2014/2015

Gossip-Based Aggregation in Large Dynamic Networks

Sistemas Distribuídos

Unidade Curricular: Sistemas Distribuídos
Professor Responsável: Pedro Souto
Trabalho Realizado por:
- João Lima, mieec1302800;
- Luís Gomes, ee09294

[Página em Branco]

I. Índice

1 Introdução

Este trabalho foi realizado no âmbito da Unidade Curricular de Sistemas Distribuídos, tendo como objetivo a implementação de um simulador que ilustre uma rede de sensores distribuída de modo a elaborar um estudo sobre Agregação entre nós de uma rede, com base no algoritmo epidémico *Gossip*.

O protocolo *Gossip* consiste num protocolo de disseminação de informação, isto é, quando um nó da rede tem uma nova informação, este envia para os nós vizinhos, onde estes por sua vez enviam para a sua vizinhança, passando o boato de nó em nó. Em analogia a este comportamento temos as rede social de hoje em dia, onde uma dada opinião ou notícia pode ser partilhada desde um nível local até a um nível global.

O seu funcionamento básico consiste numa troca periódica de mensagens entre um conjunto de pares de nós, por exemplo, um nó 1 com uma determinada informação, escolhe um nó 2 na sua vizinhança e envia-lhe essa informação. O nó 2, quando recebe a mensagem envia-lhe uma resposta e atualiza os seus dados. O nó 1, quando recebe a mensagem de nó 2, atualiza os seus dados. Após isto cada nó da transação anterior emparelha com novos nós e repetem a sequência da troca de mensagens até à disseminação total da informação na rede.

Como há uma distribuição da informação, tolerância de falhas e escalabilidade do sistema bem como robustez e simplicidade, este protocolo é utilizado em sistemas distribuídos.

2 Motivação

O objetivo deste trabalho consiste em desenvolver um simulador de forma a comprovar os resultados obtidos no documento *"Gossip-based Aggregation in Large Dynamic Networks*"*.

Este documento retrata uma análise da agregação entre sensores numa rede, cujo objetivo é a visualização global do sistema quanto à difusão do estado de cada sensor.

Esta agregação engloba dois fatores, um é o tipo de seleção do par, que pode ser *Perfect Matching*, *Random Choice* e *Distributed Solution* e o outro é o tipo de cenário aplicado à rede que pode ser *peak* e *uniform*.

3 Modelo do Sistema

O sistema do simulador implementado é constituído por duas partes, a primeira é a criação e alocação dos vários nós da rede e a segunda é a inicialização quanto ao tipo de cenário, o tipo de seleção de pares e as *thread's* associadas a cada sensor.

O sistema é constituído por um número pré-definido de sensores, alocados aleatoriamente numa matriz $n \times n$, com a garantia de que cada sensor contém exclusivamente um ID e as coordenadas de posição.

Após alocação dos sensores, cada um destes determina a sua vizinhança em relação alcance. O alcance do sensor é igual para todos os sensores do sistema e é pré-definido. Esta grandeza tem como objetivo simular o comportamento real de uma tecnologia de transmissão de dados sem fios, como a área de transmissão é radial, para a verificação da vizinhança de um sensor, utiliza-se a equação da circunferência com epicentro no sensor a analisar os seus vizinhos.

Para aplicação do algoritmo *Gossip*, cada sensor contém duas *thread's*, *Active* e a *Passive*.

O objetivo da *thread Active* é iniciar aleatoriamente um emparelhamento de um determinado sensor com o seu vizinho, para em seguida haver uma troca de dados. A *thread Passive* é uma resposta comportamental à *thread* anterior, isto é, o sensor a correr a *thread Passive* aguarda pela receção de uma mensagem de um vizinho, para iniciar a troca de dados. Dados esses que se baseiam em duas variáveis, o *stateP* e o *stateQ*. O *stateP* representa o estado local do sensor e o *stateQ* o estado do sensor vizinho. A manipulação das duas variáveis bem como o funcionamento das duas *thread's* acima mencionadas serão descritos no capítulo seguinte.

Outro critério do algoritmo aplicado é o tipo seleção do par de sensores. O simulador desenvolvido é capaz de executar em modo *Random Choice* e em *Dristributed Solution*. O tipo de seleção de par *Random Choice* consiste na criação aleatória de um par de nós para se comunicarem, sem qualquer restrição.

O tipo de seleção de par *Dristributed Solution*, muito semelhante ao anterior tipo seleção, tendo como requisito que a cada ciclo do sistema, todos os sensores tem que comunicar pelo menos uma vez.

Outro parâmetro do algoritmo é o tipo de cenário que pode ser *peak* ou *uniform*. No cenário *uniform* a cada sensor é atribuído um valor aleatório à variável *stateP*.

No cenário *peak*, é atribuído a um único sensor da rede um valor diferente de zero à variável *stateP*, enquanto para os restantes sensores, as variáveis *stateP's* são inicializadas a zero.

3.1 Alocação dos sensores

Sendo o objetivo deste trabalho a partilha do estado de cada sensor, entre os vários sensores da rede, é necessário distribuir os sensores pela rede. Esta distribuição foi definida a partir de um algoritmo linear, que após receber a informação do número máximo de sensores e do tamanho máximo da rede, aloca os sensores a uma determinada posição, descriminada nas coordenadas cartesianas (X,Y). Esta alocação simula a possível posição física dos sensores numa rede com as mesmas características. O algoritmo desenvolvido realiza uma distribuição aleatória dos vários sensores assegurando que dois sensores não podem ter o mesmo conjunto de coordenadas (X,Y).

