République Algérienne Démocratique et Populaire

École Polytechnique

Algorithmique Distribuée

Réalisé par
Marshall Badis
Réseaux et Sécurité Informatique

2022

Table des matières

Table des matières **2**

Table des figures **4**

Liste des tableaux **5**

1 Introduction **6**

I **6**

2 Diffuse_v0 **6**

3 Diffuse_v1 **7**

4 Diffuse_v2 **8**

5 Diffuse_v3 **9**

II **10**

6 Wake_Up **10**

III **12**

7 Tarry's Algorithme **12**

8 DFT_v0 **13**

9 DFT_v1 **14**

10 DFT_v2 **14**

11 DFT_v3 **14**

IV **15**

12 SPT_v0 **15**

13 SPT_v1 **17**

14 SPT_v2 **19**

15 SPT_DFT_v3 **20**

V **21**

16 RR_v0 **21**

17 RR_v1 **24**

18 RT_v0 **25**

19 RT_v1 **26**

VI **27**

20 Plus de Protocoles **28**
20.1 Center Finding 28
20.2 Full Saturation 29
20.3 Processus Noirs 30

21 Conclusion **32**

Table des figures

1 Omnet++ 6
2 Construction de SPT avec l'algorithme SPT_v0 16
3 Construction de SPT avec l'algorithme SPT_v0 17
4 Construction de SPT avec l'algorithme SPT_v0 17
5 Construction de SPT avec l'algorithme SPT_v1 18
6 Construction de SPT avec l'algorithme SPT_v2 20
7 Construction de SPT avec l'algorithme DFT_v3 21
8 Arbre minimal de Steiner 22
9 Statistiques de hopCount des messages avec RR_v0 22
10 Diagramme espace-temps RR_v0 23
11 Statistiques de hopCount des messages avec RR_v1 24
12 Diagramme espace-temps RR_v1 24
13 Statistiques de hopCount des messages avec RT_v0 25
14 Diagramme espace-temps RT_v0 26
15 Statistiques de hopCount des messages avec RT_v1 27
16 Diagramme espace-temps RT_v1 27
17 Topologie de l'arbre 29
18 Topologies de simulation d'algorithme Processus Noirs 31
19 Résultats de simulation d'algorithme Processus Noirs 31

Liste des tableaux

1 Table de Simulation Diffuse_v1 8
2 Table de Simulation Diffuse_v2 9
3 Table de Simulation Diffuse_v3 10
4 Table de Simulation Wake_Up 11
5 Table de Simulation Tarry's Algorithme 12
6 Table de Simulation DFT 15
7 Table de Simulation SPT_v0 16
8 Table de Simulation SPT_v1 18
9 Table de Simulation SPT_v2 19
10 Table de Statistique RR_v0 23
11 Table de Statistique RR_v1 25
12 Table de Statistique RT_v0 26
13 Table de Statistique RT_v1 28
14 Table de Statistique Center Finding 29

1 Introduction

Ce rapport est Un résumé de quelques algorithmes utilisés dans le module Algorithmique Distribuée, toutes les simulations incluent dans ce Rapport sont faites par OMNeT++.

OMNeT++ est une bibliothèque et un cadre de simulation C++ extensible, modulaire et basée sur des composants, principalement pour la création de simulateurs de réseaux.

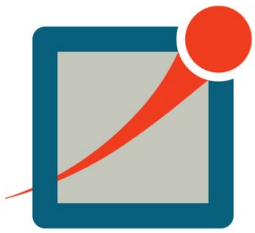


FIGURE 1 – Omnet++

Première partie

L'objectif de cette partie est d'implémenter les protocoles de diffusion discutés dans le chapitre 2. Nous commencerons par le plus simple (*Diffuse_ v0*) et sur cette base, nous implémenterons les autres versions du protocole diffuse.

Ce problème s'appelle la diffusion et fait partie d'une classe générale de problèmes appelée diffusion de l'information. Pour résoudre ce problème, il faut concevoir un ensemble de règles qui, lorsqu'elles sont exécutées par les entités, conduiront (en un temps fini) à ce que toutes les entités connaissent l'information, la solution doit fonctionner quelle que soit l'entité qui avait l'information au départ.

Restrictions

— l'ensemble des restrictions standard **R** = {BL, CN, TR} [Liens bidirectionnels (BL), Connectivité (CN) et Fiabilité(TR)]
— L'initiateur unique est celui qui détient l'information initiale **UI+**

2 Diffuse_v0

Algorithme

— initiator × S_p→ {**send(I)** to *N(x)*}
— idle × *Receving(I)* → {Process(I) ; **send(I)** to *N(x)*}
— initiator × *Receving(I)* → nil

— idle × S_p→ nil

Discussion

En raison de la connectivité et de la fiabilité totale, chaque entité finira par recevoir l'information. Le protocole atteint donc son objectif et résout le problème de la diffusion. Toutefois, ces règles posent un sérieux problème : les activités générées par le protocole ne se terminent jamais.

3 Diffuse_v1

Dans *Diffuse_ v0*, les nœuds s'enverront sans cesse des messages entre eux. Pour éviter cet effet indésirable, une entité ne devrait envoyer l'information à ses voisins qu'une seule fois : la première fois qu'elle acquiert l'information. Ceci peut être réalisé en introduisant un nouvel état *done*.

