Algoritmos e Sistemas Distribuídos: Projeto (Fase 1)

António Silva
Rafael Seara
Raquel Macedo

Professor: João Leitão

FCT-UNL

Outubro 2017

Contents

1 Arquitetura

Cada nó do sistema apresenta-se segundo a recomendação do professor, contendo a camada Partial View Membership, Information Dissemination, Global Membership e a Aplicação de Teste [Fig:1]. A implementação das diferentes camadas pode ser observada na diretoria *partialview, communicationview, globalview* e *testlayer* do projeto respetivamente às camadas suprarreferidas.

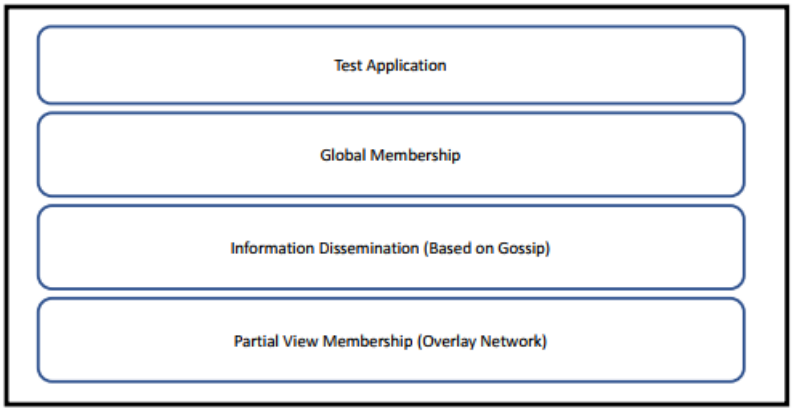

Figure 1: Arquitetura de um nó do sistema.

A camada da aplicação de testes comunica com todas as outras camadas por necessitar de imprimir os dados atuais de cada uma das camadas inferiores. A camada *Global Membership* recebe eventos da *Partial View Membership* quando algum nó falha ou um novo nó se quer juntar ao sistema. A camada *Global Membership* também troca mensagens com a camada de *Information Dissemination*. Sendo que, quando esta camada pretende fazer *broadcast* de uma mensagem para os vizinhos do nó, envia a mensagem para a camada de *Information Dissemination*, a qual a propaga para a mesma camada dos nós vizinhos. Quando uma destas mensagens é recebida por um nó, a camada de *Information Dissemination* vai entregar a mensagem à camada de *Global Membership* para que esta a processe.
A camada *Information Dissemination* trata, também de eventos gerados pela camada *Partial View Membership* que informam sobre alterações na sua lista de nós disponíveis para enviar mensagens (nós vizinhos).
A camada *Partial View Membership* não recebe eventos de nenhuma outra camada mas envia eventos (suprarreferidos) para a camada *Information Dissemination* e *Global Membership*.
Todas as interações entre camadas podem ser observadas na figura 2.

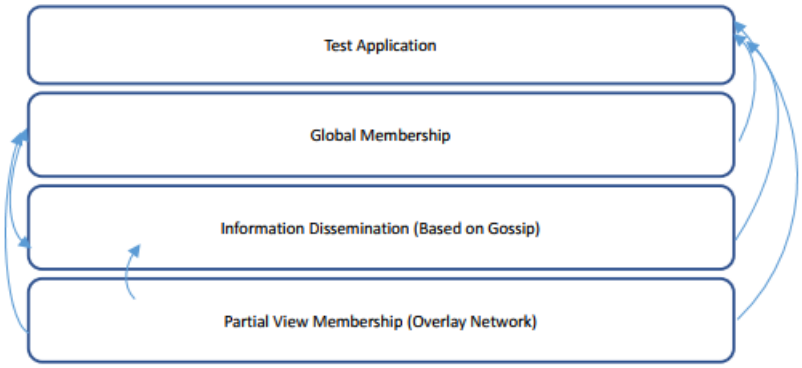

Figure 2: Interações entre camadas do sistema.

2 Protocolo

2.1 Partial View Membership

É esperado que o sistema possua cerca de 10 mil nós o que significa que muitos desses processos poderão falhar. Com esta assunção decidimos implementar uma adaptação do algoritmo Hyparview[1]. Este algoritmo faz uso da sua lista ativa, uma lista com nós com garantias de que se encontram prontos para comunicar, e usa a sua lista passiva de nós como nós de reserva caso algum nó da lista ativa falhe. Como não existe garantia de que os nós da lista passiva estejam prontos para tomar um lugar na vista ativa, foi criada a lista *passiveActive*. Esta lista de nós tem tamanho fixo e configurável e contém uma amostra aleatória de nós da passiva com garantias de que estão ativos. Interminavelmente, e após um período fixo de tempo, os nós desta lista são novamente escolhidos aleatoriamente. Esta escolha aleatória dos nós garante que todos os nós vão em algum ponto no tempo pertencer à lista *passiveActive*, sendo que, no momento em que um nó é eleito para pertencer à lista, é verificado se este está ativo e se não estiver é removido da lista passiva. De forma a ganhar tempo na recuperação do sistema, quando um nó da ativa falha vai ser substituído por um da lista *passiveActive*, pois tem garantias de estar operacional. Assim, deixa de ser necessário realizar várias tentativas de conexão a nós da lista passiva quando se está à procura de um nó ao qual enviar um *neighbor request*.
Esta camada é responsável por ir atualizando a lista de nós que a camada de comunicação usa para fazer o *broadcast* das mensagens e, também, por avisar a camada *global view membership*:

1. Quando este é um nó de contacto e recebe informação de que existe um novo nó a querer entrar no sistema gerando a adição deste nó à *global view*;

2. Quando deteta que um nó falhou, fazendo com que a camada *global view membership* propague a informação da suspeita de falha deste nó e, caso tenha realmente falhado, acabe por remove-lo da *global view*.