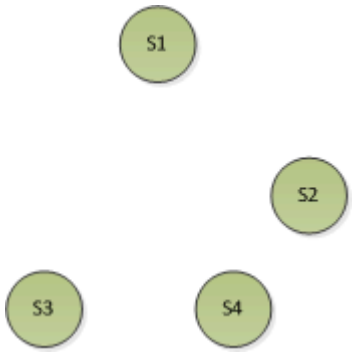


Figura 1 Alocação dos sensores

3.2 Alcance dos sensores

Uma das caracteristicas deste trabalho é a capacidade dos sensores comunicarem entre si por Wifi, realizando assim uma comunicação sem fios. Comunicação esta que num sistema físico depende do alcance da mesma. Estando perante a situação de uma simulação do estado físico do sistema e não fazendo parte dos conteúdos abordados

pela unidade curricular, decidiu-se simular esta incapacidade física com uma variável *range* incluída no trabalho representando o alcance máximo de cada sensor. Aproximou-se então o alcance máximo de cada sensor por uma circunferência de raio *range,* descrita pela seguinte expressão, onde (X,Y) representam as coordenadas cartesianas do sensor em questão:

$$(x - X)^2 + (y - Y)^2 = r^2$$

Com esta adaptação ao modelo físico por parte do simulador, deu-se origem a um termo muito importante no decorrer deste trabalho, vizinhança. Esta vizinhança para além de simular o alcance máximo da comunicação simboliza, quais os sensores que cada sensor consegue comunicar e partilhar o seu estado. Esta verificação de quantos e quais os vizinhos de cada sensor é realizada após a alocação de todos os sensores e guardada em cada sensor.

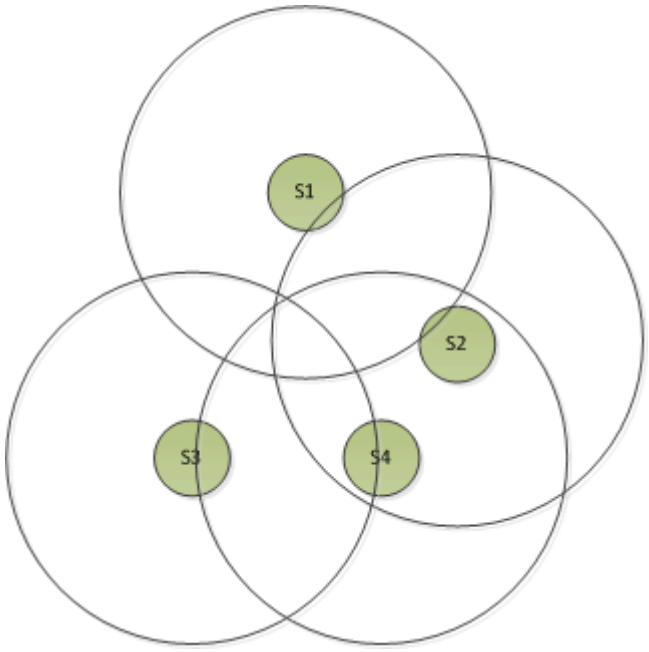


Figura 2 Alcance de cada sensor

3.3 UDP Multicast

Com o objetivo de aproximar ao máximo a simulação a uma situação real foi decidido realizar a comunicação entre sensores por um protocolo real. A escolha do protocolo multicast é justificada por ser um protocolo que é possível e plausível de implementar a nível computacional tendo uma grande afluência de mensagens. Sendo esta simulação projetada para vários sensores seria impossível realizar uma implementação UDP visto que para isso seriam necessários IP's diferentes o que numa única máquina seria impossível. O protocolo TCP também poderia ser implementado, pois utilizando uma porta diferente por sensor, seria possível realizar a comunicação, mas pensando a grande escala, este seria um protocolo que utilizaria muitos recursos computacionais e levaria o sistema a ficar mais lento.

No entanto a escolha deste protocolo teve também as suas desvantagens que rapidamente foram solucionadas com código, como é o caso da perda de mensagens, uma vez que utiliza o protocolo UDP sem garantias da entrega das mensagens e que todos os nós presentes num grupo recebem a mensagem mesmo que não direcionada.

O protocolo neste trabalho foi implementado com um único grupo em que todos os sensores se juntam quando são inicializados. Chamando o método Multicast_Init():

```
public MulticastSocket socket = new MulticastSocket(4446);
public InetAddress group = InetAddress.getByName("228.5.6.7");
socket.joinGroup(group);
```

Como todos os sensores estão associados ao mesmo grupo Multicast, optou-se por criar um formato da trama específico para esta aplicação.

A trama é constituída pelo IDENTIFICADOR, ID_LOCAL, ID_REMOTE, STATE_P e REPEAT.

O parâmetro do IDENTIFICADOR pode ser MATCH, ACK1 e ACK2.

A utilização do MATCH é feita quando um sensor envia um pedido de emparelhamento. O uso do ACK1 é necessário na confirmação de um pedido MATCH enquanto o uso do ACK2 é necessário para a confirmação do ACK1.

O parâmetro ID_LOCAL e ID_REMOTE da trama representa o remetente e destinatário da mensagem.

O quarto parâmetro, STATE_P, indica o estado do sensor remetente.

Por último, o quinto parâmetro, REPEAT, tem como objetivo indicar se a mensagem enviada é repetida ou não.

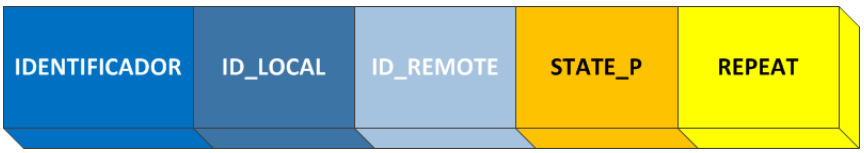


Figura 3 Formato da trama

Legenda:
IDENTIFICADOR: identifica o tipo de mensagem, MATCH, ACK1, ACK2;
ID_LOCAL: indica o remetente da mensagem;
ID_REMOTE: indica o destinatário da mensagem;
STATE_P: Estado do remetente;
REPEAT: indica se a mensagem é repetida ou não.