Algorithme

— initiator × S_p→ {**send(I)** to *N(x)* ; **become** done}
— idle × *Receving(I)* → {Process(I) ; **become** done ; **send(I)** to *N(x)*}
— initiator × *Receving(I)* → nil
— idle × S_p→ nil
— done × *Receving(I)* → nil
— done × S_p→ nil

Discussion

Cette fois, les activités de communication du protocole se terminent. toutes les entités sont terminées, puisqu'une entité terminée connaît l'information,

Complexité

Chaque entité, qu'elle soit initiatrice ou non, envoie l'information à tous ses voisins. Par conséquent, le nombre total de messages transmis est exactement.

$$\sum |N(x)| = 2|E| = 2m$$

Simulation

La Table 1 indique le nombre de messages transmet dans le réseau utilisant le protocole *Diffuse_ v1*.

Note : m est le nombre des liens, et **n** le nombre des nœuds.

Topologie	n	m	Nombres des Messages
Graphe_Complet_4	4	$\frac{n(n-1)}{2}$	12
Graphe_Complet_6	6		30
$\text{Graphe\_Complet}_{100}$	100		9900
Anneau_4	4	4	8
Anneau_{30}	30	30	60
Chaîne_3	3	2	4
Chaîne_{100}	100	99	198
$\text{HyperCube}_{k=2}$	2^2	4	16
$\text{HyperCube}_{k=3}$	2^3	12	48

TABLE 1 – Table de Simulation Diffuse_v1

4 Diffuse_v2

Nous pouvons en fait réduire le coût. Actuellement, lorsqu'une entité inactive reçoit le message, elle diffuse l'information à tous ses voisins, y compris à l'entité de laquelle elle a reçu l'information, ce qui est clairement inutile. Rappelons que, selon l'axiome d'orientation locale, une entité peut distinguer ses voisins, en particulier, lorsqu'elle traite un message, elle peut identifier le port d'où il a été reçu et éviter d'y envoyer un message.

Algorithme

— initiator × S_p→ {**send(I)** to *N(x)* ; **become** done}
— idle × *Receving(I)* → {Process(I) ; **become** done ; **send(I)** to *N(x)*-sender}
— initiator × *Receving(I)* → nil
— idle × S_p→ nil
— done × *Receving(I)* → *nil*
— done × S_p→ nil

Discussion

Cet algorithme est appelé Flooding, car le système entier est "inondé" par le message pendant son exécution, et c'est un outil algorithmique de base pour le calcul distribué.

Complexité

Comme Pour ce qui est du nombre de transmissions de messages requis par *Diffuse_v2*, puisque nous évitons de transmettre certains messages, nous savons qu'il est inférieur à 2m.

$$M[\text{Diffuse\_v2}] = 2m - n + 1$$

Soit *d(x, y)* la distance (c'est-à-dire la longueur du plus court chemin) entre x et y dans G.
Puisque n'importe quelle entité peut être l'initiateur, la complexité temporelle idéale dans le pire des cas sera $d(G) = \text{Max}\ \{r(x) : x \in E\}$, qui est le diamètre de G. En d'autres termes, la complexité temporelle idéale sera au plus égale au diamètre de G :

$$T[\text{Diffuse\_v2}] \leq d(G)$$

Simulation

Topologie	n	m	Nombres des Messages
Graphe_Complet_4	4	$\frac{n(n-1)}{2}$	9
Graphe_Complet_6	6		25
$\text{Graphe\_Complet}_{100}$	100		9801
Anneau_4	4	4	5
Anneau_{30}	30	30	31
Chaîne_3	3	2	2
Chaîne_{100}	100	99	99
$\text{HyperCube}_{k=2}$	2^2	4	13
$\text{HyperCube}_{k=3}$	2^3	12	41

TABLE 2 – Table de Simulation Diffuse_v2

Remarque :
La Table 2 montre bien l'amélioration de *Diffuse_ v2* en termes de nombre de Messages par rapport au *Diffuse_ v1*.

5 Diffuse_v3

En fait, nous pouvons pousser plus loin les performances de *Diffuse_ v2*. L'idée principale est que chaque entité doit éviter d'envoyer à ses voisins le message qu'elle a reçu ou qu'ils recevront.

Algorithme

— initiator × $S_p \rightarrow$ {**send(I,*N(x)*)** to *N(x)* ; **become** done}
— idle × *Receving(I,Z)* → {Process(I) ; **become** done ; if (Y = *N(x)* - Z ≠ ∅) **send(I,Y** ∪*Z*)*to* ***Y***}
— initiator × *Receving(I)* → nil
— idle × $S_p \rightarrow$ nil
— done × *Receving(I)* → *nil*
— done × $S_p \rightarrow$ nil

Simulation

La Table 3 indique le nombre de messages transmet dans le réseau utilisant le protocole *Diffuse_ v3*.

