolo implementado.

Os nós ficam à espera de receberem mensagens no seu servidor seguido de uma execução do protocolo. Quando esta execução acabar estes voltam a esperar por uma nova mensagem e repetem o protocolo. No caso das implementações do *Anti-entropy*, caso esta mensagem não chegue dentro de um timeout predefinido, a parte *time-triggered* do protocolo é executada e, quando finalizada, aguarda nova mensagem.

Para este projeto foram implementados 4 tipos de nós: *Gossiping, PUSH-PULL, PUSH* e *PULL*.

A. Gossiping

Os nós do tipo *Gossiping* estão permanentemente à espera de uma mensagem. Essa mensagem pode ser de 3 tipos: do tipo *DATA*, *Positive Acknowledge* ou *Negative Acknowledge*. No caso de se receber uma mensagem do tipo *DATA* o algoritmo vai verificar se a mensagem recebida é nova. Caso não seja é enviado um *Negative Acknowledge* e volta a aguardar por novas mensagens. Caso a mensagem seja nova este envia um *Positive Acknowledge* e envia a mensagem para outro nó da sua lista de nós conhecidos, da qual já foi retirado o nó que lhe acabou de enviar a mensagem.

Após envio da mensagem *DATA* o nó irá ficar à espera de receber uma nova mensagem. Uma das possibilidades será uma resposta do nó ao qual tentou difundir a mensagem. No caso do *Positive Acknowledge*, o nó continua a sua propagação e envia a mensagem para outro nó na sua lista, para o qual ainda não tenha enviado. Caso a resposta seja um *Negative Acknowledge*, este vai calcular se deve ou não continuar a sua

propagação de acordo com a probabilidade 1/k, com k a ser o número de *Negative Acknowledges* recebidos desde a receção da mensagem mais recente. Caso seja concluído que não deve enviar mais mensagens o nó termina a execução do protocolo e aguarda novamente a receção de novas mensagens.

B. Anti-entropy

1) PUSH: Na variante *PUSH* o protocolo é executado ciclicamente. O protocolo consiste em enviar a mensagem que o nó possui para outro nó conhecido.

O nó necessita também de ler as mensagens que vão chegando, e, aquando receção de uma mensagem mais recente, atualizar a mensagem que está a propagar.

2) PULL: O *PULL* utiliza o conceito oposto do *PUSH*: em vez do nó enviar a sua mensagem para os nós da vizinhança este envia um *request*. Após receção da mensagem do tipo *request* o nó envia na resposta a mensagem que possui.

3) PUSH-PULL: Combinando as duas variantes anteriores obtemos o *PUSH-PULL*: o protocolo envia ciclicamente a sua mensagem, tal como no *PUSH*, contudo existe a possibilidade de receber uma resposta por parte do nó ao qual envia a mensagem.

O nó que recebe uma mensagem pode agir de 3 formas distintas:

- Ignorar a mensagem caso esta seja igual à mensagem que possui.
- Atualizar a sua mensagem caso a recebida seja mais recente.
- Enviar de volta a sua mensagem caso possua uma mensagem mais recente que a recebida.

Este ultimo ponto corresponde à parte *PULL* da implementação.

C. Execução

Existem duas possibilidades para construir o grafo:

- Via argumentos no terminal
  java Constructor NodeType AlgorithmID SizeOfGraph ConnectionDelay Timeout onde *NodeType* corresponde a [*GOSSIP, PULL, PUSH, PUSHPULL*]; *AlgorithmID* corresponde ao ID associado a cada topologia; *SizeOfGraph* o número de nós de cada grafo; *ConnectionDelay* o delay fixo associado a cada conexão (em milissegundos); *Timeout* parâmetro associado aos algoritmos de Anti-entropy, de quanto em quanto tempo deve ser executado o protocolo (em milissegundos).
- Via menu
  java Constructor

De forma a iniciar os protocolos:

- Enviar uma mensagem inicial predefinida:
  java Starter x
- Enviar uma mensagem inicial personalizada:
  java Starter
- Enviar uma mensagem a cada 20 segundos:
  java Starter_data

IV. Resultados

A. Testes realizados

Para testar o funcionamento dos protocolos implementados foram feitos testes em 8 redes diferentes utilizando os seguintes comandos:

1) java Constructor GOSSIP 9 500 100
2) java Constructor PUSHPULL 9 500 100 100
3) java Constructor PUSH 9 500 100 100
4) java Constructor PULL 9 500 100 100
5) java Constructor GOSSIP 11 500 100
6) java Constructor PUSHPULL 11 500 100 100
7) java Constructor PUSH 11 500 100 100
8) java Constructor PULL 11 500 100 100

Estes testes testam 2 topologias, *Lobster* e *RandomEuclidean*, ambas com o mesmo tamanho de 500 nós e com o mesmo delay de 100ms. Os testes são realizados de forma igual em todos os 4 tipos de nós. O timeout para os algoritmos *Anti-entropy* é de 100ms.