Considerando uma comunicação sem perdas de pacotes, quando um determinado sensor pretende iniciar a comunicação com outro, este envia uma mensagem com o identificador MATCH. Quando o sensor destinatário ao recebe a mensagem MATCH,

este envia como resposta uma mensagem com o identificador de ACK1. O sensor que iniciou a comunicação para confirmar a receção da mensagem ACK1 envia uma mensagem com identificador ACK2.

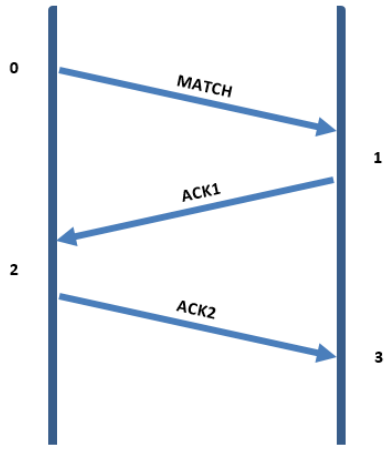


Figura 4 Diagrama de troca de mensagens

Caso haja uma comunicação com perdas de pacotes, há um controlo de timeout's para salvaguardar esta situação. Quando o sensor a iniciar a comunicação envia uma mensagem do tipo MATCH, este aguarda por um intervalo de tempo por uma resposta do tipo ACK1. Caso não receba a resposta há timeout, tendo o sensor que descartar esta comunicação e tentar iniciar uma nova com outro sensor.

Quando um sensor recebe uma mensagem tipo MATCH este fica à espera de uma mensagem do tipo ACK2, por um determinado intervalo de tempo, caso haja timeout, o sensor volta a enviar uma mensagem do tipo ACK1.

Estas duas situações de timeout serão posteriormente descritas no subcapítulo 4.4 Timeout's.

3.4 Emparelhamento

O protocolo proposto pode ser implementado com dois tipos de emparelhamento diferentes, aleatório ou distribuído. A grande diferença entre estes dois tipos reside na quantidade de emparelhamentos por ciclo por parte de cada sensor. No caso aleatório este emparelhamento é completamente aleatório e não tem qualquer restrição. Diferindo do distribuído que necessita que cada sensor realize um emparelhamento completo, com atualização do estado, pelo menos uma vez em cada período. Na implementação foram desenvolvidos os dois tipos. Sendo que a grande diferença é a utilização de uma variável de controlo do sensor no caso distribuído.

```
public boolean ciclo
```

Sendo esta ativada no inicio de cada ciclo para todos os sensores e desativada assim que cada sensor atualiza o seu estado.

3.5 Ciclo

Uma das caracteristicas deste protocolo é a noção de ciclo, simbolizando um requisito temporal do sistema. Esta variável é definida por um intervalo de tempo conveniente ao decorrer do protocolo, estando presente em todos os momentos do cálculo da variância e da convergência. O valor do intervalo de tempo denominado ciclo é dependente do número de sensores da rede, uma vez que no tipo de emparelhamento distribuído descrito neste trabalho, necessita que todos os sensores partilhem o seu estado pelo menos uma vez em cada ciclo. Com isto conclui-se que, neste caso, o ciclo define o número máximo de sensores na rede.

4 Implementação prática

Explicar como se implementa a active e passive thread e timer

Explicação para os dois algoritmos de seleção

4.1 Classe Sensor

Na implementação do Sensor neste simulador Java, tirou-se o partido da criação de uma classe Gsensor que pudesse ser referenciada nos outros processos. Esta classe é constituída por várias variáveis e métodos indispensáveis para o funcionamento deste trabalho. Como variáveis fixas temos o identificador do sensor, a posição, o alcance e o número de vizinhos que ao longo do processo não se alteram e são inicializadas a partir do construtor desta classe.

```
public int x, y, range, cnt_NEIGHBOR;
public int ID_local = -1;
        public Gsensor(int x, int y, int id, int range, float val) {

                this.x = x;
                this.y = y;
                this.ID_local = id;
                this.range = range;
                this.stateP = val;
        }
```

O valor do estado stateP é inicializado com um valor inicial, que vai sendo alterando ao longo do tempo de execução do algoritmo. Assim como o stateQ que representa o estado recebido pelo vizinho na comunicação em curso.

Para guardar os vizinhos de cada sensor optou-se por criar um vetor associado a cada sensor de forma a disponibilizar sempre que necessário quais os vizinhos de cada sensor.

```
public int[] DB_vizinhos = new int[SDIS_2015.MAX_COUNT_SENSORS];
```

Com o intuito de bloquear mais do que uma comunicação ao mesmo tempo, foi criada uma variável responsável por simbolizar a disponibilidade de cada sensor. Esta variável é bloqueada sempre que algum sensor inicia uma comunicação ou quando recebe uma comunicação de outro sensor e desbloqueada quando atualiza o seu estado.

```
public boolean wait;
```

4.1.1 Métodos da classe Gsensor

Dentro desta classe foram criados métodos que ficaram associados a cada sensor, métodos que são invocados pelas threads active e passive para por em prática o protocolo.

public int GetNeighbor() throws InterruptedException {}

Com a finalidade de encontrar um vizinho disponível, a thread active de cada sensor executa este método verificando de todos os seus vizinhos, quais os que não estão ocupados, tirando partido da variável wait.