Topologie	n	m	Nombres des Messages
Graphe_Complet$_4$	4	$\frac{n(n-1)}{2}$	3
Graphe_Complet$_6$	6		5
Graphe_Complet$_{100}$	100		99
Anneau$_4$	4	4	4
Anneau$_{30}$	30	30	31
Grille_Carrée$_{c=5,l=4}$	20	31	40
Grille_Carrée$_{c=8,l=5}$	40	67	91
Torus$_{c=8,l=5}$	40	80	110
HyperCube$_{k=3}$	2^3	12	26
HyperCube$_{k=2}$	2^2	4	8
Arbre_Binaire$_{h=4}$	$2^{h+1}-1$	$n-1$	16
Arbre_Binaire$_{h=5}$			32
Arbre_Binaire$_{h=10}$			1024

TABLE 3 – Table de Simulation Diffuse_v3

Deuxième partie

Très souvent, dans un environnement distribué, nous sommes confrontés à la situation suivante : Une tâche doit être exécutée dans laquelle toutes les entités doivent être impliquées, cependant, seules certaines d'entre elles sont actives de manière indépendante (en raison d'un événement spontané ou parce qu'elles ont terminé un calcul précédent) et prêtes à calculer, les autres sont inactives et ne sont même pas conscientes du calcul qui doit avoir lieu. Dans ces situations, pour effectuer la tâche, nous devons nous assurer que toutes les entités deviennent actives. Il est clair que cette étape préliminaire ne peut être lancée que par les entités qui sont déjà actives, toutefois, elles ne savent pas quelles autres entités (le cas échéant) sont déjà actives.

Restrictions

— l'ensemble des restrictions standard $\mathbf{R}$ = {BL, CN, TR}

6 Wake_Up

Ce problème est appelé *Wake-Up* : Une entité active est généralement appelée éveillée, une entité inactive (immobile) est appelée endormie ; la tâche consiste à réveiller toutes les

entités. Il n'est pas difficile de voir la relation entre la diffusion et le réveil : La diffusion est un réveil avec une seule entité initialement éveillée, à l'inverse, le réveil est une diffusion avec éventuellement de nombreux initiateurs (c'est-à-dire initialement, plus d'une entité possède l'information). En d'autres termes, la diffusion n'est qu'un cas particulier du problème du réveil.

Algorithme

— Asleep $\times$ $S_p \rightarrow$ {**send(W)** to *N(x)* ; **become** Awake}
— Asleep $\times$ *Receving(W)* $\rightarrow$ {**become** Awake ; **send(W)** to *N(x)*-sender}

Complexité

Concentrons-nous sur le coût du protocole *Wake_ Up*. Le nombre de messages est au moins égal à celui de la diffusion, en fait, il n'est pas beaucoup plus élevé :

$$2m - n + 1 \leq \text{M[Wake\_Up]} \leq 2m$$

Comme la diffusion est un cas spécial de réveil (*Wake_ Up*), il n'y a pas beaucoup d'améliorations possibles. (sauf peut-être dans la taille de la constante) :

$$\text{M[Wake\_Up]} \ \geq \text{M[Diffuse]} = \Omega(m)$$

Le temps idéal sera, en général, plus petit que celui de la diffusion :

$$\text{T[Diffuse]} \geq \text{T[Wake\_Up]}$$

Simulation

La Table 4 résume la simulation de problème de *Wake-Up*.

Topologie	n	m	Nombres des Messages
$\text{Graphe\_Complet}_{12}$	12	$\frac{n(n-1)}{2}$	261
$\text{Graphe\_Complet}_{8}$	8		110
$\text{Graphe\_Complet}_{7}$	7		82
$\text{HyperCube}_{k=3}$	2^3	12	46
$\text{HyperCube}_{k=4}$	2^4		122

TABLE 4 – Table de Simulation Wake_Up

Troisième partie

La traversée du réseau permet de visiter chaque entité du réseau de manière séquentielle (l'une après l'autre). Ses principales utilisations sont le contrôle et la gestion d'une ressource partagée et les processus de recherche séquentielle. Dans un algorithme de traversée, l'initiateur envoie un jeton à travers le réseau. Après avoir visité tous les autres processus séquentiellement (c'est-à-dire un par un), le jeton revient à l'initiateur.

Restrictions

— l'ensemble des restrictions standard **R** = {BL, CN, TR}
— L'initiateur unique est celui qui détient l'information initiale **UI+**

7 Tarry's Algorithme

L'algorithme de Tarry est un algorithme de traversée pour les réseaux non orientés. Il est basé sur les deux règles suivantes :
— Un processus ne transmet jamais deux fois le jeton par le même canal.
— Un processus ne transmet le jeton à son parent que lorsqu'il n'y a pas d'autre option.

Complexité

La complexité de cet algorithme est M[TerryTraversal/R] = $\Omega(2m)$, du fait qu'on a au minimum deux messages sur chaque lien.

Simulation

La Table 5 résume quelques simulations avec l'outil Omnet++.