2.2 Information Dissemination

Nesta camada é utilizada uma estratégia de *gossip*, sendo que, sempre que a camada de cima (a *global view membership*) manda propagar uma mensagem, esta camada vai enviá-la a todos os vizinhos disponiveis que serão os mesmos nós que a *partial view* tem na sua lista ativa. Esta camada mantém uma cópia local da lista de vizinhos (vista ativa) da camada de baixo. Sempre que há uma alteração da vista ativa na camada de baixo, essa camada informa esta da alteração para que se possa atualizar a sua cópia local.

2.3 Global View Membership

No *Global View Membership* é utilizada uma estratégia de propagação de eventos. Os tipos de eventos que existem são *new, maybeDead* e *stillAlive*, sendo que o primeiro significa a deteção do aparecimento de um novo nó no sistema, o segundo significa que há razões para suspeitar que um nó falhou e o último surge quando um nó do qual existem suspeitas de ter falhado informa o sistema de que ainda está correto. Sempre que o nó deteta, na camada da *partial view membership*, que um dos nós seus vizinhos falhou, essa informação é propagada para a *Global View Membership*. Isto leva à criação de um evento do tipo *maybeDead*. Quando um nó recebe um evento deste tipo, caso este seja relativo a ele próprio, vai gerar um evento do tipo *stillAlive*. Caso receba um evento do tipo *maybeDead* relativo a outro nó vai acrescentar este a uma lista de nós que se suspeita terem falhado e criar um *timer*, assim, em vez de assumir de imediato que o nó em causa falhou realmente, dá um intervalo de tempo durante o qual espera por informações em contrário. Se durante esse intervalo de tempo receber um evento do tipo *stillAlive* cancela o *timer*, remove o nó da lista de nós suspeitos de ter falhado e não chega a remover o nó da sua *gloval view*, evitando assim a remoção de um nó ainda correto. Caso tal não aconteça, terminado o intervalo de tempo assume-se que o nó realmente falhou e este é removido da *global view* e da lista de nós suspeitos de ter falhado. Se mais tarde receber um evento do tipo *stillAlive* volta a adicionar o nó à *global view*.
Quando um nó se junta ao sistema, o seu *contact node* gera um evento do tipo *new*, de forma a informar o sistema do aparecimento deste novo nó. Para além disso, o *contact node* envia ao novo nó uma cópia da sua *gloval view* e os ids dos eventos que viu recentemente, de forma a evitar que o novo nó processe eventos que já estão refletidos na *global view* que recebeu. O novo nó adopta a *global view* recebida como sua.
Quando é gerado um novo evento este é inserido numa lista de eventos pendentes, quando é feito um *gossip* os eventos desta lista são propagados para os nós vizinhos. Eventos recebidos de um nó vizinho são processados, sendo que,

para cada evento, é verificado se este já tinha sido processado, consultando-se a lista de eventos já realizados. Caso o evento ainda não tivesse sido realizado, este é acrescentado à lista de eventos pendentes, para que seja posteriormente propagado, e são feitos os procedimentos necessários para que esse evento seja refletido na *global view*.
Periodicamente, cada nó gera um *gossip*, enviando para todos os seus vizinhos um *hash* da sua *global view*, a sua lista de eventos pendentes e a informação de se há nós que se suspeita terem falhado e que estão à espera de serem removidos da *global view*. Quando o sistema está estável, sendo a *global view* constante e não havendo eventos a ser propagados, não se quer que esta transmissão de mensagens para os vizinhos ocorra com muita frequência, pelo que o periodo entre *gossips* consecutivos não deverá ser demasiado reduzido. No entanto, quando existem novos eventos, estes devem ser propagados pelo sistema mais depressa, pelo que, quando surge um evento na lista de eventos pendentes, passando a existir eventos para propagar, vai ser accionado um *timer*. Caso no fim desse *timer* não tenha ainda ocurrido o envio periodico de mensagens, então é feito logo esse envio. Pelo contrário, caso ocorra um envio periodico antes do fim do *timer*, os eventos pendentes serão logo propagados, pelo que é cancelado o *timer*.
O *hash* enviado nas mensagens de *gossip* é utilizado para detetar potenciais inconsistencias entre as *global views* dos diferentes nós. Quando um nó, **node1**, tem a sua lista de nós à espera de ser removidos vazia e não tem eventos pendentes por enviar, ou seja, encontra-se num estado estável, e recebe uma mensagem de um nó vizinho, **node2** e nesta vem a informação de que esse nó tem a sua lista de nós à espera de ser removidos vazia, então, como ambos os nós aparentam estar num estado estável, são comparados os valores dos *hash* das respetivas *global views*. Caso estes tenham valor diferente significa que foi detetado um conflito. Para tentar evitar estar a resolver conflitos que passado pouco tempo se resolvem sozinhos (por exemplo chegando a um nó eventos que este ainda não tinha processado, mas que o outro nó já tinha), o nó mantém uma lista de nós que da última vez já estavam em conflito consigo. Assim, quando o nó **node1** deteta pela segunda vez um conflito com um mesmo nó, **node2**, vai tentar resolvê-lo.
Quando vai resolver um conflito, o **node1** começa por verificar qual dos nós em conflito (ele próprio ou o que lhe enviou uma mensagem) é responsável por resolver o conflito. O responsável vai ser o nó com o identificador maior. Caso seja o próprio **node1**, ele envia uma mensagem diretamente à camada *global view membership* do **node2** a pedir a sua *global view*, para que a possa comparar com a que tem. Se pelo contrário, for o outro nó, **node2** o responsável, então é logo enviado para esse nó uma mensagem que contém a *global view* do **node1**. Ao receber uma *global view* de outro nó, vai se comparar esta com a do próprio, sendo que por cada nó **n** pertencente a uma *global view*, que não está presente na outra, vai ser enviada uma mensagem a esse nó **n**. Caso o nó responda significa que está vivo e é, por isso gerado um novo evento que fará com que os nós que não tinham **n** na sua *global view* o acrescentem. Se o nó não responder, então significa que falhou e, por isso, é gerado um evento que fará com que os nós que tinham esse nó **n** na sua *global view* o removam. Assim, são resolvidos os conflitos entre os dois nós, sendo que passam ambos a apresentar mesma *global view*.

