Implementation and Experimental Measurements of Membership and Broadcast Algorithms

Rodrigo Faria Lopes*
rfa.lopes@campus.fct.unl.pt
MIEI, DI, FCT, UNL

Miguel Candeias†
mb.candeias@campus.fct.unl.pt
MIEI, DI, FCT, UNL

Salvador Rosa Mendes‡
sr.mendes@campus.fct.unl.pt
MIEI, DI, FCT, UNL

ABSTRACT

Este artigo tem como objetivo a implementação e comparação de alguns algoritmos de Bradcast e Membership que servirão como base de uma aplicação previamente fornecida. O foco deste artigo é testar como funcionam todas as combinações dos algoritmos propostos e avaliar os seus resultados. Os algoritmos que iremos abordar são: HyParView [1] e Cyclon [2] como algoritmos de *Partial Mermbership*, e os algoritmos Plumtree [3] e Eager [1] como algoritmos de *Broadcast*.

O trabalho desenvolvido começou pelo estudo e implementação dos algoritmos de forma individual utilizando a biblioteca Babel [4]. Posteriormente foram desenvolvidos testes para cada par de algoritmos (Broadcast + Membership) variando várias parâmetros, tais como o tamanho do payload e a taxa de transmissão. Por fim, os resultados obtidos serão discutidos e avaliados de acordo com os nossos resultados.

ACM Reference Format:
Rodrigo Faria Lopes, Miguel Candeias, and Salvador Rosa Mendes. 2020. Implementation and Experimental Measurements of Membership and Broadcast Algorithms. In *The Projects of ASD - first delivery, 2020, Faculdade de Ciências e Tecnologia, NOVA University of Lisbon, Portugal*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

No contexto de propagação e disseminação de mensagens,iremos analisar dois tipos de protocolos, os protocolos *gossip* e protocolos baseado em estruturas fixas,neste caso uma broadcast tree.

Os algoritmos de *gossip* são bastante utilizados devido à sua elevada resiliência e distribuem de forma eficaz a carga entre todos os nó se funcionam seleccionando da sua vizinhança um determinado número de nós deforma aleatória para o qual vão enviar uma mensagem. Devido aos problemas de memória associados à escalabilidade destes algoritmos foram criados algoritmos de *partial view*, em que cada nó tem apenas uma visão parcial de todos os nós pertencentes à *membership* e passam a poder apenas escolher dessa *pool*, isto resolve os problemas de escalabilidade mas introduz novas falhas relativamente à tolerância de falhas dos sistemas.Neste artigo vamos apresentar diversos protocolos de *broadcast* e *membership* de variados tipos,iremos falar sobre as suas implementações,alguns estudos sobre eles. Vamos dar a conhecer também algumas métricas de desempenho resultantes de testes com as quais vamos comparar a performance relativa de cada protocolo. Os testes foram desenvolvidos para cada par de algoritmos (Broadcast + Membership) fazendo variar parâmetros tais como o tamanho do *payload* e a taxa de transmissão.

O restante artigo está dividido em quatro partes e da seguinte forma. Na secção 2 são apresentados os protocolos de disseminação e *Partial Membership* escolhidos para estudo e implementação neste artigo, incluindo breves descrições deles e das suas vantagens e desafios bem como breves menções de trabalhos relacionados com os mesmos. Na secção 3 iremos falar de forma um pouco mais aprofundada acerca da implementação de cada um dos protocolos analisados. A secção 4, estará subdividida em três partes, sendo na primeira apresentadas as metodologias de teste utilizadas para avaliar cada protocolo, na segunda os resultados que vieram dos testes realizados e na última teremos uma breve discussão acercas dos resultados obtidos e como os protocolos se comparam entre si. Por último na secção 5 apresentamos as nossas conclusões acerca dos protocolos escolhidos e da sua performance.

2 RELATED WORK

Nesta secção começamos por explicar os algoritmos de disseminação escolhidos para este artigo, tais como o Eager Push e o Plumtree, falando um pouco sobre cada um deles. Depois é feita outra descrição semelhante, mas desta vez para os algoritmos de *Partial Membership*, como o HyParView e o Cyclon.

2.1 Epidemic Dissemination Protocols

2.1.1 Eager Push Gossip. O estudo deste algoritmo, teve como bases a seguinte dissertação [5], para alem do material leccionado nas aulas.

O Eager Push é uma estratégia de *gossip*, cujo um dos seus objectivos é assegurar que uma mensagem seja disseminada por toda a rede, fazendo assim chegar a mensagem a todos os processos nesta, mas apenas enviando um reduzido numero de mensagens, que a pouco e pouco vão começando a chegar a todos os processos. Por consequência disto, a condensação do *overlay* torna-se menos pesado, tornado assim a eficiência de entrega de mensagens maior, mas até um certo ponto, sendo este um dos problemas que irá ser discutidos mais adiante no relatório.

Uma das preocupações que tivemos com o protocolo, mas não foi implementado com sucesso, foi um mecanismo para tentar poupar recursos de memória, mais concretamente a nível das mensagens que o algoritmo guarda para não estar constantemente a enviar uma mensagem que já tenham chegado a todos os processos. O mecanismo passaria por tentar limpar o armazenamento das mensagens após x tempo, mas na implementação houve o problema do algoritmo estar a enviar duplicados, fazendo assim diminuir a performance deste.

*Student number 50435. Rodrigo foi responsável pela implementação do HyParView e escrita dos scripts para a facilitação de testes automáticos no Cluster.
†Student number 50647. Miguel foi o responsável pela implementação do Cyclon e do EagerPush, assim como do script em python para analizar os logs e gerar as estatisticas experimentais.
‡Student number 50503. Salvador desenvolveu a implementação do PlumTree.

ASD20/21, 2020, FCT, NOVA University of Lisbon, Portugal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Os detalhes da implementação deste protocolo serão explicados mais à frente na secção 3.1.1.