Topologie	n	m	Nombres des Messages
Graphe_Complet$_4$	4	$\frac{n(n-1)}{2}$	9
Graphe_Complet$_8$	8		53
Graphe_Complet$_7$	7		39
Grille_Carrée$_{c=8,l=5}$	40	67	134
Anneau$_{k=12}$	12	12	24
Anneau$_{k=4}$	4	4	8

TABLE 5 – Table de Simulation Tarry's Algorithme

Algorithme

Algorithm 1 Algorithme de Tarry

Data: var $used_p[q]$: boolean
init false pour tout q $\in Neigh_p$
$father_p$: process init udef
Pour l'iniateur seulement, executer une fois :
Debut
$father_p$ = p ; choisir q $\in Neigh_p$;
$used_p[q]$ = true ; send < **tok** > to q ;
fin
Pour tout Process, recevé le <**tok**> depuis q_0 :
Debut if $father_p$ = udef **donc** $father_p = q_0$;
if $\forall$ q $\in Neigh_p$:$used_p[q]$
Donc mort
else if $\exists$ q $\in Neigh_p$: ($q \neq father_p \wedge \neg used_p[q]$)
Donc debut if $father_p \neq q_0 \wedge \neg used_p[q]$
donc q := q_0
else choisir $q \in Neigh_p \setminus \{father_p\}$
avec $\neg used_p[q]$;
$used_p[q]$:= true ; send <**tok**> to q
fin
else debut $used_p[father_p]$ **:= true ;**
send <**tok**> to $father_p$
fin
fin

8 DFT_v0

Une stratégie bien connue est la traversée en profondeur d'un graphe. Selon cette stratégie, le graphe est visité (c'est-à-dire que le jeton est transmis) en essayant d'aller le plus loin possible. s'il est transmis à un nœud déjà visité, il est renvoyé à l'expéditeur et ce lien est marqué comme un *back-edge*, si le jeton ne peut plus être transmis (il se trouve à un nœud dont tous les voisins ont été visités), l'algorithme revient en arrière jusqu'à ce qu'il trouve un nœud non visité où le jeton peut être transmis.

Compelxité

T[DFT_v0] = M[DFT_v0] = $2m$, Donc *DFT_v0* est efficace en termes de nombre de messages.

9 DFT_v1

Quand un nœud reçoit un message T pour la 1 ère fois, il informe ses voisins (p. ex., par l'envoie d'un message *Visited*) et attend un acquittement (p. ex., message *Ack*) avant de réacheminer T.

maintenant un nœud prêt à expédier le message T sait vraiment lesquels de ses voisins ont été déjà visités.

Complexité

$$\text{T[DFT\_v1]} = 4n - 2 \text{ et } \text{M[DFT\_v1]} = 4m$$

10 DFT_v2

Nous avons une amélioration parce que les messages Ack ne sont plus envoyés, nous économisons n unités de temps.

Complexité

$$\text{T[DFTv2]} = 4n - 2 \text{ et } \text{M[DFT\_v2]} \leq 4m - n + 1$$

Donc dans les grandes topologies de réseaux, il est préférable d'utiliser l'algorithme *DFT_ v2* contre *DFT_ v1*, car on économise jusqu'à n-1 de message.

11 DFT_v3

Pour économiser quelques messages additionnels, nous pouvons utiliser le message T comme un message *Visited* implicite.

Cette épargne se produira au niveau de chaque nœud excepté ceux qui, quand ils sont atteints pour la première fois par un message T, n'ont pas de voisin non visité.

Complexité

Soit f^* le nombre de ces nœuds, ainsi le nombre de messages *Visited* économisés est $n - f^*$. Par conséquent, le nombre de messages est $4m - n + 1 - n + f^*$.

$$\text{M[DFT\_v3]} = 4m - 2n + f^* + 1 \text{ et T [DFT\_v3]} = 2n - 2$$

Simulation

La Table 6 résume les simulations des protocoles *DFT*.

Topologie	n	m	M_{DFT_v0}	M_{DFT_v1}	M_{DFT_v2}	M_{DFT_v3}
$Graphe\_Complet_4$	4	$\frac{n(n-1)}{2}$	12	30	18	12
$Graphe\_Complet_5$	5		20	48	28	20
$Graphe\_Complet_7$	7		42	96	54	42
$Anneau_{k=4}$	4	4	8	24	14	8
$Anneau_{k=20}$	20	20	40	142	78	40
$Grille\_Carrée_{c=8,r=5}$	40	67	134	346	222	134
$Grille\_Carrée_{c=5,r=4}$	20	31	62	162	100	62

TABLE 6 – Table de Simulation DFT

Quatrième partie

La construction de *Spanning-tree (SPT)* est un problème classique en informatique. Dans un environnement de calcul distribué, la résolution de ce problème a, comme nous l'avons vu, de fortes motivations pratiques. Construire un arbre spanning de G signifie faire passer le système d'une configuration initiale, où chaque entité ne connaît que ses propres voisins, à une configuration du système où :

— chaque entité x a sélectionné un sous-ensemble d'arbres voisins $(x) \subset N(x)$
— la collection de tous les liens correspondants forme un arbre couvrant de G

Restrictions

— l'ensemble des restrictions standard **R** = {BL, CN, TR}
— L'initiateur unique est celui qui détient l'information initiale **UI**+

12 SPT_v0

— P_{init} demande à ses voisins dans G s'ils sont ses voisins dans ST
— Le premier message reçu par Pi est le seul pris en compte, ceci établit $Pere_i$ (= émetteur de ce message)
— P_i propage l'exploration vers un sous-ensemble de ses voisins, ceci établit $fils_i$

Complexité

M[SPT_v0] = 2 M(Diffuse) et T[SPT_v0] = T[Diffuse] +1

SPT_v0 assure une terminaison locale : chaque entité sait quand son propre exécution est terminée, cela se produit quad, elle passe à l'état *Done*.