public void Multicast_Init(){}

Este método realiza a inicialização do protocolo de comunicação para cada sensor criando um MulticastSocket associado a um grupo específico para realizar a comunicação entre sensores.

public void send_multicast(String op, int ip_random, float value, boolean flag) throws IOException {}

Para enviar mensagens por multicast foi criado este método que recebendo o formato da trama desejada, envia utilizando um DatagramPacket para o grupo de destino.

public String receive_multicast() throws IOException {}

Este método é utilizado pela thread passive com o intuito de receber os dados do protocolo multicast utilizando um DatagramPacket e retornando a mensagem recebida.

void processing(String recebido) throws IOException, InterruptedException{}

Após a rececão da mensagem pelo protocol multicast é necessário descodificar e processar a mensagem. Para isso foi desenvolvido um método para o controlo das mensagens enviadas para a rede utilizado por todos os sensores. É neste método que o nosso protocolo é verificado, partilhando os valores de cada sensor e realizando a média entre eles.
Como se utilizou um protocolo multicast, as mensagens enviadas são recebidas por todos, por isso realizou-se um pequeno protocolo para a descriminação da mensagem, verificando se a mensagem em questão é direcionada ao sensor em questão.
Caso a mensagem enviada seja recebida pelo sensor pretendido este desencadeia um conjunto de verificações e envio de mensagens descrito no subcapítulo 3.4.

public void UPDATE() throws InterruptedException {}

Este método é utilizando sempre que dois sensores executem uma partilha dos seus estados, realizando assim a média e atualizando o seu próprio estado.

4.2 Thread Active

A thread active referenciada no modelo do sistema serve para representar a escolha do vizinho e a partilha dos dados de cada sensor. Esta thread é inicializada com o ID do sensor que lhe ficou associado passado a partir do construtor:

int id_sensor;

```
public Active(int sensor) {
    this.id_sensor = sensor;
}
```

Esperando um tempo aleatório, tenta emparelhar e partilhar o seu estado. Como já explicado na classe Sensor existe uma variável que simboliza o estado ocupado ou livre de cada sensor, sendo esta a primeira verificação da thread Active.

```
if (SDIS_2015.sensores[id_sensor].wait == false) {
        SDIS_2015.sensores[id_sensor].wait = true;
        int ip_remote = sdis_2015.SDIS_2015.sensores[id_sensor].GetNeighbor();
}
```

Verificando a disponibilidade de realizar um novo emparelhamento, o sensor a partir da thread active vai tentar encontrar um novo vizinho capaz de partilhar o seu estado, bloqueando-se para que não haja a possibilidade de receber outra comunicação. Encontrando um vizinho livre e com a disponibilidade de emparelhar este envia uma mensagem multicast "MATCH" direcionada ao vizinho, chamando o método send_multicast:

```
SDIS_2015.sensores[id_sensor].send_multicast("MATCH", ip_remote, SDIS_2015.sensores[id_sensor].stateP, false);
```

Caso este não encontre um vizinho disponivel, liberta-se voltando a estar livre para uma nova comunicação voltando a esperar um tempo aleatório para tentar realizar uma nova comunicação com outro vizinho.

4.3 Thread Passive

Esta thread funciona em ciclo infinito e tem como objetivo receber todas as mensagens enviadas para o grupo multicast, chamando os métodos de receção e processamento da mensagem.

```
aux = SDIS_2015.sensores[id_sensor].receive_multicast();
SDIS_2015.sensores[id_sensor].processing( aux );
```

4.4 Timeouts

Estando a utilizar um protocolo de comunicação sem verificação de falhas, foi necessário desenvolver um pequeno controlo às possíveis mensagens perdidas na rede. Para isto foram implementados alguns temporizadores que funcionavam como um segundo envio, ou desistência da comunicação. Estes temporizadores foram desenvolvidos a base de threads concorrentes e paralelas ao funcionamento normal do processo e iniciados em duas situações diferentes.

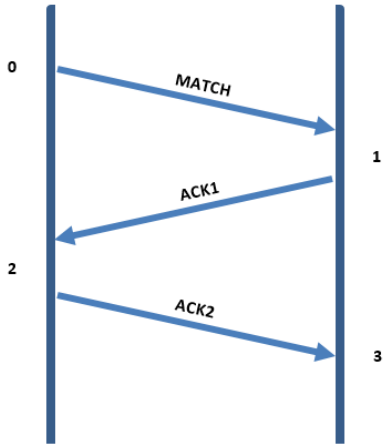


Figura 5 Diagrama de troca de mensagens

De acordo com o diagrama das trocas de mensagens pelos sensores existem duas situações em que pode existir perda de mensagens. A primeira é quando um sensor inicia uma comunicação enviando uma mensagem "MATCH" e não obtém uma resposta "ACK1" e a segunda é quando um sensor envia um "ACK1" e não recebe um "ACK2".

Foi decidido que no primeiro caso o sensor espera um determinado tempo e após esse tempo liberta-se, desistindo da comunicação e volta a executar a thread active. No segundo caso, como já existiu partilha de informação achou-se necessário voltar a enviar a mensagem após ter ultrapassado o temporizador.