2.1.2 Plumtree. Toda a informação respetivamente ao Plumtree e a sua implementação foi baseado no artigo [3]. O Plumtree foi desenvolvido com o intuito de criar um meio termo entre as vantagens e desvantagens de *gossip Protocols* e *Broadcast Trees* de modo a conseguir uma implementação com baixa complexidade e *overhead* em termos mensagens e uma alta confiabilidade. Apesar de em testes e simulações o Plumtree ter apresentado uma boa capacidade de escalabilidade e rápida recuperação de falhas, este apresenta também uma performance reduzida quando existe mais do que um nó propagador de mensagens na rede. No Plumtree o envio de mensagens é feito maioritariamente através de *push gossip*, sendo efetuado Eager Push nos ramos da árvore e *lazy push* nas restantes ligações de modo a garantir a confiabilidade do sistema.

2.2 Partial Membership Protocols

2.2.1 HyParView. Todo o estudo deste algoritmo foi baseado no artigo [1], tendo sido aplicadas melhorias ao atual algoritmo.

A melhoria proposta tem como objetivo desenvolver um pouco mais o protocolo de *Join* do algoritmo proposto em [1]. Este inicialmente apenas requer que o membro que se queira juntar à rede, tenha conhecimento de um único nó já existente na mesma, fazendo um *Join Request* a este e assumindo que corre tudo como planeado, o novo nó entra na visão ativa do nó de contacto, fazendo com que o novo nó entre na rede como um novo membro da mesma.

Esta solução, bastante básica, requer um pensamento mais critico para criar uma solução mais robusta e fiável. Se levantarmos questões tais como: "Se o meu contacto na altura estiver em baixo?" rapidamente percebemos que a solução inicialmente proposta não será a ideal. Foi então que decidimos implementar um protocolo mais robusto que visa ir ao encontro destes problemas.

Agora, para um novo nó se juntar à rede de membros, poderá não só inserir um contacto, mas sim um conjunto de contactos, que, de forma aleatória, serão escolhidos um por um para serem o contacto directo do novo nó. Isto irá mitigar o problema de o nosso contacto na altura do *Join Request* estar em baixo, porque caso isso acontece, iremos de forma aleatória escolher outro contacto do nosso conjunto, havendo por isso uma maior probabilidade de sucesso na entrada do novo membro na rede.

Os detalhes da implementação deste protocolo serão explicados mais à frente na secção 3.2.1.

2.2.2 Cyclon. Este algoritmo foi estudado com base na seguinte dissertação [5] e no seguinte artigo [2].

O funcionamento deste, é baseado na estratégia de guardar apenas uma vista parcial da vizinhança de um processo, invés deste conhecer todos os nós participantes na *overlay network*. Adicionalmente, quando um qualquer nó pretende juntar-se ao *overlay*, este tem também a possibilidade de adicionar um contacto a priori existente na rede, ou não.

Apesar de não ter sido implementado, uma das possíveis maneiras de melhorar a escalabilidade do protocolo teria passado por, durante a inicialização deste, sempre que um processo pretenda tentar comunicar com um nó cujo a sua vizinhança esteja cheia, o protocolo poderia ter também a opção de tentar comunicar com outros nós existentes na rede até conseguir obter um nó com capacidade de adicionar um vizinho.

Para a *overlay network* ser prestável, poder suportar uma rápida disseminação de mensagens e ser altamente tolerante a falhas, o Cyclon oferece ainda fortes garantias das seguintes enumeradas relativamente à *overlay network*:

- Conectividade
- Grau de distribuição
- Precisão

Os detalhes de como funciona a estratégia de manutenção da vizinhança do protocolo, e a implementação deste serão explicados mais à frente na secção 3.2.2.

3 IMPLEMENTATION

Nesta secção vamos falar sobre a implementação dos algoritmos. Decidimos dividir esta secção em duas partes de forma semelhante à secção 2, de modo a separar as componentes do projeto.

Na secção 3.1 iremos abordar a implementação dos algoritmos de disseminação (Eager Push e Plumtree), e na secção seguinte, 3.2, serão abordados os algoritmos de *Partial Membership* (HyParView e Cyclon).

3.1 Epidemic Dissemination Protocols (Broadcast)

3.1.1 Eager Push Gossip. Através do material leccionado nas aulas, e da dissertação [5], foi possível estudar e implementar o algoritmo.

Este algoritmo é do tipo *gossip*, o que permite não sub carregar o emissor e ao mesmo também condensa menos a *overlay network* quando é necessário enviar mensagens. Através do calculo do parâmetro de *fanout*, cujo o seu valor é o logaritmo do numero actual de vizinhos, é possível assim enviar uma mensagem cujo a probabilidade de chegar a todos os nós no *overlay* é bastante alta.

A função Init deste protocolo inicializa o estado do processo, e então aguarda pela notificação do protocolo de *membership* para que se possa utilizar o canal. Na função *EagerMessage* é possível então verificar a aplicação prática do envio apenas para um numero reduzido de nós a mensagem que se pretende disseminar na rede.

De modo a ilustrar melhor o comportamento do algoritmo, de seguida apresenta-se o pseudocódigo do mesmo (Algoritmo 1).

3.1.2 Plumtree. O Plumtree é uma implementação do tipo *gossip* e é uma junção parcial dos algoritmos de *gossip Eager Push* e *Lazy Push*, sendo assim mais equilibrado em termos de recursos e performance. No algoritmo é feita uma divisão dos seus *peers*, os *Eager Push Peers* e os *Lazy Push Peers*. Os *Eager Push Peers* são todos os nós que ainda não enviaram uma mensagem repetida e que portanto são beneficiados se for mantida uma comunicação bi-direcional entre eles, os nós pertencentes a este grupo comunicaram posteriormente através de uma forma do algoritmo Eager Push, que consiste em enviar a mensagem recebida para todos os *peers* que pertençam a este grupo. Os *Lazy Peers* são nós que devido a terem enviado uma mensagem repetida foram removidos dos *Eager Push Peers* e enviados por parte do receptor da mensagem repetida uma mensagem *Prune* que lhe indica que a mensagem era repetida e que portanto podem remover o nó receptor dos *Eager Push Peers* mantendo-se assim sincronizados. Os nós pertencentes a este grupo serão enviados mais tarde através de uma forma do algoritmo *Lazy Push* uma mensagem,

Algorithm 1 Eager Push Broadcast

```
1: function Init(myIp)
2:     myself ← myIp
3:     t ← ⊥
4:     neigh ← {}
5:     delivered ← {}
6:     channelReady ← False
7: end function
8: function EagerMessage(GossipMessage)
9:     if GossipMessage ∉ delivered then
10:        if GossipMessage.ttl > -1 then
11:            Trigger DeliverNotifications()
12:            neigh ← getNeigh()
13:            t ← ln(#neigh)
14:            if #neighbors > 0 then gossipTargets ← RandomSelection(t)
15:                for p ∈ gossipTargets do
16:                    Trigger Send(GossipMessage, p);
17:                end for
18:            end if
19:        end if
20:    end if
21: end function
22: function BroadcastRequest(BroadcastRequest)
23:    if channelReady then
24:        gossipMessage ← GossipMessage(request.MsgId, request.Sender, sourceProtocol. request.Payload, ttl)
25:        Trigger EagerMessage(gossipMessage)
26:    end if
27: end function
```

mensagem *Prune*, que apenas contém o ID da mensagem que o emissor possui, estas mensagens não têm obrigatoriamente de sair imediatamente após o nó ser movido para os *Lazy Peers* e por isso de forma a optimizar o processo são enviadas em batches através de um *dispatch protocol*.