3 *Accuracy*

3.1 Todos os processos corretos acabarão por ser reportados como fazendo parte da global membership de todos os processos corretos

Sempre que o nó (processo) se junta ao sistema, o *contactNode* através do qual se junta vai sempre gerar um evento do tipo *new* e acrescentar esse novo nó à sua *global view*.

Os eventos criados por um nó vão sempre ser propagados a todos os vizinhos desse nó. Os eventos recebidos por um nó, que este nó nunca tinha recebido antes, são processados e propagados a todos os vizinhos. Logo, ao receber um evento sobre o surgimento de um novo nó no sistema, esse nó vai ser acrescentado à *global view* e o evento vai ser propagado para os vizinhos.

Caso algum nó não consiga comunicar com o nó em questão vai gerar um evento a informar da possibilidade de falha do nó, mas caso este não tenha falhado ao receber este evento vai imediatamente gerar um outro evento a informar que ainda está ativo e propaga-lo. Ao receber o evento a informar de que um nó ainda está ativo, este volta a ser acrescentado à *global view*. Assim, e como um nó ativo estará sempre pelo menos na sua própria *global view*, não é possível um nó ativo não estar presente em nenhuma *global view*.

Basta um nó ativo estar presente numa *global view* para que, eventualmente, acabe por estar presente na de todos os processos corretos. Isto acontece pois, existe um mecanismo de deteção e tratamento de conflitos que, quando

dois nós vizinhos têm a *global view* com alguma diferença, vai verificar se os nós sobre os quais discordam estão ou não ativos e gerar um evento novo que será propagado pelo sistema e que leva à adição ou remoção do nó das *global views* consoante este está ou não ativo.

3.2 Todos os processos que falham acabarão por ser removidos da global membership de todos os processos corretos

Cada nó (processo) ao não conseguir comunicar com um dos nós da sua vista ativa irá suspeitar da falha deste.

Ao suspeitar de uma falha irá gerar um evento do tipo *maybedead*.

Os eventos criados por um nó vão sempre ser propagados a todos os vizinhos desse nó. Os eventos recebidos por um nó, que este nó nunca tinha recebido antes, são processados e propagados a todos os vizinhos.

Sempre que se gera ou recebe um novo evento do tipo *maybedead* o nó ao qual este evento é relativo fica numa lista de nós a remover. Passado um certo intervalo de tempo o nó é efetivamente removido da *global view*.

O nó que falhou vai sempre estar removido de pelo menos um dos nós do sistema. O nó que suspeitou da falha e gerou o evento, caso não receba um evento a informar que afinal o nó ainda está ativo, irá sempre removê-lo da sua *global view*. Se receber um evento a informar de que está ativo quando na realidade não está, então é porque esse evento foi gerado antes do nó ter falhado, pois é o próprio nó a gerar o evento. Logo o evento de suspeita de falha, **evento1**, e o que informa de que ainda está ativo, **evento2**, terão de chegar a algum nó do sistema pela ordem certa, nem que seja apenas a um dos vizinhos do nó em questão através do qual foi inicialmente propagado o **evento2**. Isto leva a que, pelo menos neste nó, o nó que falhou esteja removido da *global view*.

Caso a remoção não seja feita de todos os nós do sistema, basta o nó não estar presente na *global view* de um dos nós para que acabe por ser removido de todas as *global views*. Isto acontece devido ao mecanismo de deteção e tratamento de conflitos que verifica se os nós que estão presentes na *global view* de um nó, mas não na de um nó seu vizinho, estão ou não ativos. Ao verificar que um desses nós não responde e, portanto, está de facto inativo, vai gerar um novo evento que vai ser propagado ao longo do sistema e que vai conduzir à remoção do nó das *global views* de todos os nós.