A lógica por detrás destes temporizadores é igual nos dois casos, modificando apenas o resultado final. Como já referido anteriormente, estes temporizadores foram executados a partir de thread's paralelas por isso logica adicional teve de ser implementada nas thread active e na class Gsensor. Inicializando a thread com o identificador do sensor associado verificou-se, de 1 em 1 ms, se a mensagem já tinha sido recebida. Caso a mensagem tenha sido recebida ou mesmo quando existe timeout, a thread acaba ou é destruída.

```
int id_sensor;
public Timeout(int sensor) {
        this.id_sensor = sensor;
}

public void run() {
        int n = 0;
        while (SDIS_2015.sensores[id_sensor].received == false) {
                Thread.sleep(1);
                if (n == 500) {
                … // Código referente a cada situação de timeout
                Thread.currentThread().stop();
                Thread.currentThread().destroy();
                }
                n++;
        }
}
```

5 Resultados

Para finalizar o trabalho sobre este protocolo de agregação foram realizados alguns testes para posteriormente serem analisados e comparados com os resultados esperados. Os testes elaborados tiveram incidência no período de convergência de cada sensor em redes de 4, 9, 16, 25, 36 e 49. Estes valores escolhidos derivam da escolha do formato da rede, simulando em cada caso uma rede completa, sendo assim analisado o pior caso. Definiu-se a partida um alcance de raio 1 para cada sensor. Decidiu-se então dividir os testes em 3 fases pretendendo avaliar os 3 casos de estudo apresentados no paper estudado. Os resultados apresentados aparecem com uma média ponderada, as tabelas de dados podem ser encontradas em anexos.

5.1 Random Choise

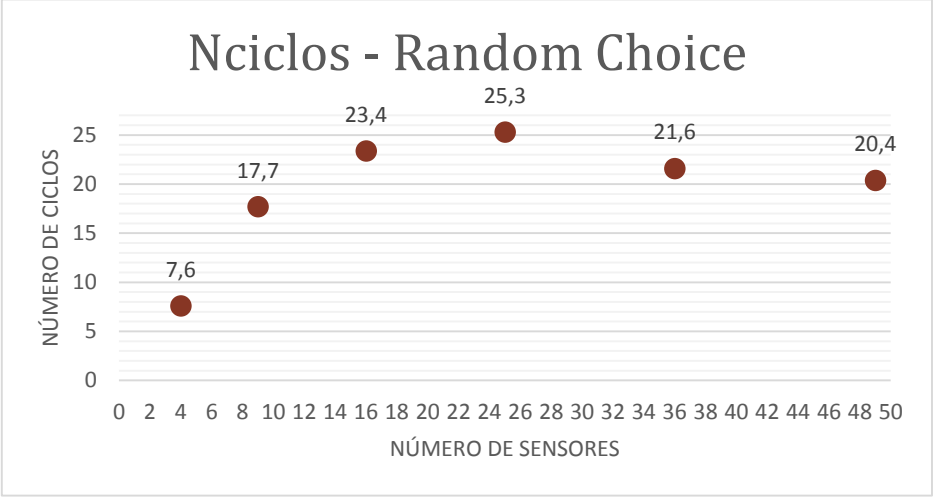


Figura 6 Número de ciclos da convergência Random Choice

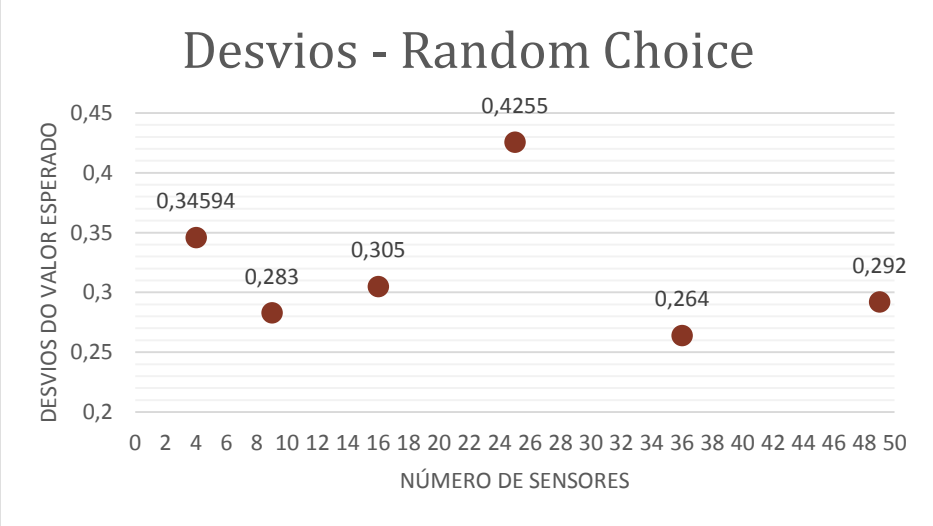


Figura 7 Desvios do valor esperado Random Choice

5.2 Distributed Solution – Uniforme

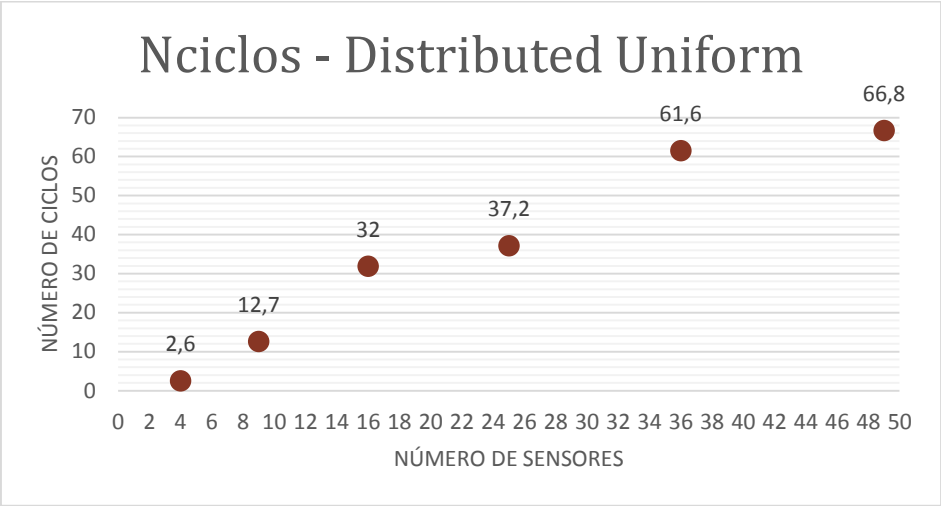


Figura 8 Número de ciclos da convergência Distributed Uniform

Sendo este um protocolo mais controlado, todos os valores retirados sofriam no máximo de um erro de 1% em relação ao valor esperado da convergência.