Topologie	n	m	Nombres des Messages
Graphe_Complet$_4$	4	$\frac{n(n-1)}{2}$	6
Graphe_Complet$_8$	8		14
Graphe_Complet$_7$	7		12
Grille_Carrée$_{c=5,r=4}$	20	31	80
Grille_Carrée$_{c=8,r=5}$	40	67	167
Anneau$_4$	4	4	8
Anneau$_{30}$	30	30	62

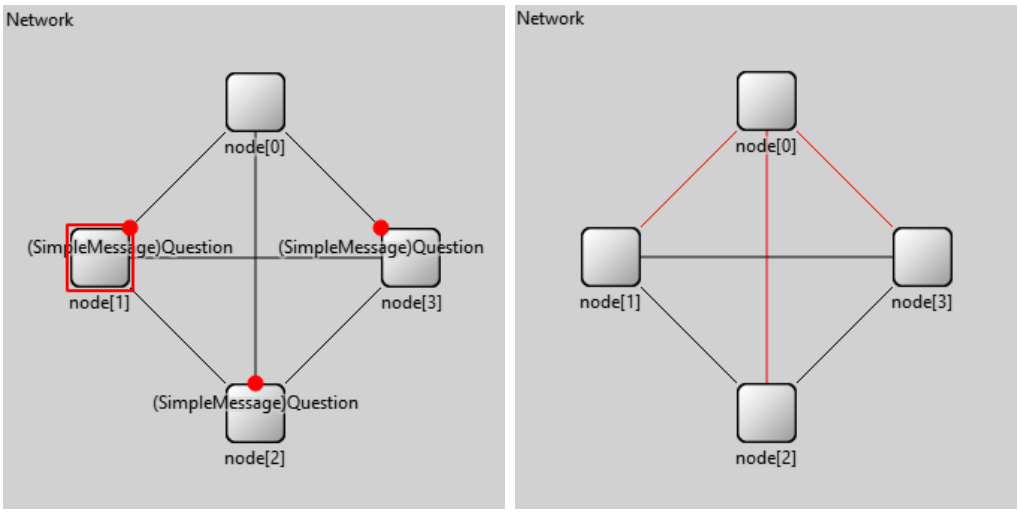
TABLE 7 – Table de Simulation SPT_v0

Simulation

Simulation OMNET++

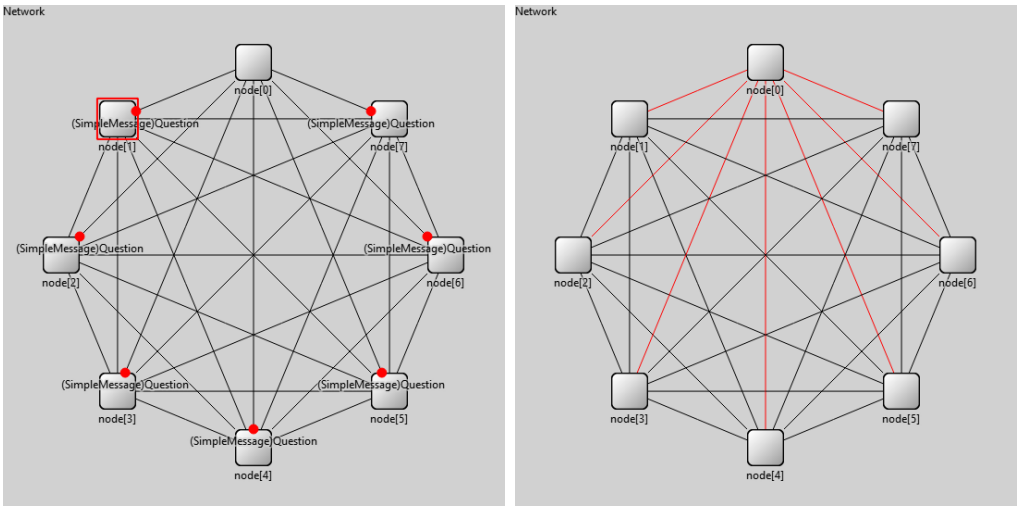
Les Figures 2, 3 et 4 montre la construction de *Spanning-Tree* des différentes topologies par le protocole *SPT_v0*.

La Table 7 résume aussi les simulations de protocole *SPT_v0*.



