Eficiência e Eficácia: Redes de Overlay Não-Estruturadas para Protocolos de Broadcast Baseado em Gossip Epidémico

Filipe de Luna*
f.luna@campus.fct.unl.pt
MIEI, DI, FCT, UNL

Rafael Gameiro†
rr.gameiro@campus.fct.unl.pt
MIEI, DI, FCT, UNL

Rafael Sequeira‡
rm.sequeira.@campus.fct.unl.pt
MIEI, DI, FCT, UNL

ABSTRACT

A entrega de mensagens em broadcast pode ser realizada recorrendo a um conjunto diverso de métodos e variando desde soluções centralizadas a sistemas peer-to-peer. Dentro destes últimos, os protocolos de gossip (ou epidémicos) surgem como uma alternativa apelativa a métodos mais tradicionais como, por exemplo, a inundação de canais. Estes protocolos variam entre si, utilizando diversas estratégias para atingir o mesmo fim e dependem, geralmente, de um protocolo que tem como responsabilidade manter uma rede de overlay de forma a permitir a troca de mensagens entre vários processos.

Neste relatório são estudados dois protocolos de broadcast baseado em gossip (Plumtree e Adaptive Gossip) e dois protocolos de overlay não estruturado (Cyclon e HyParView). As respectivas implementações são especificadas sobre a forma de pseudo-código e são apresentados resultados derivados de testes experimentais realizados sobre as suas diversas combinações. Por fim, são inferidas conclusões acerca das vantagens (e desvantagens) das estratégias por si utilizadas, sobre o efeito do saturação dos recursos disponíveis e o impacto das suas diferentes parametrizações.

ACM Reference Format:
Filipe de Luna, Rafael Gameiro, and Rafael Sequeira. . Eficiência e Eficácia: Redes de Overlay Não-Estruturadas para Protocolos de Broadcast Baseado em Gossip Epidémico. In . ACM, New York, NY, USA, 11 pages.

1 INTRODUÇÃO

Protocolos de gossip, ou epidémicos, têm surgido como uma estratégia de implementação de primitivas de broadcast escaláveis e confiáveis para redes peer-to-peer[7]. Num protocolo de gossip, quando um nó pretende propagar uma mensagem por broadcast, este selecciona um conjunto de nós aleatórios que pertençam à sua vizinhança para enviar-lhes a mensagem. Após esta mensagem ser recebida por estes, é de novo propagada e o mesmo procedimento repetido por assim em diante. Estes protocolos são altamente resilientes e permitem uma distribuição da carga na rede por todos nós pertencentes ao sistemas.

Contudo, os protocolos de gossip, por si só, não garantem a comunicação por broadcast na rede, uma vez que estes protocolo são apenas responsáveis por uma camada de rede, dedicada à troca de mensagens entre um pequeno número de processos por si conhecidos. A obtenção e gestão desta amostra, denominada por vizinhança, é da inteira responsabilidade do protocolo de overlay situado na camada abaixo.

O protocolo de overlay garante que, para uma rede com um número elevado de nós, cada nó, possui a sua própria vizinhança composta por outros nós. A intersecção das vizinhanças de cada nó resulta num grafo conexo, garantindo assim a cobertura de toda a rede.

Tipicamente uma rede de overlay pode ser classificada como estruturada ou não estruturada, com cada um destes tipos demonstrando um melhor desempenho em diferentes cenários. Redes de overlay não estruturadas são particularmente interessantes visto serem fáceis de construir e de manter.

Neste relatório iremos apresentar um estudo sobre protocolos de broadcast baseados em gossip e protocolos de redes overlay não estruturadas. Todos os protocolos, bem como a sua implementação irão ser descritos de forma detalhada, sendo incluindo o seu respectivo pseudocódigo. Também serão apresentados os resultados dos testes de desempenho e, finalmente, será realizada uma análise que tem por objectivo principal determinar quais as combinações de protocolos que apresentam melhor desempenho em correspondência com as várias métricas a serem analisadas.

Os testes realizados consistiram em correr, num cluster computacional, todas as combinações entre os vários protocolos elaborados para este trabalho e avaliar o seu desempenho com base em factores como o número de mensagem e bytes enviados/recebidos, latência, fiabilidade e cobertura de broadcasting entre os nós da rede.

Após efectuar as avaliações experimentais, concluímos que existem vantagens inerentes em utilizar técnicas de lazy push, contribuindo significativamente para a não saturação de uma rede ao trocar mensagens com payloads elevados. Adicionalmente, verificámos os efeitos positivos proporcionados por uma (relativamente) grande vizinhança para a tolerância a falhas.

O resto do documento está estruturado da seguinte forma: A secção 2 explica conceitos fundamentais à compreensão do resto do documento e um breve estudo de soluções baseadas em protocolos semelhantes aos que foram por nós desenvolvidos, sendo directamente relacionados com o nosso trabalho. Por fim, é possível encontrar uma explicação mais detalhada acerca do que são protocolos baseados em gossip epidémico e protocolos de overlay; Uma descrição da forma como foi realizada a implementação dos algoritmos, por nós escolhidos, é feita na Secção 3; Na Secção 4, são descritas as avaliações experimentais bem como o método utilizado para a sua realização, os seus resultados e a interpretação e discussão dos mesmos. Para terminar, a Secção 5 apresenta as conclusões que foram possíveis extrair durante o decorrer da realização deste trabalho, bem como dos resultados observados nos testes.

*Aluno number 48425. Filipe ficou encarregado de implementar os protocolos HyParView e Adaptive Gossip.
†Aluno number 50677. Rafael foi responsavel pela implementação do Plumtree.
‡Aluno number 50002. Rafael foi responsável pela implementação do Cyclon.

ASD 20/21, 2020, FCT, NOVA
November 17, 2020, Portugal.

2 TRABALHO RELACIONADO

Nesta secção iremos apresentar uma explicação sobre o que são protocolos de overlay, em conjunto com alguns exemplos. De seguida iremos dar uma definição para os protocolos baseados em gossip e uma descrição do seu comportamento. Na mesma sub-secção iremos expor alguns exemplos de protocolos baseados em gossip, que foram feitos no mesmo contexto em que este relatório se insere.

2.1 Protocolos de Overlay

Protocolos de overlay implementam o conceito de vizinhança e definem como estas se organizam para criar uma rede conexa. Estas redes organizam os vários nós segundo um grafo, podendo este ser estruturado ou não estruturado. Caso o grafo seja estruturado, permite que haja uma pesquisa mais eficiente entre os vários nós [9]. Se o grafo for não estruturado, os nós são organizados de forma aleatória, normalmente de forma hierárquica através de camadas.

Os protocolos têm assim como objectivo criar uma abstracção sobre a camada física de uma rede. Nesta nova camada, cada processo estabelece conexões com outros processos, permitindo assim, haver trocas de mensagens, não só por parte da mesma, mas também das camadas acima. Abaixo são apresentados alguns exemplos deste tipo de protocolos e as suas propriedades mais relevantes:

Scamp[3] é um protocolo reactivo onde os nós mantém registo da sua vizinhança excepto quando algum evento externo leva-os a mudar de vizinhança. O protocolo garante que todos os processos convergirem para ter um vizinhança cujo tamanho é o uma relação logarítmica com o numero total de nós do sistema sem que qualquer processo esteja ciente do seu número.

CellFarm[5] este algoritmo usa uma abstracção de nós - denominada por nó virtual - que representa um conjunto de nós físicos. Estas abstracções são identificadas por um identificador que é partilhado com a camada acima. O protocolo gere de forma dinâmica os nós virtuais e físicos que lhes pertencem. Todos os nós físicos num nó virtual formam um nó conexo e cada nó físico mantém ligações a outros nós virtuais, garantindo a conectividade geral, distribuição de carga e tolerância a falhas.