Foram realizadas 100 iterações em cada teste.

Com estes testes vamos tentar responder às seguintes perguntas:

- Se o protocolo é capaz de difundir uma mensagem por todos os nós (completo vs incompleto)?
- Qual a melhor abordagem *PUSH vs PULL*?
- Será possível quantificar a melhor performance do *PUSH-PULL* em relação às outras variantes de *Anti-entropy*?
- Que tipo de protocolo é mais adequado para cada topologia de rede?
- Qual a carga na rede?

B. Protocolo é completo ou incompleto?

Apesar do objetivo ser difundir por toda a rede tal nem sempre acontece. Verificando os resultados de uma iteração dos testes nas redes 5) e 6), conseguimos verificar que o *Gossiping* não garante que a mensagem chega a todos os nós. Isso é possível verificar na figura 4 na qual se observem nós ainda coloridos a preto (sem receção da mensagem)..

Já na figura 5 é possível verificar que no caso do *Anti-entropy* todos os nós são cobertos.

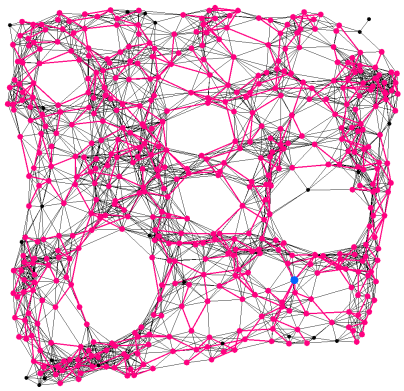

Figura 4. Rede do tipo *Gossiping* com uma mensagem difundida

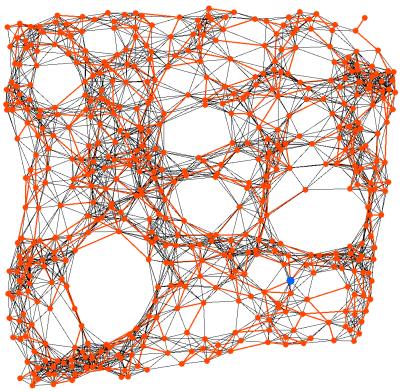

Figura 5. Rede do tipo *PUSH-PULL* com uma mensagem difundida

Ao verificar o conjunto de testes feitos nas figuras 6 e 7 é possível verificar que o *Gossiping* apenas cobre em média **57%** nós da rede, enquanto que o *PUSH-PULL* cobre, em média, praticamente **100%** da rede.

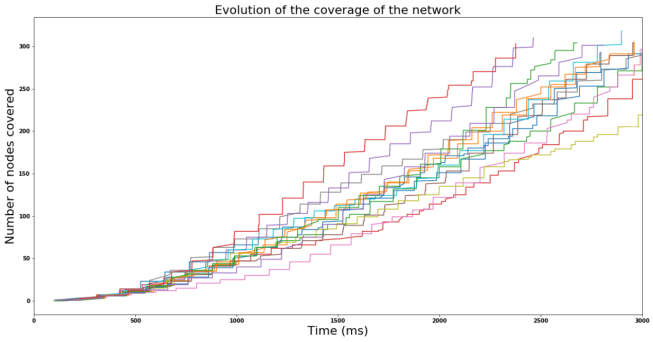

Figura 6. Comportamento do protocolo de *Gossiping* em 100 iterações

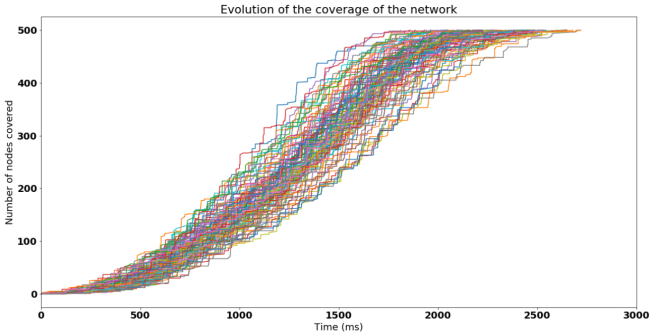

Figura 7. Comportamento do protocolo de *PUSHPULL* em 100 iterações

Confirma-se que o protocolo de *Gossiping* não garante que todos os nós recebam a mensagem em todas as iterações, logo considera-se um protocolo incompleto. No entanto, se o número de iterações for relativamente elevado e a perda de dados não for relevante, o protocolo acaba por atingir todos os nós, como se pode verificar na figura 8. Quanto ao protocolo *Anti-entropy* este foi capaz de alcançar todos os nós podendo ser considerado um protocolo completo.