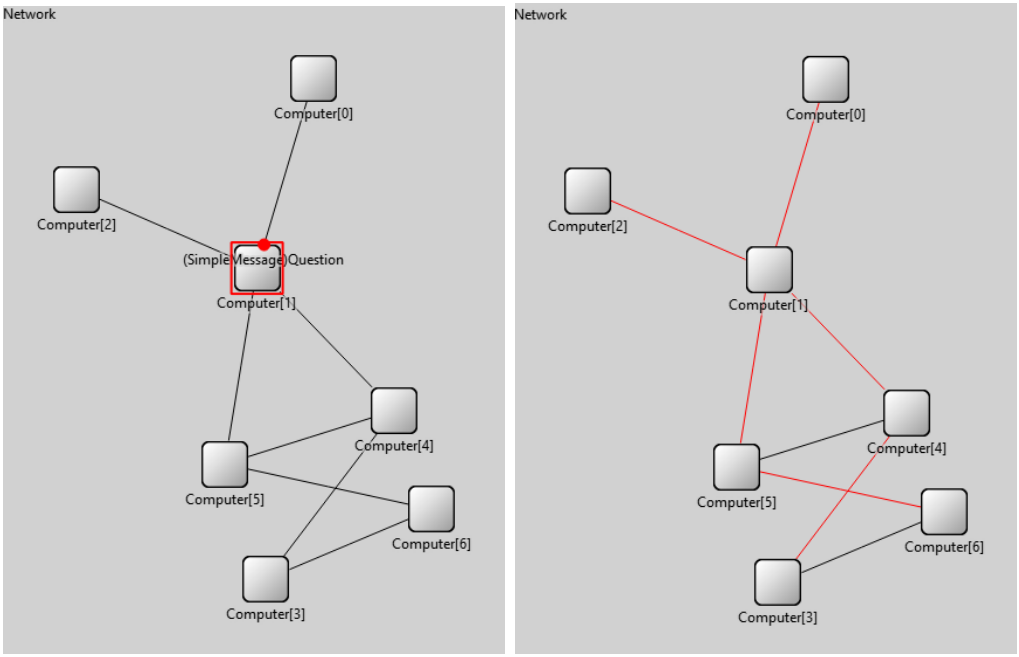
(a) Graphe_Complet de 4 nœuds (b) Spanning Tree du Graphe Complet

FIGURE 2 – Construction de SPT avec l'algorithme SPT_v0



(a) Graphe_Complet de 8 nœuds (b) Spanning Tree du Graphe Complet

FIGURE 3 – Construction de SPT avec l'algorithme SPT_v0



(a) Topologie quelconque avec 7 nœuds (b) Résultat de SPT

FIGURE 4 – Construction de SPT avec l'algorithme SPT_v0

l'exécution du protocole *SPT_v0* ne se termine pas par une terminaison globale.

13 SPT_v1

— La réponse *No* est renvoyée immédiatement
— La réponse *Yes* n'est renvoyée que lorsque tous les acquittements attendus sont arrivés

Complexité

M[SPT_v1] = 2 M(Diffuse) et T[SPT_v1] = T[Diffuse] +1

SPT_v1 assure une terminaison locale : chaque entité sait quand son propre exécution est terminée, cela se produit quand elle passe a l'état *Done*, et assure aussi une terminaison globale lorsque l'initiateur reçoit toutes les réponses de ces voisins.

Simulation

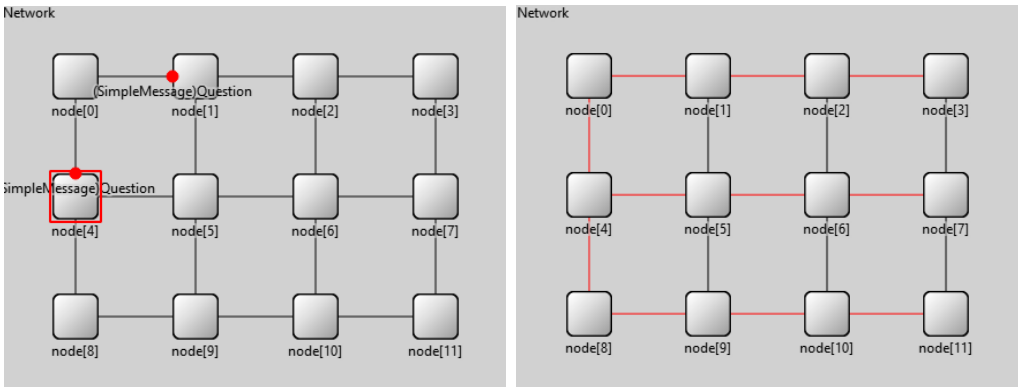
La Table 8 résume les simulations de protocole *SPT_v1*.

Topologie	n	m	Nombres des Messages
Graphe_Complet$_4$	4	$\frac{n(n-1)}{2}$	6
Graphe_Complet$_8$	8		14
Graphe_Complet$_7$	7		12
Grille_Carrée$_{c=5,r=4}$	20	31	80
Grille_Carrée$_{c=8,r=5}$	40	67	167
Anneau$_4$	4	4	8
Anneau$_{30}$	30	30	62

TABLE 8 – Table de Simulation SPT_v1

Simulation OMNET++

La Figure 5 montre l'exécution de protocole *SPT_v1* sur une topologie de *Grille_Carrée*.



(a) Grille_Carrée de 12 nœuds (b) Résultat de Spanning Tree

FIGURE 5 – Construction de SPT avec l'algorithme SPT_v1

14 SPT_v2

Par construction, les voisins *ST* d'un nœud sont ceux qui répondent *Yes* et, excepté l'initiateur, également celui qui a posé la première fois la question. Ainsi, pour cette détermination, les messages No ne sont pas nécessaires.