Os dois principais métodos de propagação de mensagens têm o nome de eager e lazy push. O primeiro simplesmente envia uma mensagem enquanto o segundo envia uma notificação a avisar que possui uma mensagem que pode enviar assim que lhe for pedido. Eager pushes permitem uma difusão mais rápida ao custo de maior redundância de dados, o que se torna exponencialmente pior com grandes payloads. Lazy pushes combatem este problema mas introduzem maiores RTTs e são mais susceptíveis a falhas [8].

2.2 Broadcast baseado em Gossip

A ideia principal para a concepção um protocolo de gossip consiste em garantir que todos os nós pertences a uma rede contribuam de forma equilibrada na disseminação mensagens. Para que tal seja possível, quando um nó pretende enviar uma mensagem, selecciona um conjunto de nós aleatórios da sua vizinhança para a receber e propagar. Este processo é repetido por cada nó que recebe uma mensagem, até todos os nós na rede a terem recebido. No entanto, caso um nó receba mais do que uma vez a mesma mensagem, este deverá descartá-la. Para que este procedimento seja possível, é necessário que todos os nós possuam informação acerca da sua vizinhança e quais as mensagens que já receberam. Alguns exemplos de protocolos que não foram implementados no nosso trabalho são:

CREW[10] utiliza um sistema de partilha de meta-dados acerca de recursos para implementar lazy push gossip. Estes meta-dados são disseminados o mais rápido possível, de forma a todos os nós poderem começar a pedir os mesmos imediatamente e de forma agressiva. Isto tem o efeito de reduzir drasticamente o overhead afectando ao mínimo a latência. Os nós utilizam, também, um sistema de estimação da sua largura de banda disponível, de forma a maximizar a velocidade das transacções efectuadas.

NEEM[4] tem como objectivo principal reduzir o congestionamento na rede. A implementação divide-se em três módulos: um protocolo epidémico básico, um módulo de congestionamento de controlo de largura de banda optimizado para TCP/IP e um gestor de buffer inteligente na fronteira da rede para descartar mensagens em caso de overflow.

3 IMPLEMENTAÇÃO

Todos os algoritmos foram implementados usando a linguagem Java 11, com auxilio da framework Babel, desenvolvida no laboratório da NOVA LINCS por Pedro Fouto, Pedro Ákos Costa e João Leitão.

Em cada uma das subsecções seguintes é apresentada uma breve descrição dos algoritmos implementados, o que foi alterado relativamente à sua versão original e, por fim, o seu pseudo-código.

3.1 Protocolos de broadcast epidémico

Relativamente aos protocolos de disseminação epidémica, foram escolhidos o Adaptive Gossip e o Plumtree. Ambos os algoritmos são ilustrados nas secções 3.1.1 e 3.1.2 respectivamente.

3.1.1 ***Adaptive Gossip.*** O Adaptive Gossip consiste em dividir a camada de broadcast em duas, com a segunda sendo transparente à primeira. A primeira, corre um algoritmo básico de gossip (1) e, a segunda, o *Payload Scheduler*, que decide como e quando enviar uma mensagem. O *Payload Scheduler* (2) permite decidir, de forma dinâmica, realizar um *eager* ou um *lazy* push.

O *Payload Scheduler* pode ser dividido em dois módulos, o *Lazy Point-to-Point* e o *Transmission Strategy* (4). O primeiro, está encarregue da comunicação entre as duas camadas do protocolo, de pedir mensagens ou responder a pedidos de mensagens e recorre ao segundo, o módulo *Transmission Strategy*, para este decidir se uma mensagem irá ser ou não enviada/pedida.

Uma vez que a framework utilizada (*Babel*), nos obriga a colocar apenas um protocolo por camada, esta camada teve de ser emulada. Todas as mensagens recebidas na camada de gossip são imediatamente passadas para o *Payload Scheduler*, para simular a camada. De forma idêntica, dado que a camada do *Payload Scheduler* não consegue enviar mensagens, este comportamento é simulado ao chamar uma função da camada de gossip que faz envio por ela.

O *Payload Scheduler* recebe um evento despoletado por um temporizador periódico presente na camada de gossip para pedir ao módulo *Transmission Strategy*, uma nova mensagem, do tipo pedido de mensagem por *ID*, para enviar. Este pedido bloqueia a execução, pois é utilizada uma pilha bloqueante para gerir os pedidos.

Algorithm 1 Gossip-Layer

```
1:  Init
2:      received ⟵ {};  // Set of received messages
3:
4:  Procedure Multicast(data)
5:      Call Flood(generateMsgID(), data, 0);
6:
7:  Procedure Flood(id, data, rounds)
8:      Trigger Deliver(data);
9:      received ⟵ received ⋃ {id};
10:     If (rounds < MAX_ROUNDS) Then
11:         Forall p ∈ getNeighbours() \ {self} Do
12:             SendToScheduler(id, data, rounds + 1, p);
13:
14:
15:
16: // Upon receiving any message
17: Upon ReceiveMessage(msg, p) Do
18:     Call ForwardToScheduler(msg, p);
19:
20: Upon ReceiveFromScheduler(id, data, rounds) Do
21:     Call Flood(id, data, rounds);
22:
```

Algorithm 2 Payload Scheduler (1)

```
1:  Init
2:      cached ⟵ ⊥;  // Cached messages
3:      known ⟵ {};  // Known messages
4:      Setup Periodic Timer IWantTimer(50);
5:
6:  Upon SendToScheduler(id, data, rounds, p) Do
7:      If (IsEager(id, data, rounds, p)) Then
8:          // Forward message through upper layer.
9:          Trigger ForwardFromScheduler(Msg(id, data, rounds), p));
10:     Else
11:         cached[id] ⟵ {data, rounds};
12:         Trigger ForwardFromScheduler(IHaveMsg(id), p);
13:
14:
15: Upon Receive(IHaveMsg(id), p)) Do
16:     If (id ∉ known) Then
17:         Trigger Queue(id, p);
18:
19:
```

Algorithm 3 Payload Scheduler (2)

```
1:  Upon Receive(Msg(id, data, rounds), p) Do
2:      If (id ∉ known) Then
3:          known ⟵ known ⋃ id;
4:          Trigger Clear(id);
5:
6:      Trigger ReceiveFromScheduler(id, data, rounds, p);
7:
8:  Upon Receive(IWantMsg(id), p) Do
9:      If (id ∉ known) Then
10:         {data, rounds} ⟵ cached[id];
11:         Trigger ForwardFromScheduler(Msg(id, data,
    rounds), p);
12:
13:
14: Upon Timer IWantTimer Do
15:     {id, p} ⟵ ScheduleNext();
16:     Trigger ForwardFromScheduler(IWantMsg(id), p);
17:
```

Algorithm 4 Transmission Strategy Module

```
1:  Init
2:      queue ⟵ {};  // Queued messages
3:
4:  Upon IsEager(id, data, rounds, p) Do
5:      If (mode = "Pure Eager Push") Then
6:          Return True;
7:
8:      If (mode = "Pure Lazy Push") Then
9:          Return False;
10:
11:     If (mode = "TTL") Then
12:         Return rounds > MAX_ROUNDS;
13:
14:     // "Undefined mode."
15:
16: Upon Queue(id, p) Do
17:     queue ⟵ queue ⋃ QueueItem(id, p);
18:
19: Upon Clear(id) Do
20:     queue ⟵ queue \ QueueItem(id, p);
21:
22: Upon ScheduleNext() Do
23:     Return TakeFirst(queue);
24:
```

3.1.2 ***Plumtree.*** Este protocolo[6] utiliza uma combinação entre eager push gossip e lazy push gossip para assim gerar uma estrutura em árvore sobre um rede de overlay não estruturada. Esta árvore depois é utilizada para fazer a propagação de mensagens. Dependendo das respostas recebidas por cada membro da vizinhança, os próximos broadcasts de mensagens podem ser feitos utilizando eager ou lazy push. Para além disso, o lazy push também pode ser utilizado para detectar e efectuar a recuperação de partições que a árvore tenha. O pseudo-código que expressa a implementação feita encontra-se ilustrado abaixo.