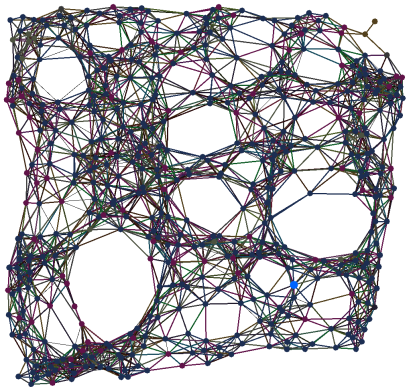

Figura 8. Rede do tipo *Gossiping* com várias iterações de mensagens

C. Qual a melhor abordagem PUSH vs PULL

É esperado que a variante *PULL* seja mais eficaz a propagar a mensagem que o *PUSH*, devido ao facto de ter uma probabilidade de propagação que crescem à medida que a propagação se efetua. Já no *PUSH* a probabilidade de propagar diminui à medida que a rede vai tomando conhecimento da mensagem.

Na parte final da propagação observa-se uma diferença bastante elevada, e, apesar da variante *PUSH* funcionar melhor no inicio da propagação, quando a rede aumenta a sua complexidade é superado pelo protocolo *PULL* o que pode ser verificado nas figuras 9 e 10.

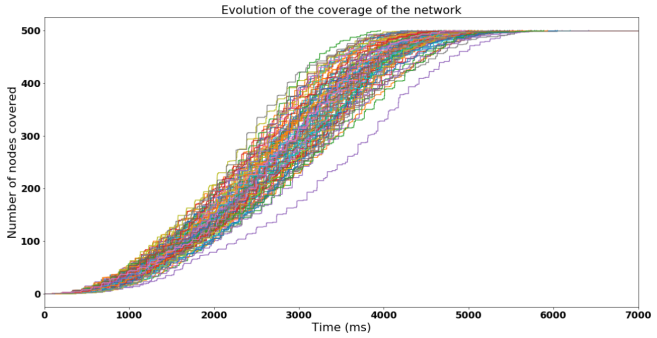

Figura 9. Comportamento do protocolo de *PUSH* nos testes

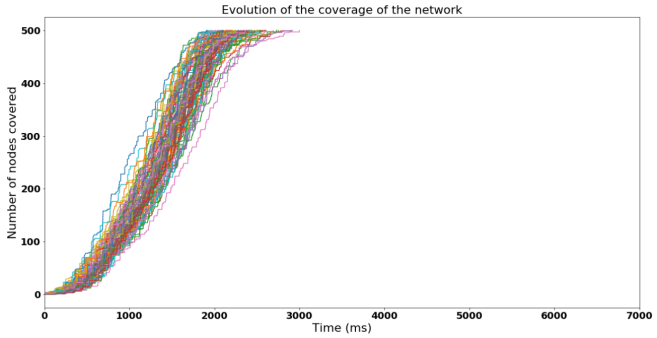

Figura 10. Comportamento do protocolo de *PULL* nos testes

D. Performance PUSH-PULL

O protocolo *PUSH-PULL* é um algoritmo que combina as funcionalidades das variantes *PUSH* e *PULL*. Será de esperar um comportamento tanto rápido no inicio da propagação como no fim da mesma.

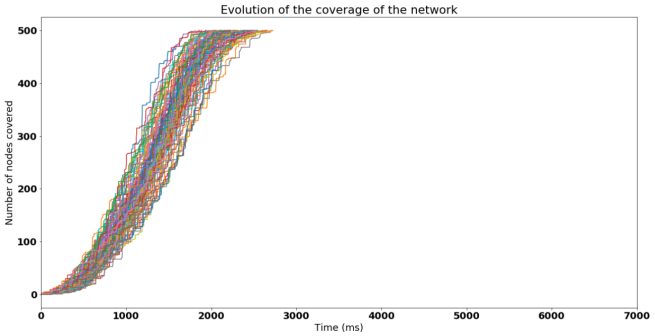

Figura 11. Comportamento do protocolo de *PUSH-PULL* nos testes

Apesar de existirem melhorias em relação ao *PULL*, estas apenas são visíveis em redes de tamanho pequeno. É no entanto possível quantificar a melhoria em relação à variante *PUSH*, comparando o número de mensagens que foram transmitidas pela parte de *PULL* do protocolo. A figura 12 indica-nos exatamente a divisão entre os nós que recebem a mensagem por via *PULL* e por via *PUSH*.