Les messages *No* sont plutôt utilisés pour la terminaison.

Complexité

Sur chaque lien $(x, y) \in E$, il y aura exactement une paire de message : soit Q dans un sens et *Yes* dans l'autre, ou deux messages Q, un dans chaque direction, d'où :

$$\text{M[SPT\_v2]} = 2m$$

Si nous ajoutons des informations supplémentaires sur les messages envoyés, tel que chaque entité doit éviter d'envoyer à ses voisins le message qu'elle a reçu ou qu'ils recevront, la complexité devient $\theta(2m)$.

Simulation

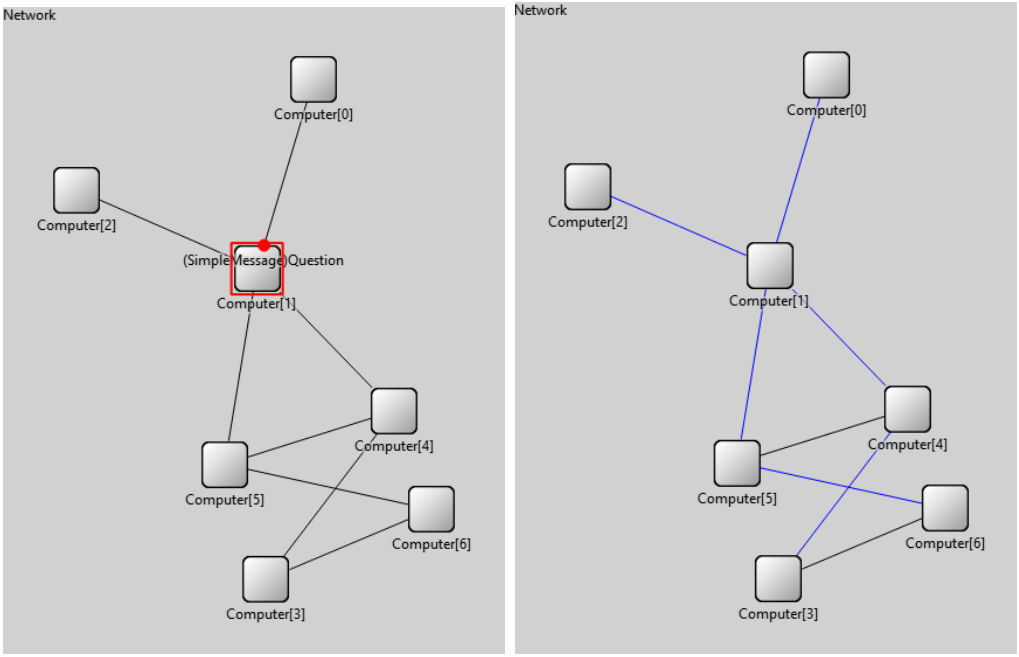
La Table 9 résume les simulations de protocole *SPT_v2*.

Topologie	n	m	Nombres des Messages
Graphe_Complet_4	4	$\frac{n(n-1)}{2}$	6
Graphe_Complet_8	8		14
Graphe_Complet_7	7		12
$\text{Grille\_Carrée}_{c=5,r=4}$	20	31	59
$\text{Grille\_Carrée}_{c=8,r=5}$	40	67	127
Anneau_4	4	4	8
Anneau_{30}	30	30	60

TABLE 9 – Table de Simulation SPT_v2

Simulation OMNET++

La Figure 6 montre l'arbre construit par le protocole *SPT_v2* sur une topologie quelconque.



(a) Topologie quelconque avec 7 nœuds (b) Résultat de SPT

FIGURE 6 – Construction de SPT avec l'algorithme SPT_v2

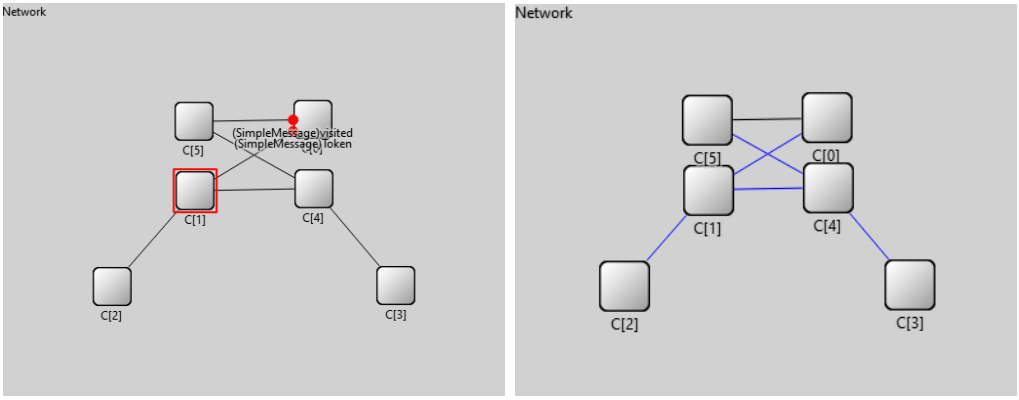
15 SPT_DFT_v3

Il est bien connu qu'une traversée en profondeur d'un graphe G construit en fait un arbre couvrant (df-tree) de ce graphe. Le df-tree est obtenu en supprimant les arêtes arrière de G (c'est-à-dire les arêtes pour lesquelles un message *Back-edge* a été envoyé dans DFT_v0).