Comparando este pseudo-código com o que é apresentado no paper do Plumtree podemos concluir que o modo de operação nas várias funções é idêntico. Contudo, e de maneira a facilitar a leitura do algoritmo, foram criadas estruturas de dados adicionais. Por exemplo, a operação relativa à recepção de uma mensagem graft(**??**) apresentava de uma forma pouco explicita como era era obtido o conteúdo da mensagem a enviar mais tarde. De maneira a simplificar o funcionamento da função, utilizámos um estrutura adicional que guarda o conteúdo de uma mensagem recebida durante um certo intervalo de tempo. Desta forma quando é feito o envio de uma mensagem gossip (**??**) percepciona-se com maior facilidade onde e como é obtido o conteúdo da mesma. Também na função de NeighbourDown (linha 19 algoritmo 4) de maneira a explicitar o seu funcionamento removemos o nó de todas as estruturas que possam

provocar algum impacto no algoritmo. Como tal, na LazyQueue são removidas todas as entradas onde a mensagem IHAVE (a ser enviada) tem por destino esse processo.

Algorithm 5 Plumtree (1)

```
1:  Interface
2:      Indications
3:          Broadcast(mID,m)
4:
5:
6:  State
7:      received;  // set with the ID of received messages
8:      rcvContent;  // map with received msg content
9:      // map of messages that did no reach the peer
10:     missing;
11:     // set of messages that will be eventually sent to its peers
12:     lazyQueue;
13:     // set of nodes that receive messages through eager push
14:     eagerPushPeers;
15:     // set of nodes that receive messages through lazy push
16:     lazyPushPeers;
17:     // map of timers related to missing message requests
18:     timers;
19:
20: Procedure dispatch
21:     announcements ⟵ setPolicy(lazyQueue);
22:     Send(annoucements);
23:     lazyQueue ⟵ lazyQueue \ annoucements;
24:
25: Procedure EagerPush(mID, m, round)
26:     Forall p ∈ eagerPushPeers: p ≠ myself Do
27:         Send(GOSSIP, mID, myself, p, round, m);
28:
29:
30: Procedure LazyPush(mID, round)
31:     Forall p ∈ eagerPushPeers: p ≠ myself Do
32:         lazyQueue ⟵ (IHAVE(mID, myself, p, round));
33:
34:     Call dispatch();
35:
36: Upon Init Do
37:     received ⟵ {};
38:     rcvContent ⟵ [;]
39:     missing ⟵ [;]
40:     lazyQueue ⟵ {};
41:     eagerPushPeers ⟵ {};
42:     lazyPushPeers ⟵ {};
43:     timers ⟵ [;]
44:
45: Upon Broadcast(mID, m) Do
46:     Call EagerPush(mID, m, 0);
47:     Call LazyPush(mID, 0);
48:     Trigger Deliver(m);
49:     received ⟵ received ⋃ {mID};
50:     If (∃ (id,node,r) ∈ missing :id=mID) Then
51:         Cancel Timer (MISSING, mID);
52:
53:     rcvContent[mID] ⟵ {m};
54:     Setup Timer (MESSAGE, mID,$timeout_3$);
55:
```

Algorithm 6 Plumtree (2)

```
1:  Upon Receive(GOSSIP, mID, m, round, sender) Do
2:      If (mID ∉ received) Then
3:          Trigger Deliver(m);
4:          received ⟵ received ⋃ {mID};
5:          If (∃ (id,node,r) ∈ missing :id=mID) Then
6:              Cancel Timer (MISSING, mID);
7:
8:          rcvContent[mID] ⟵ {m};
9:          Setup Timer (MESSAGE, mID,$timeout_3$);
10:         Call EagerPush(mID, m, round + 1);
11:         Call LazyPush(mID, round + 1);
12:         eagerPushPeers ⟵ eagerPushPeers ⋃ {sender};
13:         lazyPushPeers ⟵ lazyPushPeers \ {sender};
14:         Call Optimize(mID, round, sender);
15:     Else
16:         lazyPushPeers ⟵ lazyPushPeers ⋃ {sender};
17:         eagerPushPeers ⟵ eagerPushPeers \ {sender};
18:         Send(PRUNE, sender, myself);
19:
20:
21: Upon Receive(PRUNE, sender) Do
22:     lazyPushPeers ⟵ lazyPushPeers ⋃ {sender};
23:     eagerPushPeers ⟵ eagerPushPeers \ {sender};
24:
25: Upon Receive(IHAVE, mID, round, sender) Do
26:     If (mID ∉ received) Then
27:         If (∄ id ∈ timers: id=mID) Then
28:             Setup Timer (MISSING, mID, $timeout_1$);
29:
30:         missing[mID] ⟵ missing[mID] ⋃ {mID, sender, round};
31:
32:
33: Upon Timer(MISSING, mID) Do
34:     Setup Timer (MISSING, mID, $timeout_2$);
35:     (mID, node, round) ⟵ removeFirstAnnouncement(missing,
    mID);
36:     eagerPushPeers ⟵ eagerPushPeers ⋃ {sender};
37:     lazyPushPeers ⟵ lazyPushPeers \ {sender};
38:     Send(GRAFT, node, mID, round, myself);
39:
40: Upon Timer(MESSAGE, mID) Do
41:     rcvContent ⟵ rcvContent \ rcvContent[mID];
42:
43: Upon Receive(GRAFT, mID, round, sender) Do
44:     eagerPushPeers ⟵ eagerPushPeers ⋃ {sender};
45:     lazyPushPeers ⟵ lazyPushPeers \ {sender};
46:     If (mID ∈ received) Then
47:         m ⟵ rcvContent[mID];
48:         Send(GOSSIP, mID, m, round, sender);
49:
50:
51: Upon NeighbourDown(node) Do
52:     eagerPushPeers ⟵ eagerPushPeers \ {node};
53:     lazyPushPeers ⟵ lazyPushPeers \ {node};
54:     Forall (id,s,n,r) ∈ lazyQueue: n=node Do
55:         lazyQueue ⟵ lazyQueue \ {(id,s,n,r)};
56:
57:     Forall (id,n,r) ∈ missing: n=node Do
58:         missing ⟵ missing \ {(id,n,r)};
59:
60:
```

Algorithm 7 Plumtree (3)

```
1:  Upon NeighbourUp(node) Do
2:      eagerPushPeers ⟵ eagerPushPeers ⋃ {node};
3:
4:  Procedure Optimize(mID, round, sender)
5:      If (∃ (id, node, r) ∈ missing: id=mID) Then
6:          If (r < round ∧ round - r >= threshold ) Then
7:              Trigger Send(GRAFT, node, null, round, myself);
8:              Trigger Send(PRUNE, sender, myself);
9:
10:
11:
```

3.2 Protocolos de redes de overlay não estruturadas

Os protocolos de redes de ovelay não estruturadas escolhidos foram o Cyclon e o HyParView. Ambos os algoritmos são apresentados nas secções 3.2.1 e 3.2.2 respectivamente.