4 Testes

Nesta secção apresentamos os testes realizados para verificar a correção e eficiência da implementação. Para esse fim, desenvolvemos um conjunto de aplicações de forma a obter os seguintes dados:

grafo representativo da rede - vista ativa e passiva de cada nó

gráfico que apresenta o número de mensagens recebidas e enviadas de cada tipo

gráfico que apresenta os nós das *global view* de cada nó

Os dados apresentados foram obtidos utilizando *scripts python* que acompanham o relatório e que podem ser executados através do *script bash ns.sh*. Cada nó escreve o seu estado (vistas e contadores de mensagens) para o ficheiro *data.json* sempre que sofre uma alteração na vista global ou na lista de eventos pendentes. Com isto pretende-se limitar as escritas no ficheiro e evitar escritas concorrentes, pois os *scripts python* leem o conteúdo desse ficheiro em intervalos de 3 segundos.

É também possível utilizar os seguintes comandos em cada processo durante a sua execução:

help - imprime os comandos

0.1 - apresenta o tamanho máximo da vista ativa

0.2 - apresenta o tamanho máximo da vista passiva

0.3 - apresenta o tamanho máximo da vista passivaActiva

1.1 - apresenta o IP do nó de contacto

1.2 - apresenta a lista de nós da vista ativa

1.3 - apresenta a lista de nós da vista passiva

1.4 - apresenta a lista de nós da vista passivaActiva

1.5 - apresenta o número de mensagens *Join* recebidas

1.6 - apresenta o número de mensagens *FowardJoin* enviadas e recebidas

1.7 - apresenta o número de mensagens *NeighborRequest* enviadas e recebidas

1.8 - apresenta o número de mensagens *Shuffle* enviadas e recebidas

1.9 - apresenta o número de mensagens *Disconect* enviadas e recebidas

2.1 - apresenta a lista de nós da vista global

2.2 - apresenta a lista de eventos

2.3 - apresenta a lista de eventos pendentes

2.4 - apresenta a lista de nós que poderão estás desconectados

2.5 - apresenta o número de mensagens *CheckIfAlive* envidas

2.6 - apresenta o número de mensagens *Conflit* envidas

3.1 - apresenta o número de mensagens *Broadcast* envidas

3.2 - apresenta a lista de nós disponíveis na *communication layer*

4.1 - simula quebra na conexão

4.2 - desconecta o nó

5.1 - escreve o estado do sistema para ficheiro

Como exemplo foi realizado o teste para um cenário onde existe uma rede constituída por cinco nós onde a dimensão da *active view* é 2 e a dimensão da *passive view* é 7 (estes valores devem ser ajustados dependendo da dimensão esperada da rede). Depois de verificada a estabilidade da mesma, foram removidos dois nós, adicionado um novo nó e, por último, removidos todos dos nós até ficarem apenas dois.
A rede constituída por cinco nós pode ser vista na figura 4, onde os arcos a vermelho representam as ligações da *active view* e os arcos pretos representam as ligações da *passive view*. A interpretação do grafo é feita da seguinte forma:

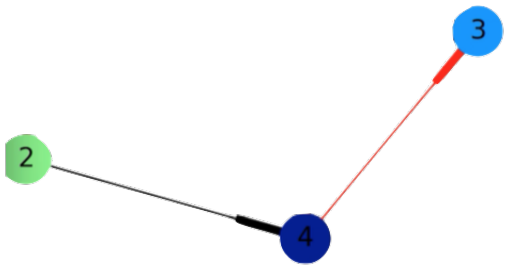

Figure 3: - o nó 2 tem o nó 4 na passive view - o nó 4 tem o nó 3 na active view

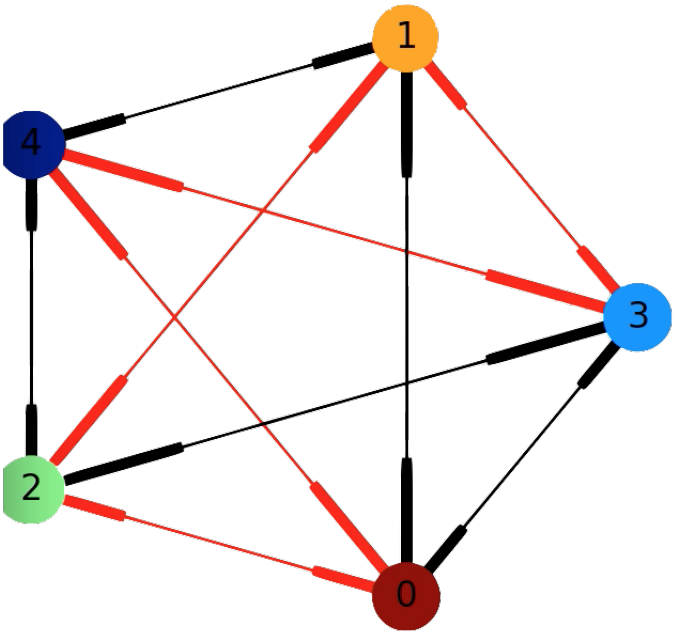

Figure 4: Grafo de uma rede constituída por 5 nós.

A quantidade de mensagens de cada tipo trocadas pelos nós que constituem a rede pode ser vista na figura 5. O primeiro nó, nó 0, é o nó de contacto para os restantes, onde estes foram acrescentados de forma sucessiva. O descrito é facilmente verificado através da interpretação do gráfico. O nó 0 foi o único que recebeu mensagens do tipo *Join* e pela quantidade de mensagens do tipo *Shuffle* enviadas e recebidas pelo nó 4 verificamos que este foi o último a ser adicionado. Verifica-se também que o nó 0 recebeu dois pedidos de desconexão e o nó 3 recebeu um pedido de resolução de conflitos. Este tipo de pedido ocorre quando as *hash* das *global view* de dois nós são diferentes, suspeitando-se de um conflito entre estes, sendo que o pedido de resolução de conflitos é a mensagem que contém a *global view* de um nó, pedindo ao outro nó para resolver o conflito.