O algoritmo implementado e usado nos testes das secções seguintes, foi passado para pseudocódigo (Algoritmo 2) de modo a permitir uma melhor percepção de como este funciona.

3.2 Partial Membership Protocols (Unstructed Overlay)

3.2.1 HyParView. A implementação deste algoritmo foi totalmente inspirada em [1], tendo este sido implementado de acordo com as instruções do artigo referido.

Inicialmente o protocolo inicia com uma mensagem de *Join* a um elemento aleatório da lista de contactos do novo nó que se quer juntar à rede. O envio desta mensagem tem origem na função *Init* onde também é lá que se inicializam os temporizadores (JoinTimer e ShuffleTimer). Após o envio da mensagem de *Join* o processo espera, durante um período de tempo, por uma mensagem de *JoinReply*, caso a mensagem chegue dentro do tempo definido no *JoinTimer*, este já faz parte da rede e continua o algoritmo normalmente como definido em [1], desligando o *JoinTimer*. Caso contrário, isto é, o tempo definido para o *JoinTimer* passou, é lançado um processo de selecção de um novo contacto de forma aleatória. Com este mecanismo, caso o novo nó apenas tenha um contacto, este ficará num ciclo infinito

Algorithm 2 PlumTree

```
1: function Init
2:     eagerPushPeers ← {}
3:     lazyPushPeers ← {}
4:     lazyQueue ← {}
5:     received ← {}
6:     channelReady ← False
7: end function
8: function BroadcastRequest(Request, sourceProtocol)
9:     if channelReady then
10:        gossipMessage ← GossipMessage(request.MsgId, request.Sender, sourceProtocol. request.Payload, ttl)
11:        Call eagerPush(gossipMessage, myself, currentProtocolId, channelId)
12:        lazyPush(gossipMessage, currentProtocolId)
13:        trigger DeliverNotification
14:        received ← ∪ request.MsgId
15:    end if
16: end function
.
17: function receive(ProtoMessage, HostFrom, sourceProto, channelId)
18:    if type(ProtoMessage) = "LazyMessage" then
19:        Call receive(ProtoMessage, HostFrom, sourceProto, channelId) //Prune Msg
20:    else
21:        Call receive(ProtoMessage, HostFrom, sourceProto, channelId)
22:    end if
23: end function
24: function dispatch
25:    for lazyMessage ∈ LazyQueue do
26:        trigger Send(lazyMessage, lazyMessage.Destination)
27:        LazyQueue ← LazyQueue \ lazyMessage
28:    end for
29: end function
30: function eagerPush(GossipMessage, Host, sourceProtocol, channelId)
31:    for h ∈ LazyQueue eagerPushPeers ∧ h ≠ myself do
32:        Send(GossipMessage, h) ∪ LazyMessage(h, message.MsgId, HostMyself,sourceProtocol)
33:        call dispatch()
34:    end for
35: end function
36: function lazyPush(GossipMessage, sourceProtocol))
37:    for h ∈ LazyQueue eagerPushPeers ∧ h ≠ myself do
38:        lazyQueue ← lazyQueue
39:    end for
40: end function
```

à espera que este aceite o sei pedido de *Join*, ao contrário do algoritmo definido em [1] onde caso o contacto não existisse ou estivesse em baixo, este processo terminava. Todo este novo mecanismo de *Join* de um novo nó está presente no Algoritmo 1 HyParView - Parte 1.

Algumas verificações adicionais poderiam ser implementadas para mitigar alguns dos problemas desta solução, algumas delas irão ser referidas na secção 5 como *future work*.

Após o protocolo de *Join* o algoritmo continua o seu processo de adesão do novo nó, espalhando mensagens à vizinhança do contacto de modo a estes saibam da existência de um novo nó na rede.

Todo o algoritmo implementado e usado nos testes das secções seguintes, foi passado para pseudocódigo (Algoritmos 3 a 6) de modo a permitir uma melhor percepção de como este funciona.

3.2.2 Cyclon. A implementação deste algoritmo, foi baseada no material de apoio do Cyclon que foi leccionado na cadeira, e também respeitando algumas propriedades encontradas na dissertação [5] e finalmente o artigo [2]

O primeiro passo de execução do algoritmo, consiste em, verificar se quando o processo que se pretende juntar ao *overlay* tem algum nó conhecido dentro deste, então este será a o primeiro vizinho do mais recente processo. Este processo ocorre na função init. Como explicado na secção anterior, nesta implementação do protocolo decidimos não validar se o contacto conhecido dentro do *overlay* ainda existia ou não, isto deve-se ao facto de como o protocolo recorre a um procedimento periódico, que é este o responsável pela manutenção do *overlay* em termos de frescura de de processos validos, como irá ser explicado mais adiante, garantindo assim que o contacto invalido será esquecido.

Como dito anteriormente, a função de *shuffle* que ocorre periodicamente é a responsável pela manutenção do *overlay*, inicialmente esta função incrementa a idade de todos os processos que se encontram na vizinhança de um nó, após este incremento, é então eleito o nó mais antigo dentro da vizinhança de um processo, e temporariamente retira-se o mesmo e procede-se à tentativa de comunicação com este.