5.3 Distributed Solution – Pico

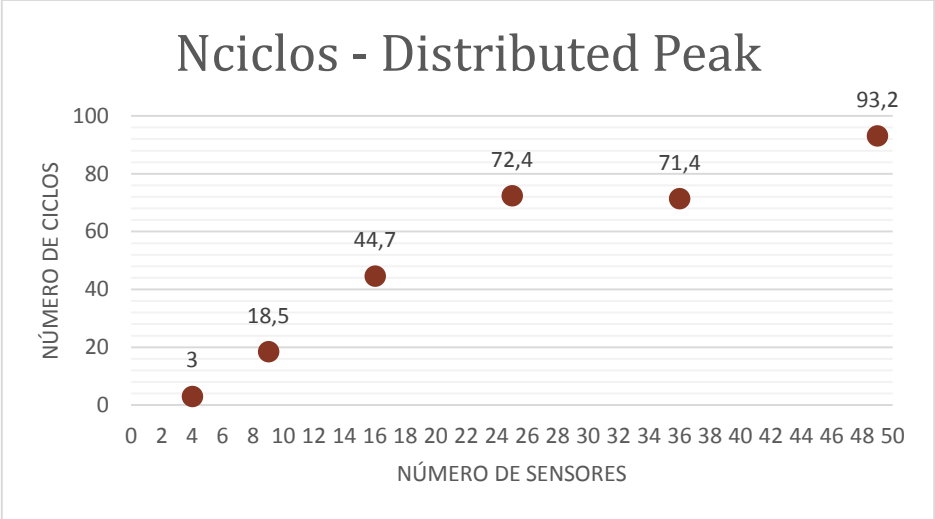
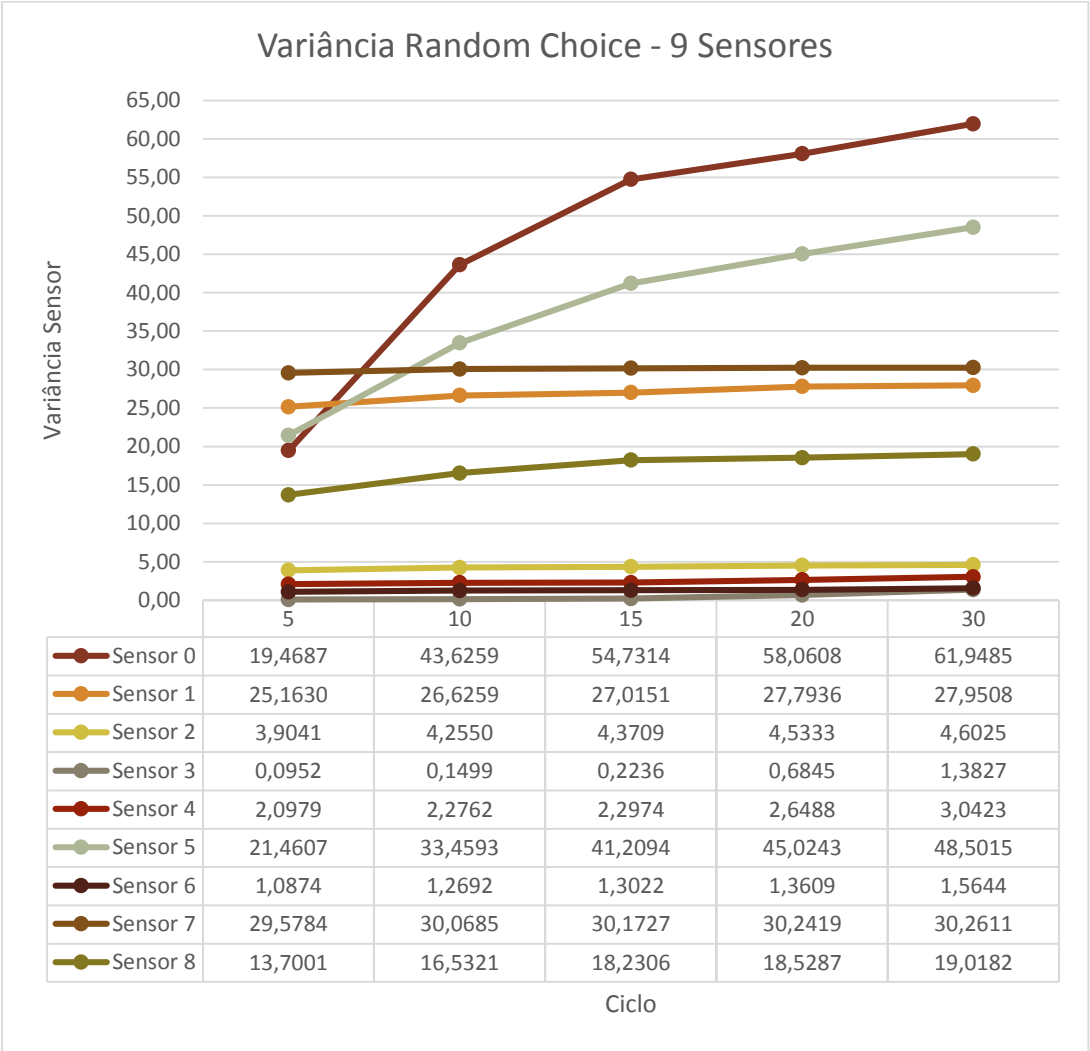


Figura 9 Número de ciclos da convergência Distributed Peak

Neste caso manteve-se o erro máximo de 1% com uma rede de 25 sensores, mas verificou-se que a partir destes valores o número de ciclos necessário para atingir este erro máximo era demasiado elevado, relaxando assim o valor máximo do erro para 5%. Sendo que as medidas retiradas para os 36 e 49 sensores têm 5% de erro máximo na convergência.