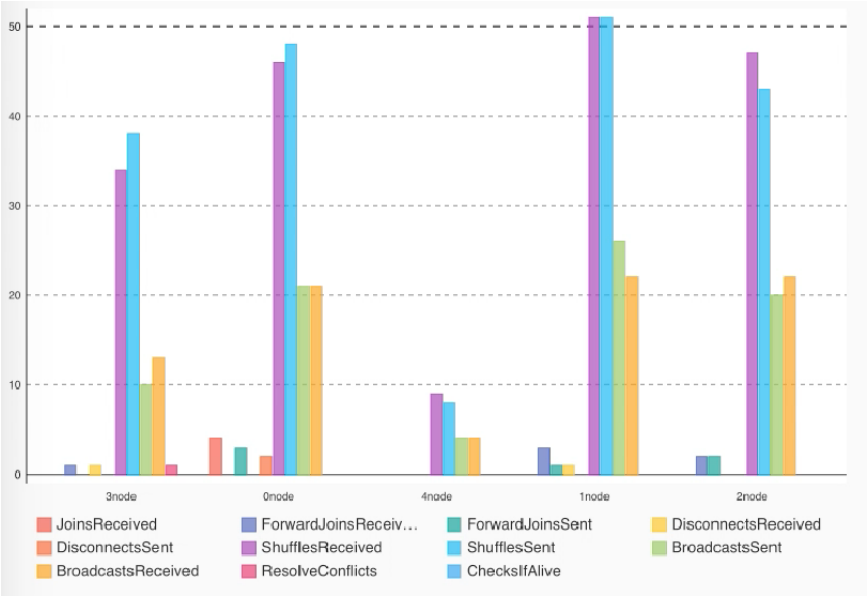

Figure 5: Mensagens trocadas para a constituição de um rede com 5 nós.

O propósito desta rede é que cada nó tenha conhecimento de todos os outros existentes na rede. O gráfico da figura 6 permite verificar os nós presentes nas *global view*s. Na legenda do gráfico verificam-se os identificadores dos nós presentes na *globalview* de cada nó, enquanto que a barra representa o tamanho da *global view*. Desta forma é facilmente verificável se todos os nós têm a mesma *global view* através do alinhamento das barras e conjunto de identificadores.

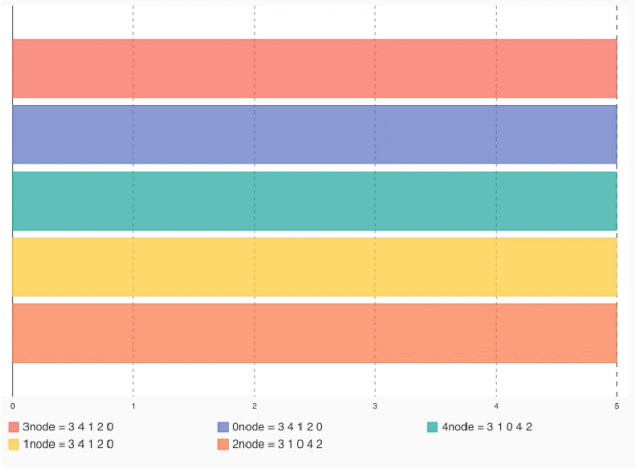

Figure 6: *Global view* de cada nó.

Através destes dados é possível concluir que a rede está correta e estável. Podemos agora verificar como a rede reage quando removemos 2 nós (figuras 7 e 8).