A função apresentada na linha 29 no pseudo codigo do algoritmo, é chamada quando um nó vizinho do processo, pede uma amostra da vizinhança actual para poder actualizar a sua. Esta função assim como a que está representada na linha 26, terminam sempre numa chamada a um procedimento chamado *"Merge Views"*. Este procedimento fundamental do algoritmo, sendo este o responsável pela actualização da nova vizinhança do processo onde este procedimento foi despoletado, este consiste em analisar para cada nó na vizinhança que foi enviada por um vizinho, se o nó já existe na vizinhança, caso exista e este tiver uma *timestamp* mais recente, actualiza-se a vizinhança. Se o nó não existir e houver espaço na vizinhança, apenas se insere o nó nesta, caso contrario tenta-se escolher um nó que esteja contido na vizinhança recebida e na vizinhança do nó onde está a correr o procedimento, caso não seja possível, então finalmente escolhe-se um nó da vizinhança onde está a correr o procedimento, retira-se o mesmo, e insere-se o nó da vizinhança recebida.

De seguida apresenta-se o pseudocódigo do algoritmo, de maneira a ilustrar o funcionamento do mesmo (algoritmo 4).

4 EXPERIMENTAL EVALUATION

Algorithm 3 HyparView- Parte 1

```
1:  function Init
2:      n ← n ∈ contactsNodes
3:      Setup Periodic Timer ( JoinTimer, t1 )
4:      Setup Periodic Timer ( ShuffleTimer, t2 )
5:      Send( Join, n, myself )
6:  end function
7:
8:  function Receive(Join, newNode)
9:      if isfull( activeView ) then
10:         trigger dropRandomElementFromActiveView
11:     end if
12:     trigger addNodeActiveView(newNode)
13:     Send(JoinReply, newNode, myself)
14:     for n ∈ activeView and n ≠ newNode do
15:         Send(ForwardJoin, n, newNode, ARWL, myself)
16:     end for
17: end function
18:
19: function Receive(JoinReply, contactNode)
20:     trigger addNodeActiveView(newNode)
21:     Cancel Periodic Timer(JoinTimer, t)
22: end function
23:
24: function Receive(ForwardJoin, newNode, ttl, sender)
25:     if ttl==0 || #activeView==0 then
26:         trigger addNodeActiveView(newNode)
27:     else
28:         if ttl==PRWL then
29:             trigger addNodePassiveView(newNode)
30:         end if
31:         n ← n ∈ activeView and n ≠ sender
32:         Send(ForwardJoin, n, newNode, ttl-1, myself)
33:     end if
34: end function
35:
36: function dropRandomElementFromActiveView
37:     n ← n ∈ activeView
38:     Send(Disconnect, n, myself)
39:     activeView ← activeView \ {n}
40:     passiveView ← passiveView ∪ {n}
41: end function
42:
43: function addNodeActiveView(node)
44:     if node ≠ myself and node ∉ activeView then
45:         if isfull( activeView ) then
46:             trigger dropRandomElementFromActiveView
47:         end if
48:         activeView ← activeView ∪ {n}
49:     end if
50: end function
51:
52: function addNodePassiveView(node)
53:     if node ≠ myself and node ∉ activeView and node ∉
    passiveView then
54:         if isfull( passiveView ) then
55:             n ← n ∈ passiveView
56:             passiveView ← passiveView \ {n}
57:         end if
58:         passiveView ← passiveView ∪ {node}
59:     end if
60: end function
```

Algorithm 4 HyParView - Parte 2

```
1: function Receive(Disconnect, peer)
2:     if peer ∈ activeView then
3:         activeView ← activeView \ {peer}
4:         trigger addNodePassiveView(peer)
5:     end if
6: end function
7:
8: function Receive(Neighbor, peer, isPriority)
9:     if isPriority then
10:        trigger addNodeActiveView(peer)
11:    else
12:        if isfull(activeView) then
13:            Send(Reject, peer, myself)
14:        else
15:            passiveView ← passiveView \ {peer}
16:            trigger addNodeActiveView(peer)
17:        end if
18:    end if
19: end function
20:
21: function Receive(Reject, peer)
22:    passiveView ← passiveView \ {peer}
23:    trigger attemptPassiveViewConnection
24:    passiveView ← passiveView ∪ {peer}
25: end function
26:
27: function Receive(Shuffle, ttl, passiveViewSample, activeViewSample, peer)
28:    if ttl - 1 == 0 and #activeView > 1 then
29:        n ← n ∈ activeView and n ≠ peer
30:        Send(Shuffle, ttl - 1, n, myself)
31:    else
32:        sample ← trigger getSample(passiveView, #passiveViewSample)
33:        Send(ShuffleReply, sample, peer, myself)
34:    end if
35:    sample ← passiveViewSample ∪ activeViewSample
36:    trigger integrateElementsIntoPassiveView(samples)
37: end function
38:
39: function Receive(ShuffleReply, sample, peer)
40:    trigger integrateElementsIntoPassiveView(samples)
41: end function
42: function integrateElementsIntoPassiveView(sample)
43:    sample ← sample \ {activeView ∪ passiveView ∪ {myself} }
44:    if #passiveView + #sample > passiveViewMaxSize then
45:        trigger removeRandomElements(passiveView, #sample)
46:    end if
47:    passiveView ← passiveView ∪ sample
48: end function
49:
50: function JoinTimer
51:    n ← n ∈ contactsNodes
52:    Send(Join, n, myself)
53: end function
```

Algorithm 5 HyParView - Parte 3

```
1: function attemptPassiveViewConnection
2:     if #passiveView > 0 then
3:         if #activeView == 0 then
4:             isPriority = true
5:         else
6:             isPriority = false
7:         end if
8:         n ← n ∈ passiveView
9:         passiveView ← passiveView \ {n}
10:        trigger addNodeActiveView(newNode)
11:        Send(Neighbor, isPriority, n, myself)
12:    end if
13: end function
14:
15: function ShuffleTimer
16:    if #activeView > 0 then
17:        aSample ← trigger getSample(activeView, KA)
18:        pSample ← trigger getSample(passiveView, KP)
19:        n ← n ∈ activeView
20:        Send(Shuffle, aSample, pSample, n, myself)
21:    end if
22: end function
23:
24: function getSample(set, size)
25:    retult ← { }
26:    for i=0 to size-1 do
27:        n ← {n} ∈ set
28:        result ← {result} ∪ n
29:    end for
30: end function
```