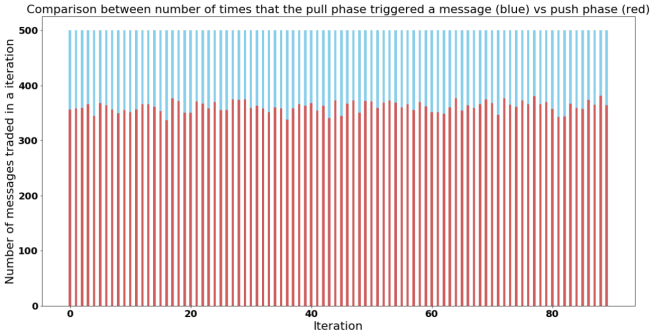

Figura 12. Divisão de mensagens *PUSH* (azul) e *PULL* (vermelho)

E. Protocolo mais adequado a cada Topologia

Ao realizar os testes na topologia *Lobster* é possível notar bastantes diferenças em relação à topologia *RandomEuclidean*.

Utilizando o protocolo *Gossiping* é possível verifica-se uma muito mais eficiente transmissão de mensagens. Isto deve-se ao facto de não existirem *Negative Acknowledges* pois a topologia de rede só permite um caminho entre dois nós, evitando assim conflitos. A utilização deste protocolo neste tipo de topologia garante que o protocolo funcione de forma completa, tal como se pode verificar na figura 13.

Já no caso dos protocolos de *Anti-entropy* verifica-se uma enorme dificuldade em propagar a mensagem nestes tipos de topologia. Definindo um tempo de execução máximo de 20 segundos verifica-se que os protocolos têm dificuldades em difundir a mensagem para todos os nós, e, conforme é visível

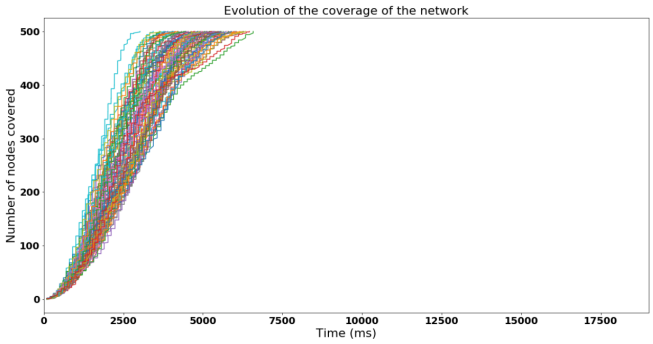

Figura 13. Comportamento do protocolo de *Gossiping* nos testes com topologia *Lobster*

na figura 14, necessitam de pelo menos 10 segundos para que a mensagem seja propagada para toda a rede. Comparando com o algoritmo de *Gossiping* verifica-se que este consegue tempos inferiores ao melhor caso do *Anti-entropy* em todas as iterações.

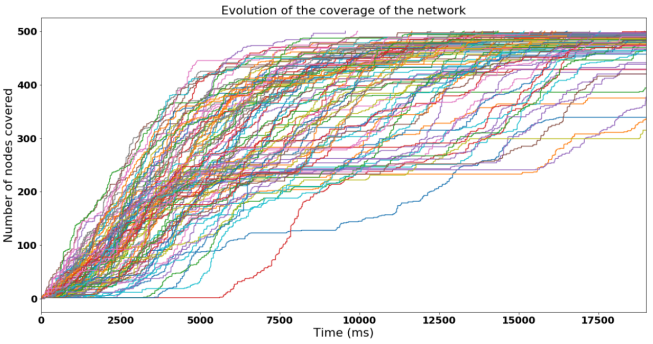

Figura 14. Comportamento do protocolo de *PUSH-PULL* nos testes com topologia *Lobster*

Concluímos então que o comportamento dos protocolos vária de acordo com a topologia da rede em que se encontram implementados. Para redes com topologias mais lineares a protocolo Gossiping tem clara vantagem ao invés que para redes mais entrópicas protocolos *Anti-Entropy* têm maior taxa de sucesso em menor tempo, como possível de observar nas figuras 10, 12 e 13.

F. Carga na rede

Outro aspeto a ter em conta é a carga que os protocolos colocam na rede.