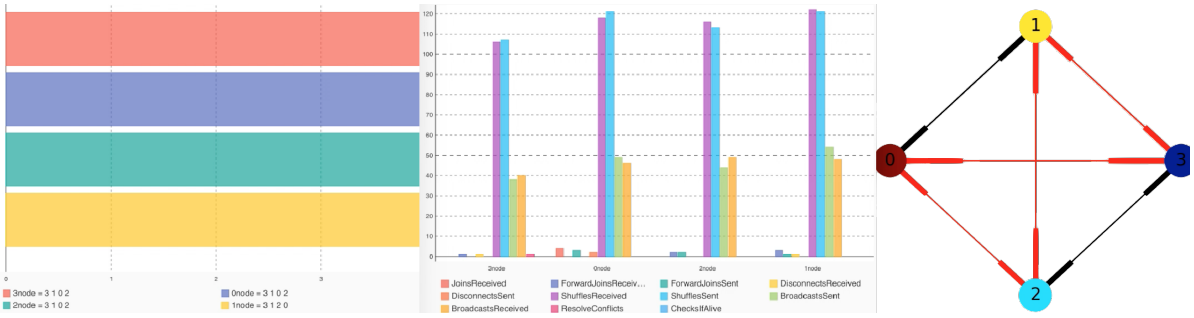

Figure 7: Dados após a primeira remoção

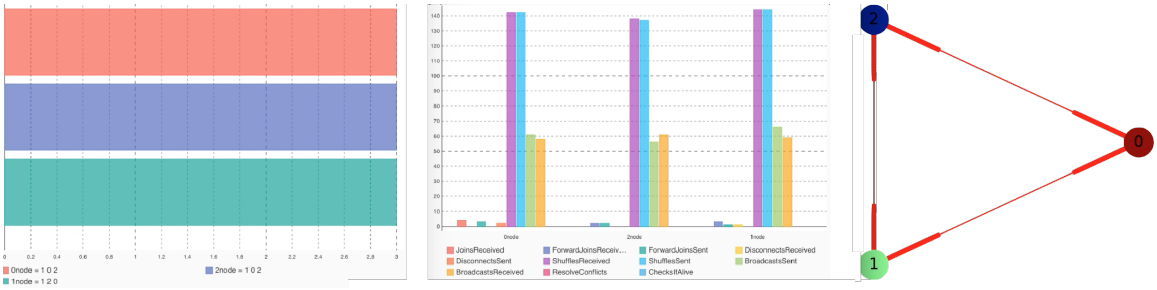

Figure 8: Dados após a segunda remoção

A passagem entre a rede de 4 nós estável para a rede de 3 nós também estável, demorou apenas 50 segundos.
De seguida adicionou-se um novo nó e o conjunto de dados apresentado na figura 9 apresenta o estado inconsistente

mas temporário da rede aquando da inserção do nó com o identificador 4. Já a figura 10 apresenta o conjunto de dados após a estabilização da rede que apenas demorou 10 segundos. Verifica-se também que foram trocadas muito poucas mensagens para o efeito.

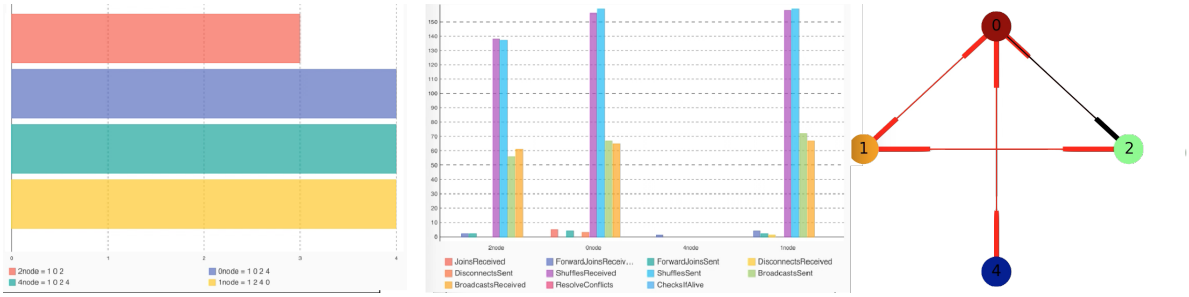

Figure 9: Dados (inconsistentes) após a inserção de um novo nó (id 4).

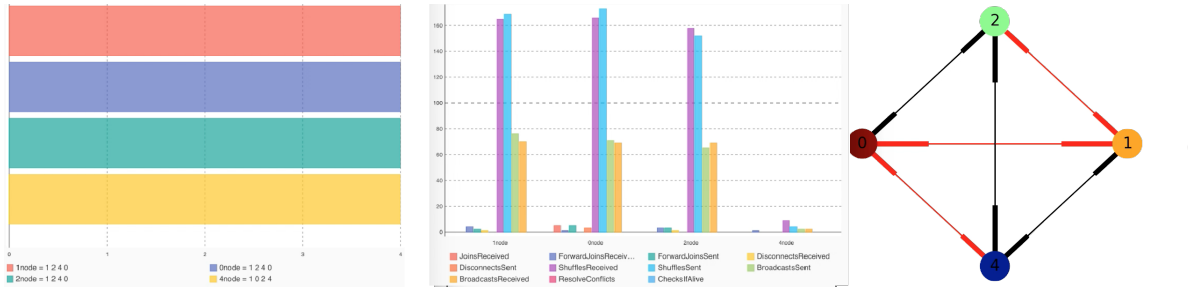

Figure 10: Dados (consistentes) após a inserção de um novo nó (id 4).

O teste realizado pode ser visto na sua totalidade em **www.youtu.be/pGRNyq62c7o**.

Concluiu-se que a quantidade de mensagens trocadas entre as camadas de *global view membership* para resolução de conflitos é pequena o suficiente para não causar excesso de tráfego e para não se estar constantemente a trocar a *global view* inteira entre os nós.
A quantidade de mensagens de *gossip* na camada de *information dissemination* é também bem mais reduzido do que algumas das mensagens da camada da *partial view membership*.

References

[1] L. R. João Leitão, José Pereira, "Hyparview: a membership protocol for reliable gossip-based," 5 2007.