4.1 Methodologies

Nesta avaliação utilizamos o Cluster do Departamento de Informática da Faculdade de Ciências e Tecnologias da Universidade Nova de Lisboa [6]. Neste Cluster foram utilizados 2 nós (nós 5 e 6) cujo as suas características podem ser analisadas no anexo 1. Tal como referido nas secções anteriores, foram implementados quatro algoritmos, sendo dois de *Partial Membership* e outros dois de *Gossip Dissemination*. Como tal, foram feitos quatro testes, que combinados, geram todas as combinações dos algoritmos a funcionarem uns com os outros. As combinações foram as seguintes:

- Eager Push + HyParView
- Eager Push + Cyclon
- Plumtree + HyParView
- Plumtree + Cyclon

Para todas as combinações as configurações são iguais, assim como a quantidade nós na rede. A quantidade escolhida foi de 100 nodes. As configurações fixas e todos os testes foram as mesmas utilizadas em [1] [3], com:

- Tamanho máximo da Active View = 5
- Tamanho máximo da Passive View = 30
- Shuffle Timer com 15 segundos de intervalo
- Join Timer com 1 segundo de espera
- k=6
- c=1
- ARWL=6
- PRWL=3
- ka=3
- kp=4

Algorithm 6 Cyclon

```
1:  function Init(contactNode, timer)
2:      trigger CreateNotification(channelId)
3:      if contactNode ≠ ⊥ then
4:          neigh ← neingh ∪ { }
5:      end if
6:      Setup Periodic Timer Shuffle(timer)
7:  end function
8:  function Shuffle
9:      if #neigh > 0 then
10:         for (p, age) ∈ neigh do
11:             neigh ← neingh \ (p,age) ∪ (p,age +1)
12:         end for
13:         oldest ← GetOldest(neigh)
14:         neigh ← neigh \ oldest
15:         sample ← randomSubset(neigh)
16:         Trigger Send (ShuffleRequest,p,sample ∪ {(myself,0)
    })
17:     end if
18: end function
19: function Receive(ShuffleReply, s, PeerSample)
20:     Call mergeViews(peerSample, sample)
21: end function
22: function Receive(ShuffleRequest, s, PeerSample)
23:     temporarySample ← temporarySample ∪ neigh
24:     Trigger Send(ShuffleReply, s, temporarySample)
25:     Call mergeViews(peerSample,temporarySample)
26: end function
27: function Merge Views(peerSample, mySample)
28:     for (p, age) ∈ peerSample do
29:         if (p', age') ∈ neigh ∧ p = p' then
30:             if age'> age then
31:                 neigh ← (neigh \(p', age')) ∪ {(p, age) })
32:             end if
33:         else
34:             if #neigh< sampleSize then
35:                 neigh ← neigh ∪ {(p, age)})
36:             else
37:                 (x,age'):(x,age') ∪ neigh ∧ (x, age") ∪ mySam-
    ple
38:                 if #neigh< sampleSize then
39:                     (x, age') ← (x,age'):(x, age') ∪ neigh
40:                 end if
41:                 neigh ← (neigh \ (x, age')) ∪ {(p, age)})
42:             end if
43:         end if
44:     end for
45: end function
```

Porém, variámos para cada teste de combinação de algoritmos, o tamanho do payload, tendo feito um teste com payload grande (10000 bytes) e outro pequeno (10 bytes). Também variámos a taxa de transmissão, utilizando uma taxa grande de 3 segundo e outra mais pequena de 1 segundo. Com todas estas variações de algoritmos e propriedades, chegamos a um total de 16 testes, considerando todas as suas possíveis junções.

Para além de tudo isto, ainda adicionamos testes isolados aos algoritmos de *Partial Membership* de acordo com as características anteriores e configurações variáveis de payload e taxa de transmissão, o que nos adiciona aos anteriores 16 testes, mais 8, dando um total de 24 testes que iremos demonstrar nas subsecções seguintes.

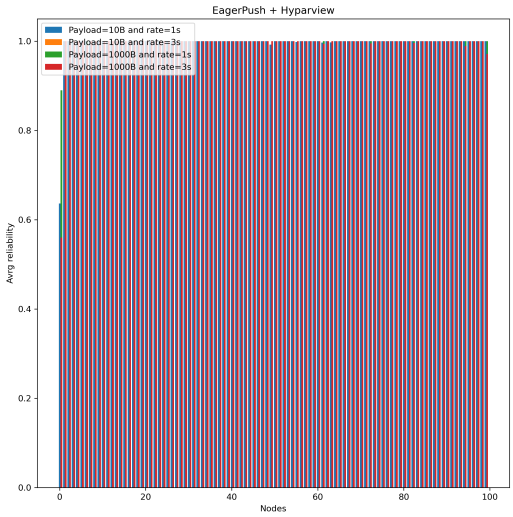

Figure 1: EagerPush + Hyparview (Reliability)

4.2 Results

Nesta secção iremos falar mais detalhadamente, sobre os resultados obtidos na experimentação dos algoritmos no Cluster do DI.

Como dito na secção anterior, os 16 testes estão divididos por combinações de protocolos, por exemplo do tipo E + C, onde o protocolo é também identificado pela sua letra inicial. Os testes efectuados aos protocolos foram de *reliability*, *latency*, *total sent messages*, e *total failed messages* em cada nó.

Em cada gráfico é possível também observar que a cada cor correspondem as parametrizações de payload e bit rate usados.

Como se verifica nos resultados adiantes, devido a um erro durante a implementação do Plumtree, os resultados usando este algoritmo não foram os esperados.

4.2.1 Reliability. Os resultados obtidos sobre a confiabilidade das mensagens envidas na aplicação foram bastante bons e com resultados na ordem dos 100%, excepto com a junção do Plumtree com o Cyclon, onde, na maioria dos nós na rede, se verifica uma discrepância fora do comum.

4.2.2 Sent Messages. Nos gráficos relativos ao envio de mensagens observa-se que na sua maioria os protocolos têm tendência a enviar menos mensagens quando o *payload* destas aumenta, no entanto o Plumtree destaca-se não mostrando uma grande diferença com a alteração deste parâmetro, algo que será discutido em maior detalhe mais à frente.