Para o caso do *Gossiping* o comportamento não é completamente previsível, verificando-se uma quantidade elevada de mensagens num curto espaço de tempo, o que pode prejudicar o comportamento estável da rede. No entanto tem a vantagem de, após paragem do protocolo, a carga na rede passa a ser zero até nova execução do protocolo, o que implica a difusão de uma nova mensagem. É possível verificar na figura 15 em comparação com a figura 16 que o numero de mensagens necessárias para o protocolo é bastante reduzida, quando

comparando a escala utilizada para o número de troca de mensagens entre um protocolo e o outro.

As variantes de *Anti-entropy* possuem a vantagem de, por serem *time-triggered*, ter um comportamento mais constante e previsível na rede, o que pode ser necessário para garantir o comportamento correto da rede noutras funções. No entanto, a rede é obrigada a estar em constante alerta, fazendo com que a carga na rede esteja sempre presente. O facto de os nós estarem constantemente a fazer pedidos resulta numa quantidade de mensagens superior ao *Gossiping*, como é possível verificar na figura 16.

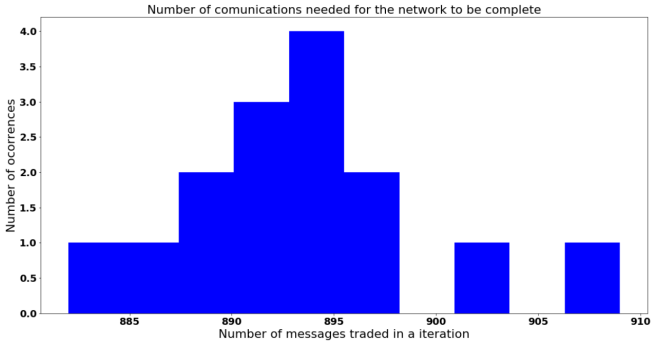

Figura 15. Distribuição do número de mensagens transmitidas no protocolo *Gossiping*

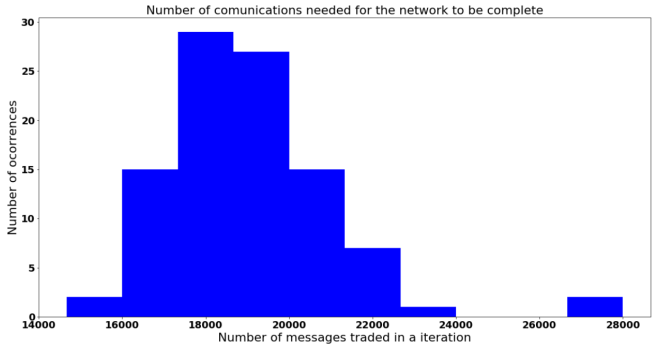

Figura 16. Distribuição do número de mensagens transmitidas no protocolo *PUSH-PULL*

V. Análise Crítica

Concluímos que não existe um protocolo melhor que o outro em todos os aspetos. Para que tal acontecesse era necessário que um protocolo fosse superior em 3 parâmetros: Rapidez, cobertura e carga na rede.

Como todas as categorias estão intrinsecamente ligadas à topologia associada é necessário primeiramente estudar a topologia de rede em que se pretende inserir o protocolo e posteriormente tomar a decisão acerca de qual deverá ser o protocolo a utilizar. Para cada topologia vs protocolo haverá sempre um trade-off em relação aos parâmetros definidos.

Mesmo dentro das variantes de *Anti-entropy* é possível verificar que cada uma destas possui vantagem num dos parâmetros: o *PUSH* coloca menos carga na rede e o *PUSH-PULL* é mais rápido. O *PULL* é redundante com o algoritmo de *PUSH-PULL* visto que introduzem a mesma carga na rede. No entanto o *PUSH-PULL* é mais rápido para a transmissão em redes de pequena dimensão.

Referências

[1] Graphstream. http://graphstream-project.org. [Online; acedido a 15-12-2018]. 2018.

[2] FEUP - SDSI. udp_java. https://web.fe.up.pt/ pfs/aulas/sd2008/tp/tp1/udp_java.pdf. [Online; acedido a 10-12-2018].

[3] FEUP - SDIS. Multicast Comunications. https://sigarra.up.pt/feup/pt/conteudos_geral.ver?pct_pag_id=249640&pct_parametros=pv_ocorrencia_id=420326 [Online; acedido a 12-01-2019].2018.

[4] FEUP - SDI. Clients ans Servers. https://sigarra.up.pt/feup/pt/conteudos_geral.ver?pct_pag_id=249640&pct_parametros=pv_ocorrencia_id=420326 [Online; acedido a 10-01-2019]. 2013.