Algorithm 1: Partial View Membership

Interface:
Requests:
Indications:
globalNewNode(node)
globalMaybeDead(node)
ActiveViewAdd(node)
ActiveViewRemove(node)

State:
activeView `//conjunto de vizinhos do nó (active partial view)`
passiveView `//vista passiva do nó`
onGoingNeighborRequests`//lista de nós dos quais se está à espera de uma resposta a um ne`
passiveactiveview`//nós da vista passiva que estão a ser monitorizados e serão os primeir`
myself `//identificador do próprio nó`

Upon Init (me**,** $contactNode$**) do:**
myself $\longleftarrow$ me;
activeView $\longleftarrow$ $\{contactNode\}$;
passiveView $\longleftarrow$ $\{\}$;
passiveactiveView $\longleftarrow$ $\{\}$;
onGoingNeighborRequests $\longleftarrow$ $\{\}$;
Trigger Send (Join, $contactNode, myself$**);**
Setup Pediodic Timer shufflePassiveActive (T$_1$);

Upon Receive(Join, $newNode$ **) do:**
Call addNodeActiveView($newNode$**);**
Trigger Send (ForwardJoinAccepted, $newNode, myself$**);**
Trigger globalNewNode($newNode, true$ **);**
foreach $n \in activeView \wedge n \neq newNode$ **do**
Trigger Send (ForwardJoin, $n, newNode, ARWL, myself$**);** `//ARWL: Active random walk length`

Upon Receive (ForwardJoin, $newNode, timeToLive, sender$**) do:**
If timeToLive = 0 $\vee$ #activeView = 1
Call addNodeActiveView($newNode$**);**
Else
If timeToLive = PRWL `//PRWL: Passive random walk length`
Call addNodePassiveView($newNode$**);**
n $\longleftarrow$ $n \in$ activeView $\wedge n \neq$ sender;
Trigger Send (ForwardJoin, $n, newNode, timeToLive - 1, myself$**);**

Upon Receive (ForwardJoinAccepted, $sender$**) do:**
Call addNodeActiveView($sender$**);**

Upon shufflePassiveActive () do:
newElements $\longleftarrow$ chooseRandomElementsFrom(passiveView);
passiveactiveView $\longleftarrow$ $\{newElements\}$;

Upon Crash (p**) do:**
priority $\longleftarrow$ low;
activeView $\longleftarrow$ activewView $\setminus \{p\}$;
If #activeView = 0
priority $\longleftarrow$ high;
Call getNewActiveNode($priority$**);**
Trigger globalMaybeDead(p**);**

Algorithm 2: Partial View Membership (continuação)

Procedure getNewActiveNode ($priority$**) do:**
n $\longleftarrow$ n $\in$ passiveActiveView$\wedge$n $\notin$ passiveView ;
If n = $\perp$
n $\longleftarrow$ n $\in$ $passiveView$;
While connectTo(n) = false
passiveView $\longleftarrow$ passiveView $\setminus \{n\}$;
n $\longleftarrow$ n $\in$ $passiveView$;
onGoingNeighborRequests $\longleftarrow$ onGoingNeighborRequests $\cup \{n\}$;
Trigger Send (NeighborRequest, $n, priority$**)**;

Upon Request (NeighborRequest, $s, priority$**) do:**
If priority = high $\vee$ isFull(activeView)=false
Call addNodeActiveView(s**);**
Trigger Send (AcceptedRequest, s**)**;
Else
Trigger Send (RefusedRequest, s**)**;

Upon Request (AcceptedRequest, s**) do:**
onGoingNeighborRequests $\longleftarrow$ onGoingNeighborRequests $\setminus \{s\}$;
passiveView $\longleftarrow$ passiveView $\setminus \{s\}$;
activeView $\longleftarrow$ activeView $\cup \{s\}$;

Upon Request (RefusedRequest, s**) do:**
If #activeView = 0
priority $\longleftarrow$ high;
Else
priority $\longleftarrow$ low;
onGoingNeighborRequests $\longleftarrow$ onGoingNeighborRequests $\setminus \{s\}$;
Call getNewActiveNode($priority$**);**

Procedure addNodeActiveView ($node$**) do:**
If node $\neq$ myself $\wedge$ node $\notin$ activeView
If isFull($activeView$)
Call dropRandomElementFromActiveView();
activeView $\longleftarrow$ activeView $\cup$ node;
Trigger ActiveViewAdd ($node$**)**;

Procedure dropRandomElementFromActiveView () do:
n $\longleftarrow$ n $\in$ activeView;
Trigger Send (Disconnect, $n, myself$**)**;
activeView $\longleftarrow$ activeView $\setminus \{n\}$;
passiveView $\longleftarrow$ passiveView $\cup \{n\}$;
Trigger ActiveViewRemove (n**)**;

Algorithm 3: Information dissemination

Interface:
Requests:
BroadCast(m)
Indications:
communicationDeliver ($m, sender$ **)** `//m é a mensagem a entregar, sender é o nó que`
`// enviou a mensagem`
State:
activeView `//conjunto de vizinhos do nó (active partial view)`

Upon Init () do:
activeView $\longleftarrow \{\}$;

Upon Broadcast (m) do:
foreach $p \in$ activeView
Trigger Send(Gossip, p**,** m**)**;

Upon Receive (Gossip, m**,** $sender$**) do:**
Trigger communicationDeliver(m**,** $sender$**)**;

Upon ActiveViewAdd ($node$**) do:**
activeView $\longleftarrow$activeView $\cup \{node\}$;

Upon ActiveViewRemove ($node$**) do:**
activeView $\longleftarrow$activeView $\setminus \{node\}$;

Algorithm 4: Global View Membership

Interface:
Requests:
Indications:
State:
globalView `//Lista de nós pertencentes à global view`
eventList `//Lista de eventos recentes`
pendingEvents `//Lista de eventos pendentes`
toRemove `//Lista de nós suspeitos de ter falhado que estão à espera de ser removidos`
potentialConflict `//Lista de nós em que se detetou um potencial conflito`
`//última vez que se comparou os hash`
myself `//identificador do próprio nó`