3.2.1 ***Cyclon***. O Cyclon[12] consiste num protocolo cíclico em que cada processo da rede mantém uma vista parcial (ou vizinhança) de tamanho fixo e utiliza, periodicamente, uma mensagem de shuffle para a partilhar e actualizar. De forma mais detalhada, sempre que um dado intervalo de tempo constante passa, é enviada esta mensagem shuffle com um subconjunto da sua vista parcial ao vizinho mais antigo. Este, que é retirado da vizinhança quando é seleccionado como target do shuffle, deve responder com o seu próprio sub conjunto de vizinhos. Por fim, sempre que algum processo recebe um subconjunto de vizinhos de outro processo deve adicioná-los à sua própria vizinhança (removendo nós se necessário) ou actualizar as suas idades.

Relativamente à implementação, optámos por divergir do comportamento descrito no seu paper original, utilizando como alternativa o pseudo-código disponibilizado pela equipa docente da unidade curricular que originou este trabalho. A motivação principal para esta escolha foi a simplicidade deste segundo em comparação com o primeiro, abolindo a necessidade de timeouts relativos à resposta a um shuffle.

Adicionalmente, foi realizada (apenas) uma alteração face a este pseudo-código. Esta surge quando é recebida uma resposta a um shuffle. Na nossa implementação, o processo que recebe um subconjunto de vizinhos (ou sample) é removido caso esteja presente pois nunca deve pertencer à sua própria vizinhança. Dado o número reduzido de mudanças face ao pseudo-código já referido e a simplicidade inerente ao protocolo, optámos por não o incluir neste trabalho.

3.2.2 ***HyParView***. Este protocolo[7] procura oferecer melhor tolerância a falhas que ocorram na rede de overlay não estruturada, comparativamente com algoritmos como o Cyclon[12] ou Scamp[3]. Este protocolo propõe o uso de duas vistas parciais, uma usada mais pequena e usada para disseminação de mensagens, enquanto que a segunda é maior e serve para guardar futuros candidatos a ser movidos para a primeira vista caso ocorra alguma falha. O pseudo-código que expressa a implementação feita encontra-se ilustrado abaixo.

Comparando este pseudo-código com o que é apresentado no paper do HyParView podemos observar que houveram algumas alterações relativamente ao modo de adicionar nós a cada uma das views, mais precisamente nas funções addHostToPassiveView (9) e addHostToActiveView (9). No pseudo-codigo apresentado no paper do HyParView existe uma função chamada dropRandomElementFromActiveView que no nosso caso não foi criada. Contudo o procedimento usado nessa função é mesmo de quando um nó deve ser removido da active view pela função addHostToPassiveView.

No nosso pseudo-codigo também expressámos as operações de shuffle(9), shuffle reply (9) e o timer criado para começar a ronda de shuffle (9) ao passo que no paper onde este protocolo é apresentado não o chega a fazer.

Verificou-se que o algoritmo tinha um defeito inerente: Se algum processo é removido da active view de todos os nós na sua active view, receberá os respectivos disconnects e ficará com a sua active view vazia, tornando-se incontactável. Isto resulta em processos sós e é especialmente recorrente na fase inicial, em que active views têm um número baixo de contactos e estão em constante mudança. Para combater este problema, alterámos o algoritmo para que, quando um nó tivesse a active view vazia e fosse fazer um shuffle, fizesse um join com um processo aleatório da sua passive view.

Algorithm 8 HyParView (1)

```
1:  State
2:      av, pv;  // set of nodes for active and passive view
3:      AVSize, PVSize;  // max size of active and passive view
4:      passiveSizeShuffle; // size of passive view received on shuffle
5:      ARWL, PWRL;  // active and passive random walk length
6:      shuffleTTL;  // ttl value for shuffle requests
7:      shufflePVSize;  // size of passive view to send for shuffles
8:      shuffleAVSize;  // size of active view for shuffle
9:
10: Upon Init Do
11:     av, pv ⟵ {};
12:     Setup Periodic Timer (SHUFFLE, timetout_1);
13:     addHostToActiveView(ContactNode);
14:     Send(JOIN, ContactNode, myself);
15:
16: Upon Receive(JOIN, newNode) Do
17:     addNodeActiveView(newNode);
18:     Forall n ∈ av ∧ n ≠ newNode Do
19:         Send(FORWARDJOIN, n, newNode, ARWL, myself);
20:
21:
22: Upon Receive(FORWARDJOIN, newNode, timeToLive, sender) Do
23:     If (timeToLive == 0 ∨ #av == 1) Then
24:         addNodeActiveView(newNode);
25:     Else
26:         If (timeToLive == PWRL) Then
27:             addNodePassiveView(newNode);
28:
29:         n ⟵ n ∈ av ∧ n ≠ newNode ∧ n ≠ myself;
30:         Send(FORWARDJOIN, n, newNode, timeToLive - 1, myself;
31:
32:
```

Algorithm 9 HyParView (2)

```
1:  Upon addHostToActiveView(node) Do
2:      If (node ≠ myself ∧ node ∉ av) Then
3:          If (isFull(av)) Then
4:              n ⟵ n ∈ av;
5:              av ⟵ av \ {n};
6:              addNodePassiveView(n);
7:
8:          av ⟵ av ⋃ {node};
9:
10:
11: Upon Timer(SHUFFLE) Do
12:     If (#av = 0) Then
13:         node ⟵ node ∈ pv;
14:         addHostToActiveView(pv);
15:         Send(JOIN, pv, myself);
16:     Else
17:         node ⟵ node ∈ av;
18:         rndPassv ⟵ rndPassv ∈ pv ∧ #rndPassv == shufflePVSize;
19:         rndActive ⟵ rndActive ∈ av ∧ #rndPassv == shuffleAVSize;
20:         Send(SHUFFLE, rndActive, rndPassv, myself, shuffleTTL);
21:
22:
23: Upon addHostToPassiveView(node) Do
24:     If (node ≠ myself ∧ node ∉ av ∧ node ∉ pv) Then
25:         If (isFull(pv)) Then
26:             n ⟵ n ∈ pv;
27:             pv ⟵ pv \ {n};
28:
29:         Send(DISCONNECT, node, myself);
30:         pv ⟵ pv ⋃ {node};
31:
32:
33: Upon Receive(SHUFFLE, p, pAV, pPV, ttl, sender) Do
34:     If (ttl > 1 ∧ #av > 1) Then
35:         n ⟵ n ∈ av ∧ n ≠ p ∧ n ≠ myself;
36:         Send(SHUFFLE, pAV, pPV, p, ttl - 1, n);
37:     Else
38:         // set of random nodes
39:         rndNodes ⟵ rndNodes ∈ av ∧
40:                 #rndNodes == #pAV + #pPV;
41:         rcvNodes ⟵ pAV ⋃ pPV;
42:         Send(SHUFFLEREPLY, rndNodes, rcvNodes, p);
43:         rndNodes ⟵ rndNodes \ p;
44:         Call shuffle(rndNodes, rcvNodes);
45:
46:
47: Upon Receive(SHUFFLEREPLY, rndNodes, rcvNodes, peer) Do
48:     rndNodes ⟵ rndNodes \ p;
49:     Call shuffle(rndNodes, rcvNodes);
50:
```