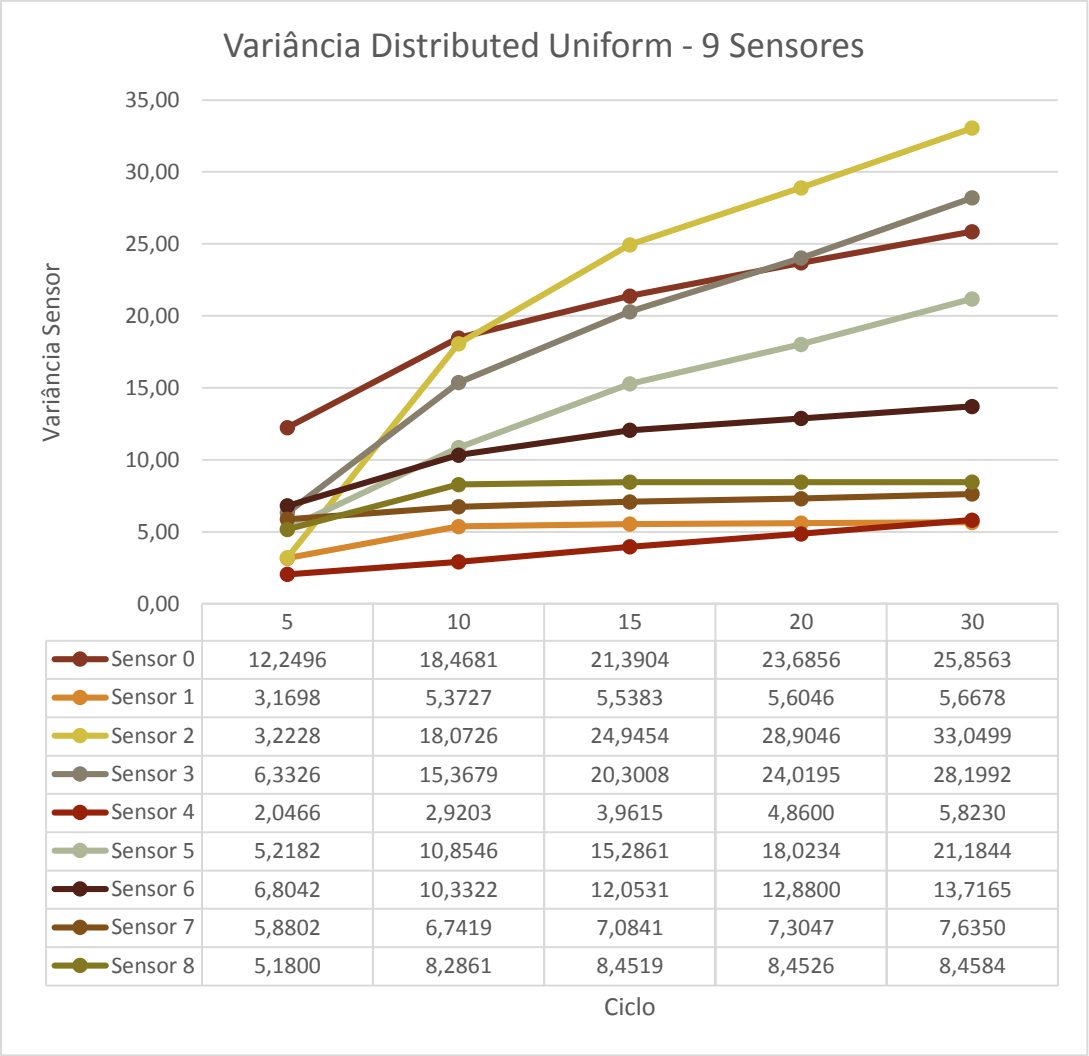
5.4 Variância

De acordo com as expressões retiradas do estudo sobre este protocolo, foram realizados alguns testes de maneira a retirar os valores da variância. Estes testes dizem respeito a variância de cada sensor e por isso as suas tabelas estão disponibilizadas em anexo. Para melhor visualização escolheu-se uma rede de 9 sensores para verificarmos a variância de cada sensor.