Upon Init (me**) do:**
globalView $\longleftarrow \{\}$;
eventList $\longleftarrow \{\}$;
pendingEvents $\longleftarrow \{\}$;
toRemove $\longleftarrow \{\}$;
potentialConflict $\longleftarrow \{\}$;
myself $\longleftarrow$ me;
Setup Pediodic Timer SendHash (T_1);

Upon SendHash ( ) do:
Cancel SendEvents();
Call globalBroadcast();

Upon SendEvents ( ) do:
Call globalBroadcast();

Upon Remove (p**) do:**
globalView $\longleftarrow$globalView $\setminus \{p\}$;
toRemove $\longleftarrow$ toRemove $\setminus \{p\}$;

Upon globalMaybeDead(p**) do:**
event $\longleftarrow \{(maybeDead, p)\}$;
eventId $\longleftarrow$ generateId($event$);
Call addToEventList($eventId, event$**);**
toRemove $\longleftarrow$toRemove $\cup \{p\}$;
Setup Timer remove (T_3, p**);**

Upon globalNewNode($newNode, needsGlobal$**) do:**
event $\longleftarrow \{(new, newNode)\}$;
eventId $\longleftarrow$ generateId($event$);
Call addToEventList($eventId, event$**);**
Call globalAdd($newNode, needsGlobal$**);**

Procedure globalBroadcast ( ) do:
hash $\longleftarrow$ generateHash($globalView$);
m $\longleftarrow \{(hash, pendingEvents, isEmpty(toRemove))\}$;
Trigger Broadcast(m);
pendingEvents $\longleftarrow \{\}$;

Procedure globalAdd ($newNode, needsGlobal$**) do:**
globalView $\longleftarrow$globalView $\cup \{newNode\}$;
If needsGlobal
Send(global, $p, globalView, eventList$**);**

Upon Receive (global, $s, newView, eventIds$**) do:**
globalView $\longleftarrow$newView ;
eventList $\longleftarrow$ eventIds;

Algorithm 5: Global View Membership (continuação)

Procedure addToEventList ($eventId, event$**) do:**
  If #pendingEvents = 0
  Setup Timer SendEvents (T_2);
  Else If (maybeDead, $myself$) = $event$
    Cancel Timer SendEvents ();
    Setup Timer SendEvents (T_2);
  If isFull($eventList$)
    eventList ⟵eventList $\setminus oldest(eventList)$;
    eventList ⟵eventList $\cup \{eventId\}$;
    pendingEvents ⟵pendingEvents $\cup \{(eventId, event)\}$;
  If isFull($pendingEvents$)
    Cancel Timer SendEvents ();
    Call globalBroadcast ();

Upon communicationDeliver(m, s**) do:**
  compareHash ⟵false ;
  (h, newEvents, toRemoveEmpty) ⟵ m;
  If pendingEvents ={} ∧ toRemoveEmpty
    compareHash⟵true ;
  Foreach $(eventId, event) \in$ newEvents **do**
    If eventId $\notin$ eventList
      Call addToEventList($eventId, event$**);**
      (type, p) ⟵ event;
      If type = new
        Call globalAdd($p, false$**);**
      Else If type = maybeDead
        If p$\neq$ myself
          toRemove ⟵toRemove $\cup \{p\}$;
          Setup Timer remove (T_3, p**);**
        Else
          Call imAlive();
      Else If type = stillAlive
        If p$\in$ toRemove
          toRemove ⟵toRemove $\setminus \{p\}$;
          Cancel Timer remove (p);
      Else
        Call globalAdd(p, false);
  If compareHash ∧ isEmpty(toRemove)
    Call globalCompare(h, s**);**

Procedure imAlive( ) do:
  event ⟵ $\{(stillAlive, myself)\}$;
  eventId ⟵ generateId($event$);
  Call addToEventList($eventId, event$**);**

Procedure globalCompare($hash, sender$**) do:**
  myHash ⟵ generateHash(globalView) ;
  If myHash $\neq$ hash
    If $sender \in$ potentialConflict
      potentialConfict⟵potentialConflict$\setminus \{sender\}$
      If biggerId(myself, sender) = sender
        Trigger Send(Conflict, $sender, globalView$**);**
      Else
        Trigger Send(GiveGlobal, $sender$**);**
    Else
      potentialConfict⟵potentialConflict $\cup \{sender\}$

Algorithm 6: Global View Membership (continuação)

Upon Receive(GiveGlobal, $sender$ **) do:**
  Trigger Send(Conflict, $sender, globalView$);

Upon Receive(Conflict, $sender, otherGlobal$ **) do:**
  Foreach node ∈ otherGlobal ∧ node ∉ globalView**do**
    isAlive ⟵ checkIfAlive(node) ;
    If isAlive
      Trigger globalNewNode($node, false$**);**;
    Else
      event ⟵ $\{(maybeDead, node)\}$;
      eventId ⟵ generateId($event$);
      Call addToEventList($eventId, event$**);**
  Foreach node ∉ otherGlobal ∧ node ∈ globalView **do**
    isAlive ⟵ checkIfAlive(node) ;
    If isAlive
      event ⟵ $\{(stillAlive, node)\}$;
      eventId ⟵ generateId($event$);
      Call addToEventList($eventId, event$**);**
    Else
      Trigger globalMaybeDead($node$**);**;