Algorithm 10 HyParView (3)

```
1:  Procedure shuffle(rndNodes, rcvNodes)
2:      // set of with total of received nodes
3:      rcvNodes ⟵ rcvNodes \ (pv ⋃ av ⋃ myself);
4:      If (#rcvNodes <= PVSize - #pv) Then
5:          pv ⟵ pv ⋃ rcvNodes;
6:          Return;
7:
8:      rndNodesToRemove ⟵ rndNodesToRemove ∈ rndNodes
    ∧
9:          #rndNodesToRemove == #rcvNodes - (PVSize - #pv);
10:     pv ⟵ pv \ rndNodesToRemove;
11:
12:     If (#rcvNodes <= PVSize - #pv) Then
13:         pv ⟵ pv ⋃ rcvNodes;
14:         Return;
15:
16:     rndNodesToRemove ⟵ rndNodesToRemove ∈ pv ∧
17:         #rndNodesToRemove == #rcvNodes - (PVSize - #pv);
18:     pv ⟵ pv \ rndNodesToRemove;
19:     pv ⟵ pv ⋃ rcvNodes;
20:
```

4 AVALIAÇÃO EXPERIMENTAL

Nesta secção iremos expor as avaliações experimentais efectuadas para verificar e validar as implementações feitas. Cada teste efectuado consistiu na combinação entre um protocolo de broadcast epidémico e um protocolo de overlay não estruturado, previamente apresentados na secção 3.

4.1 Metodologia de Teste

Para criar um ambiente mais aproximado a uma situação real, foram utilizados contentores Docker pré-configurados com interfaces virtuais de rede que adicionam latência a todos os envios através de uma matriz de latências. Estes contentores foram-nos proporcionados pela equipa docente.

De modo a automatizar a realização e interpretação dos testes realizados com os diferentes algoritmos, foram criados dois scripts em Python:

4.1.1 ***Script de teste:*** Este script foi baseado num script bash que nos foi fornecido e estendido através da adição de diversas funcionalidades. É responsável por preparar o ambiente de execução, criar os contentores Docker que vão correr o programa bem como a swarm que os interliga e lançar o programa em todos os contentores simultaneamente. Algumas funcionalidades que o distinguem do script proporcionado, são a possibilidade de pré-configurar x rotinas para executar com diferentes processos e y repetições para cada uma, de forma a ser possível mais tarde extrair valores médios. O script permite também detectar se todos os processos já terminaram a execução. É criado, de forma dinâmica, um ficheiro com o output de cada processo para uma directoria configurável. É gerado um ficheiro adicional, para ser consumido pelo script de parsing, com as configurações dos testes.

4.1.2 ***Script de parsing:*** Este script é responsável por ler e interpretar os outputs gerados pelo script de testes. O seu funcionamento pode ser dividido em três fases: parsing dos ficheiros, análise de resultados e exportação dos resultados em gráficos e ficheiro CSV. Durante a primeira fase, todos os envios e recepções respectivos à camada de aplicação (denominada como BroadcastApp) são colocados numa estrutura de dados e as mensagens das camadas de gossip e de broadcast, contabilizadas. Na segunda fase, toda a estrutura

de dados gerada contendo os broadcasts recebidos e enviados é percorrida para determinar a latência dos broadcasts e a quantos nós cada um chegou. Numa última fase, é realizada uma média de todas as repetições de uma rotina para obter um valor concreto e, através deste valor, são gerados gráficos e um CSV. Para poupar tempo, a primeira e segunda fase são completamente paralelizadas.

4.2 Métricas de teste

Para cada par de algoritmo de overlay e algoritmo de broadcast epidémico extraímos 9 métricas de modo a avaliar o seu desempenho. Estas são:

- Número mensagens e bytes trocados na camada de overlay, broadcast epidémico e total: estas métricas permitem-nos deduzir o grau de saturação do canal por par de protocolos;
- Cobertura média de broadcasts: Esta métrica é considerada, no enunciado do projecto, como a fiabilidade do sistema, mas, a nosso ver, um broadcast só deve ser considerado correcto se chegar a todos os nós da rede. Dado isto, mantivemos a métrica mas com uma nomenclatura que julgamos ser mais correcta. Esta métrica dá-nos o número médio de nós a que todos os broadcasts chegaram. É útil nos casos em que, por algum motivo, nenhum broadcast chega a 1 ou 2 nós, resultando numa fiabilidade de 0, mesmo numa rede de 100.000 nós;
- Fiabilidade de broadcasts: Esta métrica é a percentagem de broadcasts correctos realizados que tiveram sucesso (chegaram a todos os nós na rede). Consequentemente, numa rede com $n \to +\infty$ nós, um broadcast que chegue a $n-1$ nós, terá a sua cobertura a tender para $+\infty$ mas a sua fiabilidade irá ser 0;
- Latência de broadcast: Este valor, em milissegundos, representa o tempo médio passado entre o envio de cada broadcast correcto e o instante em que se torna recebido por todos os nós. Consequentemente, num broadcast em que falhe, no mínimo, um nó, a sua latência será considerada como infinita e não será tida em conta;

4.3 Hardware

Para garantir a coerência dos resultados, todos os testes foram executados em dois servidores em simultâneo equipados com um processadores AMD EPYC 7281, cada um com 16 cores e 24 threads, 128GB de memória e 2 interfaces de rede de 10 Gbps. Ambas as máquinas corriam uma distribuição Ubuntu LTS. As máquinas estavam hospedadas num cluster de computação pertencente ao Departamento de Informática da FCT NOVA[2], cujo acesso nos foi proporcionado pela equipa docente com o intuito de correr as experiências necessárias à realização deste relatório. As máquinas utilizadas pelo cluster eram geridas pelo software de gestão de clusters *OAR*[1].

4.4 Configurações

4.4.1 ***Aplicação:*** optámos por correr a aplicação em apenas 50 nós devido a ter sido observada uma saturação, de recursos de processamento, memória e rede, ao correr a mesma simultaneamente em 100 nós. Acreditamos que isto terá sido, em grande parte, devido ao overhead adicional relativo à utilização de contentores Docker para correr cada processo. A matriz de latências utilizada para definir a latência entre contentores Docker foi de 100ms constantes. O tamanho de payloads trocadas foram de 1KB e 100 KB (1MB estava a saturar os canais) e o valor para os intervalos entre broadcasts foram de 300ms e 3 segundos. O tempo de execução utilizado foi de 30 segundos, devido ao vasto tamanho observado dos logs gerados pelos processos, resultando num tempo de processamento offline muito elevado, nas nossas máquinas. O tempo de cooldown foi de 60 segundos, garantindo que a rede tinha tempo suficiente para difundir todas as mensagens ainda em circulação. O tempo de preparação (entre o processo lançar e começar a difundir broadcasts) utilizado, foi de 60 segundos, de modo a dar tempo suficiente para o overlay estabilizar;

4.4.2 ***Adaptive Gossip:*** a natureza multifacetada deste algoritmo obrigou-nos a testar qual o melhor modo de funcionamento para o nosso cenário. Testámos o modo TTL, em que faz eager push até atingir um TTL configurado e pure eager push, em que é feito sempre. Concluímos que o modo eager push era o superior e é este o que utilizámos para comparação com os outros algoritmos. Para os pârametros relativos ao t (TTL máximo) e f no protocolo adaptive gossip, utilizámos os valores 4 e 3 (este último é irrelevante em modo pure eager push), respectivamente. Os valores basearam-se nas propriedades logarítmicas da relação entre o grafo conexo e número do nós (50), dado que $\ln(50) \approx 4$, conforme [11]. Para o timer que implementa a rotina de pedidos IWant, utilizámos um intervalo de 50ms. Foram também corridos alguns testes ;