4.2.3 Failed Messages. Nas *Failed Messages*, houve um pequeno erro na geração dos gráficos, onde por lapso, no eixo vertical, os valores não estão ordenados. Porém dá para ter uma percepção geral dos resultados, principalmente nos gráficos onde o algoritmo de *membership* utilizado foi o HyParView, onde se nota claramente que 90% das barradas, independentemente das configurações utilizadas, estão com 0 mensagens falhadas. No entanto utilizando o Cyclon já se verifica uma maior discrepância nos resultados.

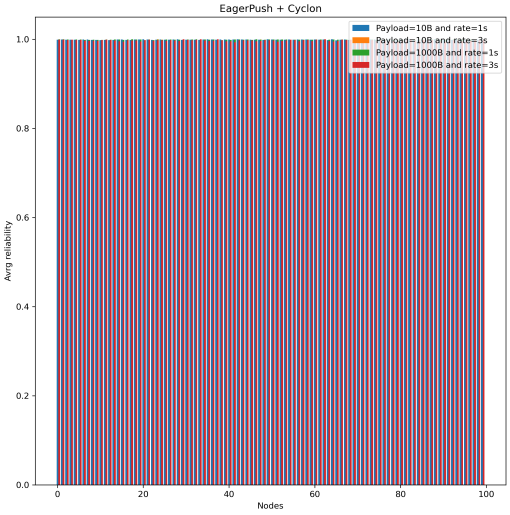

Figure 2: EagerPush + Cyclon (Reliability)

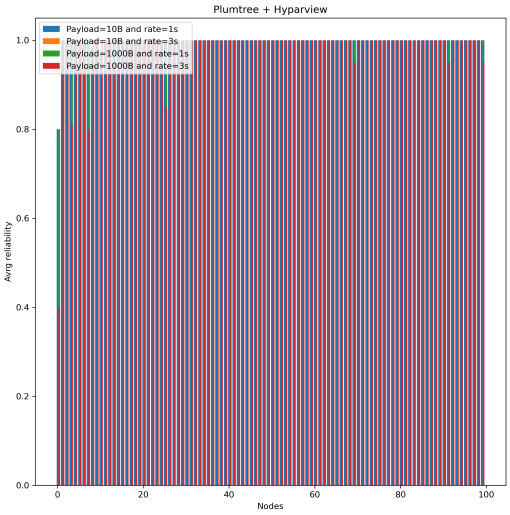

Figure 3: Plumtree + Hyparview (Reliability)

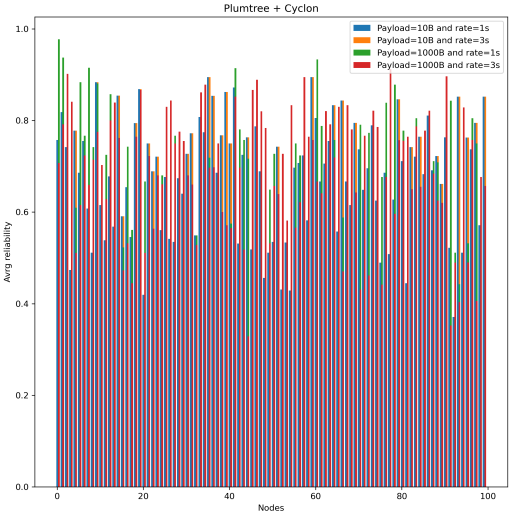

Figure 4: Plumtree + Cyclon (Reliability)

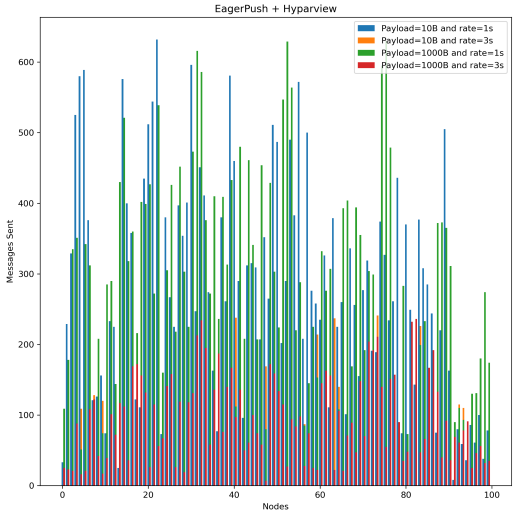

Figure 5: EagerPush + Hyparview (Sent Messages)

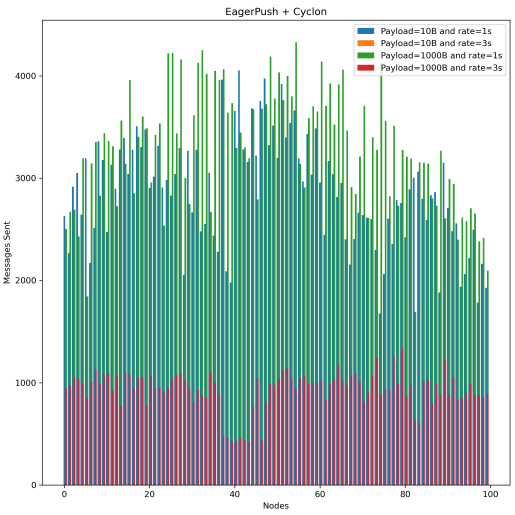

Figure 6: EagerPush + Cyclon (Sent Messages)

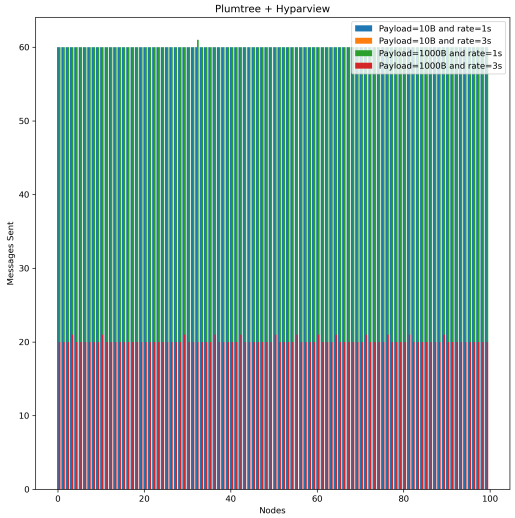

Figure 7: Plumtree + Hyparview (Sent Messages)