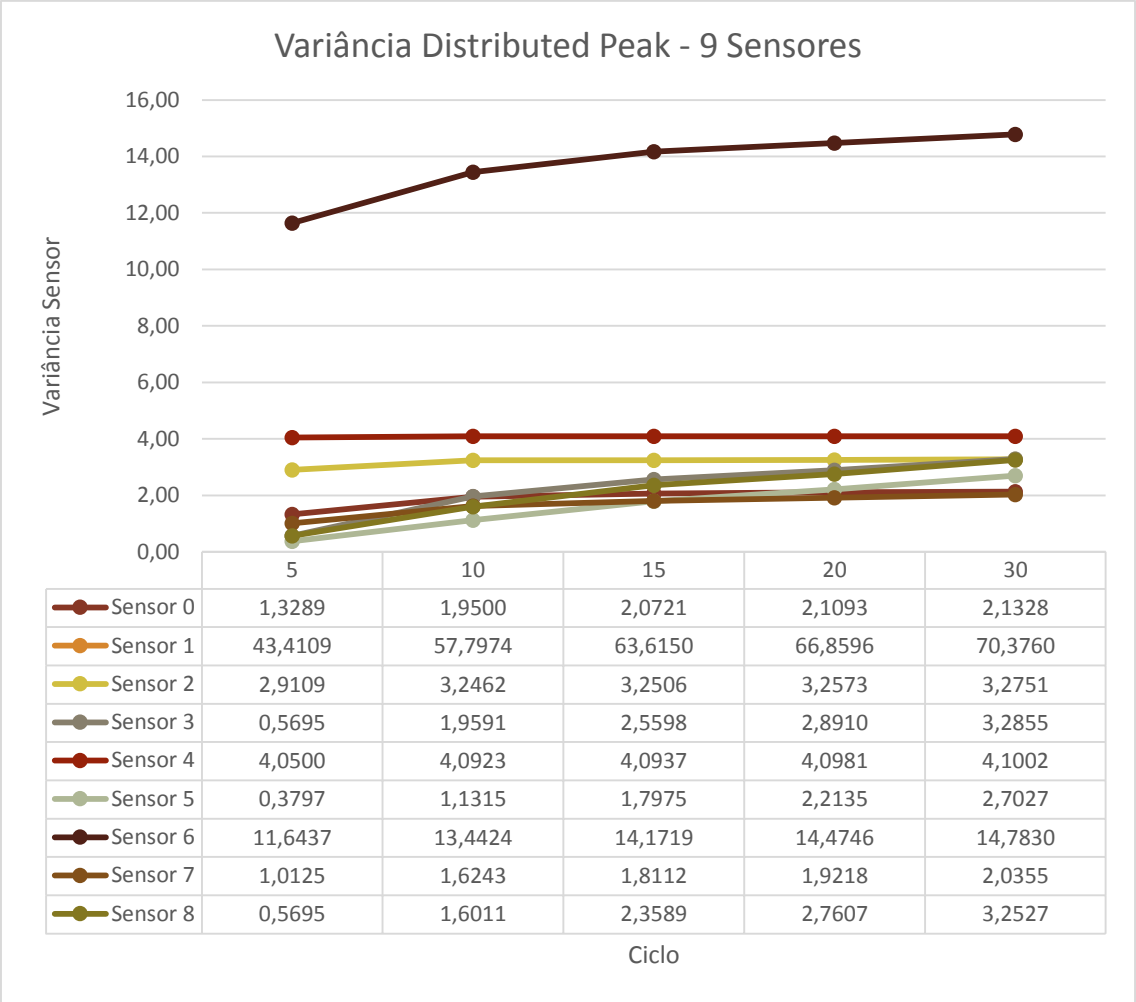
	5	10	15	20	30
Sensor 0	19,4687	43,6259	54,7314	58,0608	61,9485
Sensor 1	25,1630	26,6259	27,0151	27,7936	27,9508
Sensor 2	3,9041	4,2550	4,3709	4,5333	4,6025
Sensor 3	0,0952	0,1499	0,2236	0,6845	1,3827
Sensor 4	2,0979	2,2762	2,2974	2,6488	3,0423
Sensor 5	21,4607	33,4593	41,2094	45,0243	48,5015
Sensor 6	1,0874	1,2692	1,3022	1,3609	1,5644
Sensor 7	29,5784	30,0685	30,1727	30,2419	30,2611
Sensor 8	13,7001	16,5321	18,2306	18,5287	19,0182

Tabela 1 Variância Random Choice - 9 Sensores



	5	10	15	20	30
Sensor 0	12,2496	18,4681	21,3904	23,6856	25,8563
Sensor 1	3,1698	5,3727	5,5383	5,6046	5,6678
Sensor 2	3,2228	18,0726	24,9454	28,9046	33,0499
Sensor 3	6,3326	15,3679	20,3008	24,0195	28,1992
Sensor 4	2,0466	2,9203	3,9615	4,8600	5,8230
Sensor 5	5,2182	10,8546	15,2861	18,0234	21,1844
Sensor 6	6,8042	10,3322	12,0531	12,8800	13,7165
Sensor 7	5,8802	6,7419	7,0841	7,3047	7,6350
Sensor 8	5,1800	8,2861	8,4519	8,4526	8,4584

Tabela 2 Variância Distributed Uniform - 9 Sensores



	5	10	15	20	30
Sensor 0	1,3289	1,9500	2,0721	2,1093	2,1328
Sensor 1	43,4109	57,7974	63,6150	66,8596	70,3760
Sensor 2	2,9109	3,2462	3,2506	3,2573	3,2751
Sensor 3	0,5695	1,9591	2,5598	2,8910	3,2855
Sensor 4	4,0500	4,0923	4,0937	4,0981	4,1002
Sensor 5	0,3797	1,1315	1,7975	2,2135	2,7027
Sensor 6	11,6437	13,4424	14,1719	14,4746	14,7830
Sensor 7	1,0125	1,6243	1,8112	1,9218	2,0355
Sensor 8	0,5695	1,6011	2,3589	2,7607	3,2527

Tabela 3 Variância Distributed Peak - 9 Sensores

6 Conclusões

Em anexo segue os resultados dos testes realizados para a seleção do par Random e Distributed, com diferentes números de sensores. Os resultados visam a provar a convergência dos valores de cada estado do sensor.

Infelizmente não foi possível confirmar o fator de convergência, mas foi provado que uma seleção do par Dristributed é mais eficiente do que uma seleção Random. Esta eficiência é representada pela relação entre o número de ciclos necessários para atingir a convergência de todos os estados do sensor.

Com base nos resultados, é possível confirmar que a convergência dos estados dos sensores, não depende dos valores iniciais de cada estado do sensor e do tamanho da rede.