4.4.3 ***Plumtree:*** para o temporizador periódico IHave, foi utilizado o valor de 500ms que é 2.5 vezes o RTT esperado. Para o timeout graft, obrigatoriamente mais curto que o ihave, optou-se pelo intervalo de 300ms, que é 1.5 vezes o RTT esperado. Para o tempo máximo até uma mensagem ter o seu conteúdo apagado , utilizaou-se o valor de 10s;

4.4.4 ***Cyclon:*** optou-se por um fanout (tamanho da vista parcial) de 6, um pouco superior ao logaritmo neperiano do número de nós, após este número apresentar melhores resultados com pouco impacto adicional. O tamanho da amostra da vizinhança para o procedimento de shuffling enviada utilizado, foi 3, devido a ser metade do tamanho da vizinhança. Para o intervalo entre shuffles, utilizámos 5 segundos por termos um tempo de execução total curto. Este último é sempre superior ao dobro da latência máxima esperada;

4.4.5 ***HyParView:*** optámos por um fanout (tamanho da active view) de 6, igual ao do Cyclon. Para o tamanho da passive view, utilizou-se 20, um valor um pouco inferior ao recomendado em [7], dado o número reduzido de nós. O TTL para o shuffle utilizado foi 4 ($\ln(50)$) e os valores da active e passive view trocadas de 3 e 10, respectivamente, por serem metade do seu tamanho total. O número de random walks para a active e passive view escolhidos foram de 5 e 3, respectivamente. O intervalo entre shuffles escolhido foi também igual ao do Cyclon;

4.5 Resultados

Em todos os testes realizados no cluster, com recurso a contentores Docker, verificámos niveis muito altos de consumo de recursos, que

não se verificaram quando estes foram corridos localmente sem recorrer a esta tecnologia.

Através dos gráficos presentes na figura 1, é possível verificar que, para na ausência de saturação e com todos os processos correctos, é expectável que a proporção entre mensagens enviadas e recebidas, na camada de overlay, seja aproximadamente 1:1. Isto verificou-se, como é natural, independentemente das configurações ou protocolos utilizados para disseminação e gossip, dado que estes se situam em diferentes camadas. Também podemos observar que o método de shuffling utilizado pelo Cyclon, que executa este procedimento a contactar directamente outro nó, resulta numa redução significativa do número de mensagens trocadas em comparação com o HyParView, que transmite uma mensagem com um TTL afixado, causando a criação de $TTL - 1$ outras mensagens.

Nos algortimos de overlay Cyclon e HyParView, utilizámos fanouts de 6, significativamente superior ao logaritmo neperiano do número de nós na rede. Isto, apesar de ajudar a criar um overlay mais robusto, devido à redundância adicional, implica um maior número de redundância de mensagens oriundas de disseminações. Esta discrepância mais acentuada entre o número de mensagens enviadas e recebidas é verificável na figura 2. Por outro lado, a utilização de protocolo de gossip com capacidades de lazy push, como o Plumtree, o único deste tipo nestes testes, permite mitigar este efeito, mantendo a robustez adicional oferecida pelo aumento do fanout.

O método utilizado para obter o tamanho de uma mensagem enviada, foi através da obtenção do seu tamanho em formato de String de Java. Logo, é expectável que, na camada de overlay, onde todas as mensagens têm tamanhos muito semelhantes, o número de bytes transmitidos é directamente proporcional ao número de mensagens transmitidas, como é verificável na figura 3.

Ao comparar, na figura 4 com payloads de 1KB, os bytes transmitidos entre Plum Tree com Cyclon e Adaptive com Cyclon, é visível uma redução no total de bytes transmitidos entre um e outro. Isto é expectável devido à própria natureza dos algoritmos de lazy push, que evitam a disseminação de mensagens redundantes. Faria sentido, esta diferença ser ainda mais acentuada na imagem de baixo, onde são foram utilizados payloads de 100KB, dado que estes algoritmos tendem a mostrar-se mais vantajosos proporcionalmente ao tamanho das payloads trocadas. Ao acedermos ao Grafana, um painel que permite monitorizar os recursos utilizados pelo sistema em tempo real, verificámos que estava a ocorrer uma saturação significativa de recursos a nível de rede e processadores. Após analisar os logs, verificámos que os canais perdiam constantemente as ligações, devido a esta mesma saturação. É importante notar que o mesmo não é observável na figura 3. Isto, devido a ser dado um tempo de preparação à camada de overlay, antes do começo da difusão de broadcasts, para permitir a estabilização da mesma, e esta troca de mensagens nesta fase não é suficiente para saturar os recursos.

Na figura 5, nos pares com Cyclon, é possível se verificar uma cobertura média de broadcasts quase perfeita, apesar das falhas acima mencionadas. Acreditamos que isto se deve ao facto de utilizarmos um período de cooldown bastante longo (60s), que permite aos broadcasts eventualmente chegarem a todos, aliado a um fanout significativo que torna o overlay mais resiliente a falhas. Na figura 6, podemos verificar que, para payloads de 100KB, a latência média ultrapassa por muito a latência de canal de 100ms. Demonstrando que a saturação causada está a aumentar o tempo de entrega e a causar perdas de mensagens.

No caso do HyParView, na figura 5, verifica-se um nível significativamente mais baixo de cobertura. Este padrão de pior desempenho pelo HyParView, face ao Cyclon, tem sido recorrente em todas as experiências. Mesmo após uma análise detalhada das causas deste fraco desempenho, não foi possível apurar uma causa aparente. Dada a maior resiliência demonstrada pelo algoritmo face ao Cyclon, durante falhas [7]. Suspeitamos que exista uma falha na implementação do algoritmo, da nossa parte. Os efeitos desta falta de conectividade entre os nós, são demonstrados na figura 7, onde é verificável que o número de broadcasts que chegam a todos os nós é quase nulo ou até mesmo nulo.

Na figura 7, podemos também verificar que, com payloads de 100KB, o PlumTree mostra melhor desempenho que o Adaptive em pure eager push. Isto deve-se ao facto de que a saturação causada pela utilização de técnicas puramente baseadas em eager push aumenta exponencialmente com o tamanho das payloads. Mas, mesmo assim, é possível ver, na figura 6, que, mesmo sobre saturação, a técnica baseada em apenas eager push, demonstra sempre menor latência.

Quanto ao modo TTL, do protocolo Adaptive Gossip, foram realizados vários testes e todos eles demonstraram valores extremamente baixos de fiabilidade e latência e significativamente baixos de cobertura. Para payloads de 1KB a cada 300ms, observou-se 1.3%, ≈30s e 35%, respectivamente. Esperávamos que a natureza do lazy push resultasse em melhores resultados face a eager push com payloads substancialmente maiores. Mas, o mesmo também não se verificou. O que nos levou a à conclusão de que existe, muito provavelmente, uma falha de implementação que não foi detectada.

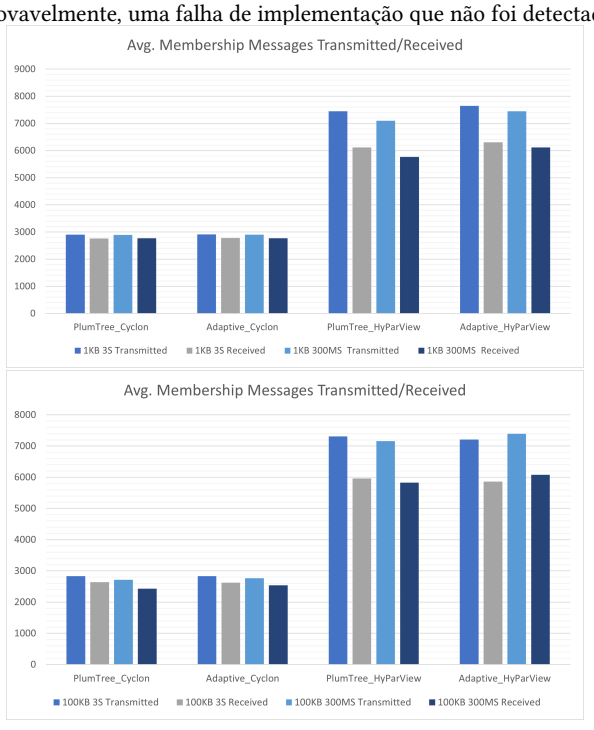

Figure 1: Média de mensagens trocadas na camada de overlay.