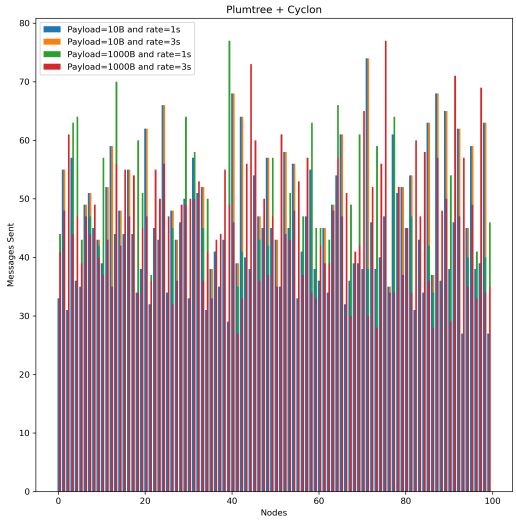

Figure 8: Plumtree + Cyclon (Sent Messages)

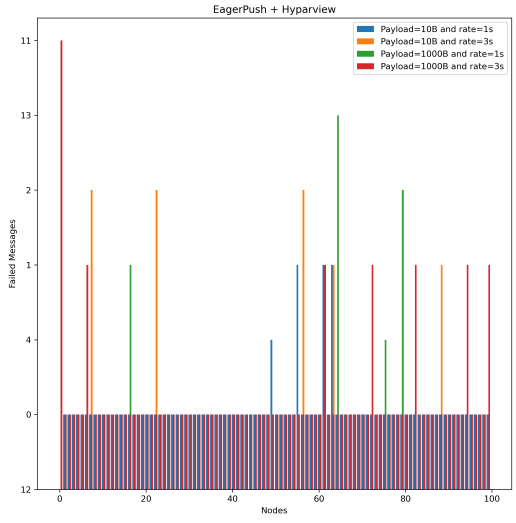

Figure 9: EagerPush + Hyparview (Failed Messages)

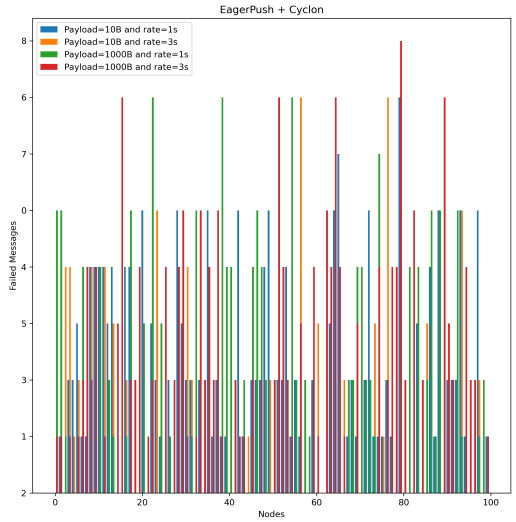

Figure 10: EagerPush + Cyclon (Failed Messages)

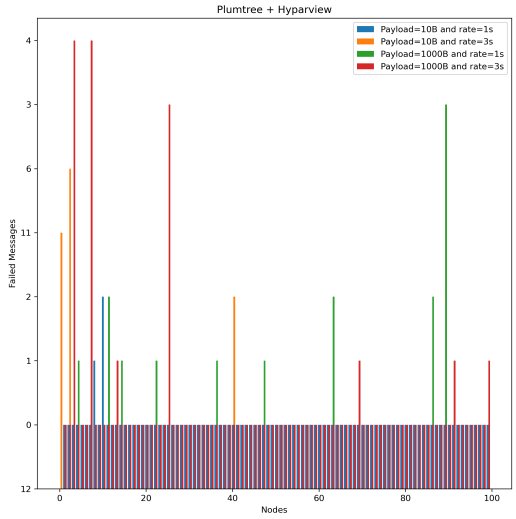

Figure 11: Plumtree + Hyparview (Failed Messages)

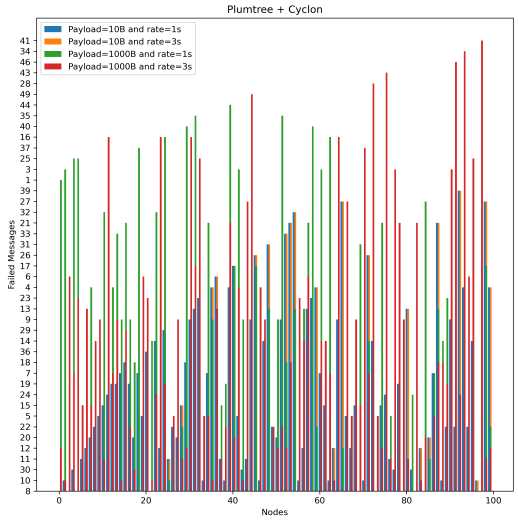

Figure 12: Plumtree + Cyclon (Failed Messages)

4.2.4 Latency. A latência media do algoritmo de *broadcast* foi calculada da seguinte maneira: A partir do momento em que uma mensagem é enviada pela primeira vez para a rede, até chegar finalmente ao último processo que a recebeu. Como dito anteriormente, é possível verificar que houve problemas a medir a latência quando o algoritmo usado no teste era o Plumtree.

Parameters / Algotithm	E + H	E + C	P + H	P + C
Payload=10B, BR=1s	9.12s	61.27s	0.0s	0.0s
Payload=10B, BR=3s	5.06s	23.52s	0.0s	0.0s
Payload=10MB, BR=1s	10.26s	112.67s	0.0s	0.0s
Payload=10MB, BR=3s	3.74s	25.75s	0.0s	0.0s

4.3 Discussion

De acordo com os dados apresentados nos gráficos, é possível verificar que no gráfico da figura 7 (Plumtree + HyParView), que o total de mensagens enviadas pelos nós correspondem ao esperado. Independentemente do tamanho do *payload*, e sabendo que cada nó esteve a executar durante 60 segundos, o esperado utilizando uma taxa de transmissão de 1 e 3 segundos, deu o

resultado respectivo de 60 e 20 como esperado. No entanto, nas outras combinações de algoritmos, verificou-se uma grande duplicação de mensagens não correspondendo ao valor esperado. Estes resultados podem ser devidos a vários factores, tais como *bugs* na implementação dos algoritmos ou no processamento dos *logs*.