Nos algortimos de overlay Cyclon e HyParView, utilizámos fanouts de 6, significativamente superior ao logaritmo neperiano do número de nós na rede. Isto, apesar de ajudar a criar um overlay mais robusto, devido à redundância adicional, implica um maior número de redundância de mensagens oriundas de disseminações. Esta discrepância mais acentuada entre o número de mensagens enviadas e recebidas é verificável na figura 2. Por outro lado, a utilização de protocolo de gossip com capacidades de lazy push, como o Plumtree, o único deste tipo nestes testes, permite mitigar este efeito, mantendo a robustez adicional oferecida pelo aumento do fanout.

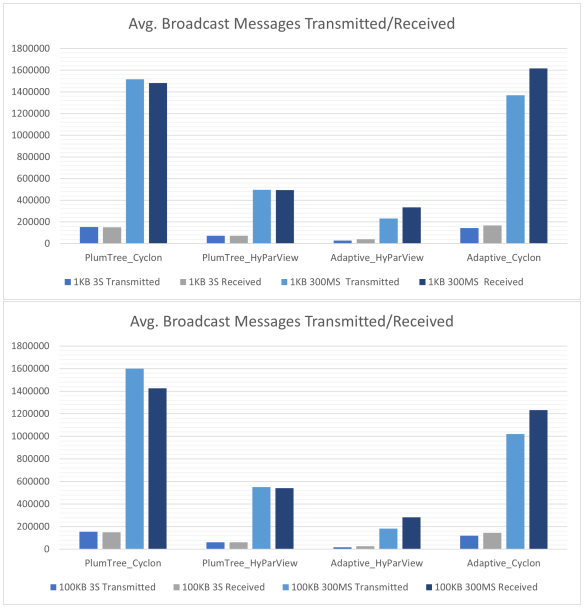

Figure 2: Média de mensagens trocadas na camada de broadcast.

O método utilizado para obter o tamanho de uma mensagem enviada, foi através da obtenção do seu tamanho em formato de String de Java. Logo, é expectável que, na camada de overlay, onde todas as mensagens têm tamanhos muito semelhantes, o número de bytes transmitidos é directamente proporcional ao número de mensagens transmitidas, como é verificável na figura 3.

Ao comparar, na figura 4 com payloads de 1KB, os bytes transmitidos entre Plum Tree com Cyclon e Adaptive com Cyclon, é visível uma redução no total de bytes transmitidos entre um e outro. Isto é expectável devido à própria natureza dos algoritmos de lazy push, que evitam a disseminação de mensagens redundantes. Faria sentido, esta diferença ser ainda mais acentuada na imagem de baixo, onde são foram utilizados payloads de 100KB, dado que estes algoritmos tendem a mostrar-se mais vantajosos proporcionalmente ao tamanho das payloads trocadas. Ao acedermos ao Grafana, um painel que permite monitorizar os recursos utilizados pelo sistema em tempo real, verificámos que estava a ocorrer uma saturação significativa de recursos a nível de rede e processadores. Após analisar os logs, verificámos que os canais perdiam constantemente as ligações, devido a esta mesma saturação. É importante notar que o mesmo não é observável na figura 3. Isto, devido a ser dado um

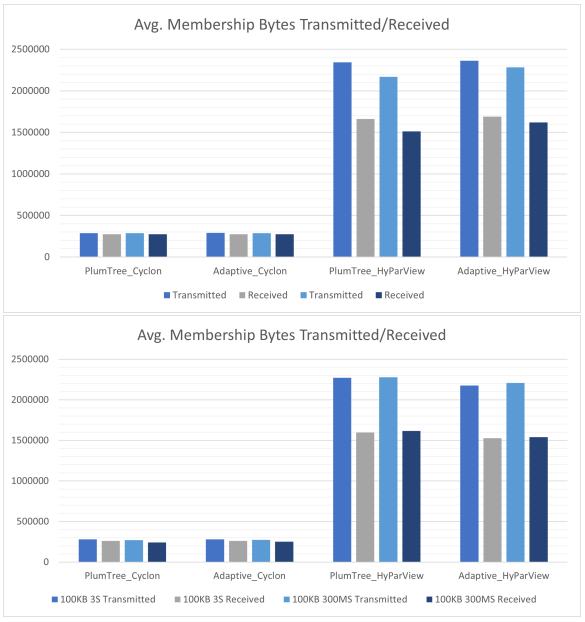

Figure 3: Média de bytes trocados na camada de overlay.

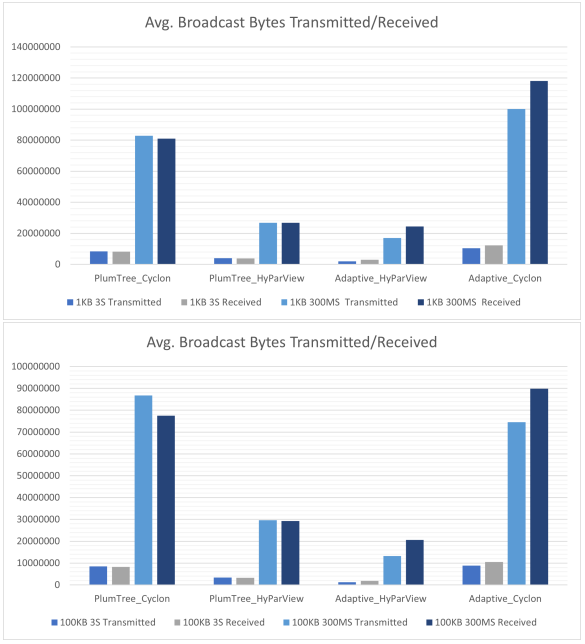

Figure 4: Média de bytes trocados na camada de broadcast.

tempo de preparação à camada de overlay, antes do começo da difusão de broadcasts, para permitir a estabilização da mesma, e esta troca de mensagens nesta fase não é suficiente para saturar os recursos.

Na figura 5, nos pares com Cyclon, é possível se verificar uma cobertura média de broadcasts quase perfeita, apesar das falhas

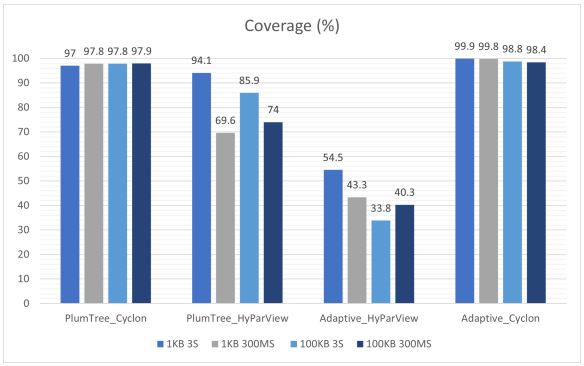

Figure 5: Cobertura média de broadcasts (%).

acima mencionadas. Acreditamos que isto se deve ao facto de utilizarmos um período de cooldown bastante longo (60s), que permite aos broadcasts eventualmente chegarem a todos, aliado a um fanout significativo que torna o overlay mais resiliente a falhas. Na figura 6, podemos verificar que, para payloads de 100KB, a latência média ultrapassa por muito a latência de canal de 100ms. Demonstrando que a saturação causada está a aumentar o tempo de entrega e a causar perdas de mensagens.

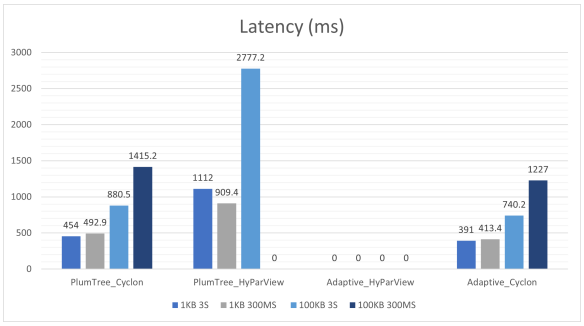

Figure 6: Latência média de broadcasts (ms).

No caso do HyParView, na figura 5, verifica-se um nível significativamente mais baixo de cobertura. Este padrão de pior desempenho pelo HyParView, face ao Cyclon, tem sido recorrente em todas as experiências. Mesmo após uma análise detalhada das causas deste fraco desempenho, não foi possível apurar uma causa aparente. Dada a maior resiliência demonstrada pelo algoritmo face ao Cyclon, durante falhas [7]. Suspeitamos que exista uma falha na implementação do algoritmo, da nossa parte. Os efeitos desta falta de conectividade entre os nós, são demonstrados na figura 7, onde é verificável que o número de broadcasts que chegam a todos os nós é quase nulo ou até mesmo nulo.

Na figura 7, podemos também verificar que, com payloads de 100KB, o PlumTree mostra melhor desempenho que o Adaptive em pure eager push. Isto deve-se ao facto de que a saturação causada pela utilização de técnicas puramente baseadas em eager push aumenta exponencialmente com o tamanho das payloads. Mas, mesmo assim, é possível ver, na figura 6, que, mesmo sobre saturação, a

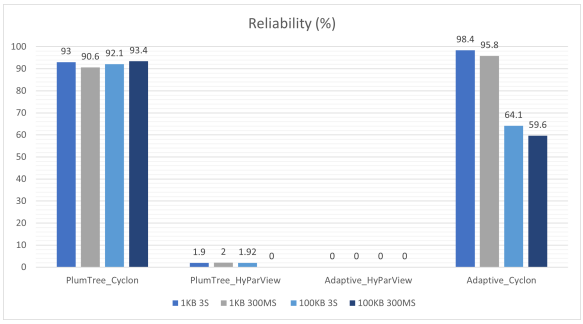

Figure 7: Fiabilidade média de broadcasts (nr. de nós).

técnica baseada apenas em eager push, demonstra sempre menor latência.

Quanto ao modo TTL, do protocolo Adaptive Gossip, foram realizados vários testes e todos eles demonstraram valores extremamente baixos de fiabilidade e latência e significativamente baixos de cobertura. Para payloads de 1KB a cada 300ms, observou-se 1.3%, ≈30s e 35%, respectivamente. Esperávamos que a natureza do lazy push resultasse em melhores resultados face a eager push com payloads substancialmente maiores. Mas, o mesmo também não se verificou. O que nos levou a à conclusão de que existe, muito provavelmente, uma falha de implementação que não foi detectada.

5 CONCLUSÃO

Neste relatório fizemos um conjunto de avaliações experimentais em algoritmos ja existentes. Para cada teste efectuamos uma combinação entre um protocolo de gossip e outro de overlay a fim de determinar qual a melhor combinação que demonstrava a melhor performance, dados as avaliações efectuadas.

Pudemos concluir que existem vantagens inerentes à utilização de técnicas de lazy push, para a redução do tráfego na rede com grandes payloads. Mas também que estas vantagens não surgem sem impactos significativos na latência. Porem, também foi verificado que, perante um grau suficientemente elevado de saturação estas contribuem de forma significativa para a fiabilidade de comunicação.

Outra das conclusões retirada destes resultados passa pela quantidade baixa de disseminação de mensagens utilizada pelo Cyclon para manter as suas vistas parciais em comparação com o HyParView, com fanouts equivalentes.

Também foi observado que a utilização de um fanout mais elevado, por parte dos protocolos de overlay, cria uma maior redundância, garantindo assim que o overlay é mantido mesmo perante a presença de um elevado número de falhas. Porém, isto é estas vantagens são obtidas à custa de uma maior saturação da rede, principalmente durante a utilização de protocolos de disseminação que utilizam mecanismos baseados em eager push. Contudo, verificámos que ao utilizar um protocolo baseado em lazy push a redundância no número de mensagens é diminuída mas a tolerância a falhas é mantida, criando um balanço saudável.

Por último, não podem deixar de ser mencionadas as nossas dificuldades em obter um bom desempenho com um grande número de nós através da utilização de contentores Docker. Estes fizeram

grande parte dos percalços encontrados durante o desenvolvimento do trabalho. Lamentamos também o facto de termos falhas aparentes na implementação, tanto relativas ao HyParView como ao modo TTL do Adaptive Gossip, que impediram uma comparação mais rica entre as diversas soluções.

REFERENCES

[1] Pierre Neyron Olivier Richard Salem Harrache Bruno Bzeznik, Nicolas Capit. 2018 (acedido a 16 de Novembro, 2020). OAR. https://oar.imag.fr/

[2] DI FCT NOVA. 2020 (acedido a 16 de Novembro, 2020). DI-Cluster Wiki. https://cluster.di.fct.unl.pt/

[3] A. Ganesh, A. Kermarrec, and L. Massoulié. 2003. Peer-to-Peer Membership Management for Gossip-Based Protocols. *IEEE Trans. Computers* 52 (2003), 139–149.

[4] M. J. Monteiro R. Oliveira A. M. Kermarrec J. Pereira, L. Rodrigues. 2003. NEEM: Network-friendly Epidemic Multicast. (2003).

[5] Luis Rodrigues Joao Leitao. 2012. CellFarm: An Overlay for Supporting Replication and Load Distribution inLarge-Scale Systems. (2012).

[6] J. Leitão, J. Pereira, and Luís E. T. Rodrigues. 2007. Epidemic Broadcast Trees. *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)* (2007), 301–310.

[7] J. Leitão, J. Pereira, and Luís E. T. Rodrigues. 2007. HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast. *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)* (2007), 419–429.

[8] João Carlos Antunes Leitão. 2007. Gossip-based broadcast protocols. *UNIVERSIDADE DE LISBOA FACULDADE DE CIÊNCIAS DEPARTAMENTO DE INFORMÁTICA* (2007).

[9] Eng Keong Lua, J. Crowcroft, M. Pias, Ravi Sharma, and S. Lim. 2005. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials* 7 (2005), 72–93.

[10] Iosif Lazardis Bijit Hore Nalini Venkatasubramanian Sharad Mehrotra Mayur Deshpande, Bo Xing. 2006. CREW: A Gossip-based Flash-Dissemination System. *26th IEEE International Conference on Distributed Computing Systems* (2006).

[11] A.-M.Kermarrec L.Massoulie P.T. Eugster, R.Guerraoui. 2004. From Epidemics to Distributed Computing. (2004).

[12] Spyros Voulgaris, D. Gavidia, and M. V. Steen. 2005. CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. *Journal of Network and Systems Management* 13 (2005), 197–217.