Relativamente ao teste da totalidade de mensagens falhadas é possível verificar que usando as combinações com o HyParView a taxa de mensagens falhadas é pouco representativa, porem na combinação Plumtree + Cyclon é possível verificar que houve um grande número de mensagens falhadas. No Eager Push + Cyclon apesar de se verificar algum numero de mensagens falhadas, a sua percentagem relativamente ao total de mensagens enviadas (60 para um *rate* de 1s e 20 para um rate de 3s) é inferior.

No caso da *reliability*, é possível verificar que para todas as combinações de algoritmos excepto o Plumtree + Cyclon, os resultados foram bastante positivos, tendo apenas uma fracção muito pequena de nós onde a reliability foi inferior ao esperado. No caso do Plumtree + Cyclon é possível verificar uma instabilidade na maioria dos nós da rede possivelmente devido a problemas de implementação, ou possivelmente até mesmo devido a algumas instabilidades na rede do *cluster*.

Por último, relativamente à latência é possível verificar que o HyParView relativamente ao Cyclon demonstra uma menor latência independentemente das configurações. Mais não podemos concluir devido aos dados estatísticos do Plumtree, que em todos os testes deu uma media de 0.0s, o que impossibilitou uma comparação mais pormenorizada para com as outras combinações de algoritmos.

5 CONCLUSIONS

Concluindo este relatório, através do desenvolvimento deste projecto foi possível aplicar os conhecimentos teóricos adquiridos nas aulas e pôr os mesmos em prática e testa-los numa situação real como no *cluster* do Departamento de Informática.

Os protocolos à base de gossip tem um papel fundamental, no sentido em que deixa de ser necessário fazer *flood* na rede para assegurar que todos os processos nesta, receberam efectivamente a mensagem. Sendo assim este tipo de protocolos, baseia-se na evidência de que é apenas necessário enviar um reduzido numero de mensagens para assegurar a disseminação efectiva de uma mensagem na rede, aumentando assim a disponibilidade e fazendo menos pressão no envio de mensagens num processo.

Relativamente aos protocolos de *Membership*, estes desempenham um papel fundamental, pois tem como papel principal a manutenção e organização da *network overlay* que liga todos os processos, com o objectivo de facultar ao algoritmo de *broadcast* uma vista parcial da rede para ser possível a disseminação de mensagens.

Foi ainda possível concluir, pelos dados estatísticos, que existem certas combinações de protocolos que funcionam melhor do que outras, e que, por exemplo quando o *payload* era aumentado havia uma condensação/pressão maior na rede, levando assim a uma disponibilidade menor, o que é suportado pelos dados estatísticos.

O *bitrate* é ainda outro factor que é determinante na performance dos testes, a escolha dos diferentes *bitrates* utilizados no teste, não foi a mais adequada visto que a diferença entre eles era ínfima.

Ainda assim, houve algumas limitações na implementação dos algoritmos, levando assim a alguma dificuldade na extracção dos dados estatísticos, mas futuramente seria algo que poderia ser melhorado, como por exemplo no algoritmo EagerPush, relativamente ao tempo em que este devia guardar as mensagens, por razões de complexidade espacial e finalmente aumentar o número de testes utilizando outras combinações de parâmetros com o fim de aumentar a diversidade das métricas avaliadas.

REFERENCES

[1] João Leitão, José Pereira, and Luís Rodrigues. *HyParView*: a membership protocol for reliable gossip-based broadcast.
`https://asc.di.fct.unl.pt/ jleitao/pdf/dsn07-leitao.pdf`
[2] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. *Cyclon*: Inexpensive membership management for unstructured p2p overlays.
`http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.133.4965`
[3] João Leitao, José Pereira, and Luís Rodrigues. *Plumtree*: Epidemic broadcast trees.
`https://www.gsd.inesc-id.pt/ ler/reports/srds07.pdf`
[4] *Babel*
`https://asc.di.fct.unl.pt/ jleitao/babel/help-doc.html`
[5] João Carlos Antunes Leitão. Dissertação de Mestrado.
`https://asc.di.fct.unl.pt/ jleitao/pdf/masterthesis-leitao.pdf`
[6] Cluster DI.
`https://cluster.di.fct.unl.pt/`

Appendices

.1 Configuration file

```
port=10000
interface=eth0
address=127.0.0.1
sample_size=3
sample_time=3000
protocol_metrics_interval=1000
channel_metrics_interval=1000
payload_size={PAYLOAD_SIZE}
prepare_time=5
run_time=60
cooldown_time=5
broadcast_interval={INTERVAL_RATE}
broadcast={BROADCAST_ALG}
membership={MEMBERSHIP_ALG}
activeMembershipSize=5
passiveMembershipSize=30
shuffleTime=15000
joinTime=1000
k=6
c=1
arwl=6
prwl=3
ka=3
kp=4
n=100
```

.2 Docker info

```
Client:
 Debug Mode: false
Server:
 Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
 Images: 380
 Server Version: 19.03.6
 Storage Driver: overlay2
  Backing Filesystem: extfs
  Supports d_type: true
  Native Overlay Diff: true
 Logging Driver: json-file
 Cgroup Driver: cgroupfs
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald
       json-file local logentries splunk syslog
 Swarm: active
  NodeID: wpj20zepewpo13plsneygvk03
  Error: rpc error: code = Unknown desc = The
       swarm does not have a leader. It's
       possible that too few managers are online.
       Make sure more than half of the managers
       are online.
  Is Manager: true
  Node Address: 172.30.10.108
  Manager Addresses:
   172.30.10.107:2377
   172.30.10.108:2377
 Runtimes: runc
 Default Runtime: runc
 Init Binary: docker-init
 containerd version:
 runc version:
 init version:
 Security Options:
  apparmor
  seccomp
   Profile: default
 Kernel Version: 5.3.0-64-generic
 Operating System: Ubuntu 19.10
 OSType: linux
 Architecture: x86_64
 CPUs: 24
 Total Memory: 62.87GiB
 Name: node8
 ID:
  XFNZ:7CU6:2VDG:M4K7:XRMR:WE6A:63WW:FORC:VE2V:MMRA:LDBI:KNVE
 Docker Root Dir: /var/lib/docker
 Debug Mode: false
 Registry: https://index.docker.io/v1/
 Labels:
 Experimental: false
 Insecure Registries:
  127.0.0.0/8
 Live Restore Enabled: false
```