En d'autres termes, les voisins d'arbre d'une entité x seront ceux dont elle reçoit un message *Return* et, si x n'est pas l'initiateur, celui dont x a reçu le premier T. De simples modifications au protocole DFT_v3 garantiront que chaque entité calculera correctement ses voisins dans l'arbre df. ses voisins dans le df-tree et se termine localement en un temps fini.

Simulation OMNET++

Quelques changements sur le protocole DFT_v3 amène nous sur le protocole SPT_DFT. La Figure 7 montre l'exécution de ce protocole.



(a) Topologie quelconque avec 7 nœuds (b) Résultat de SPT

FIGURE 7 – Construction de SPT avec l'algorithme DFT_v3

Cinquième partie

Le routage est le mécanisme par lequel des chemins sont sélectionnés dans un réseau pour acheminer les données d'un expéditeur jusqu'à un ou plusieurs destinataires. Le routage est une tâche exécutée dans de nombreux réseaux, tels que le réseau téléphonique, les réseaux de données électroniques comme Internet, et les réseaux de transports. Sa performance est importante dans les réseaux décentralisés.

Restrictions

— l'ensemble des restrictions standard **R** = {BL, CN, TR}
— L'initiateur unique est celui qui détient l'information initiale **UI**+

16 RR_v0

Dans cette section, nous nous concentrons sur le routage. Le but de ce dernier est de livrer des paquets à leurs destinations par les meilleurs chemins (généralement, les plus courts ou les moins chers). Pour des raisons de clarté, nous commençons par la forme la plus simple de routage, à savoir le routage aléatoire, où un nœud de non-destination qui reçoit un paquet l'envoie simplement à l'un de ses voisins, choisi au hasard.

Topologie

La topologie utilisée pour les simulations des protocoles de routage est l'arbre minimal de *Steiner* (voir la Figure 8).

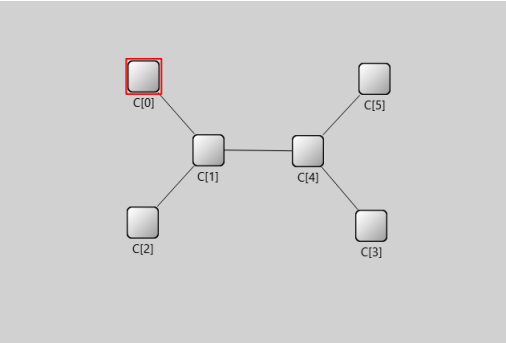


FIGURE 8 – Arbre minimal de Steiner

Simulation

Le hopCount est le nombre de sauts qu'un message doit parcourir avant d'atteindre sa destination. Chaque nœud capture le hopCount des messages traversée par lui, voici le diagramme sur la Figure 9 montre les statistiques des messages :

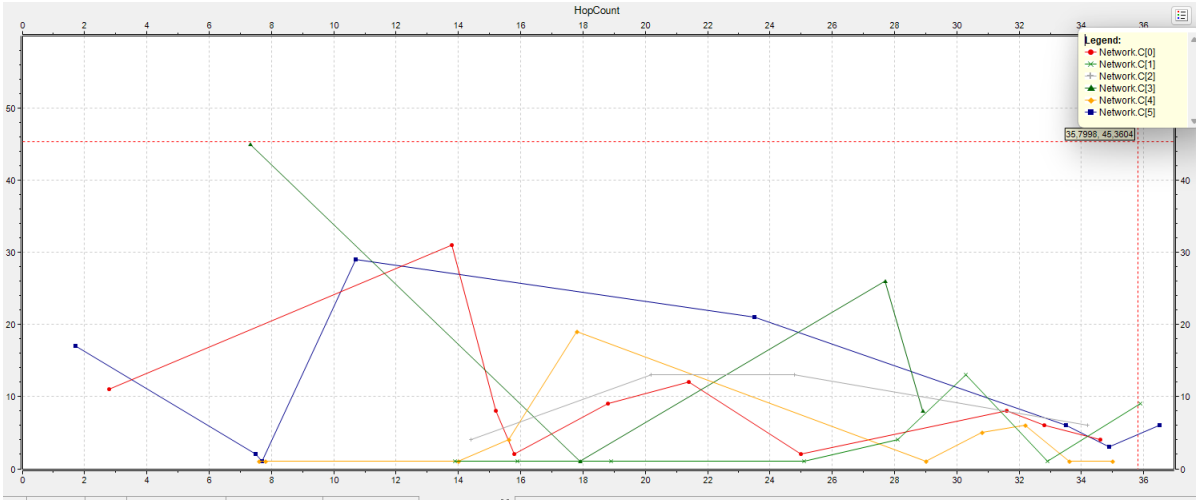


FIGURE 9 – Statistiques de hopCount des messages avec RR_v0

La Figure 10 illustre le Diagramme d'espace-temps des messages.

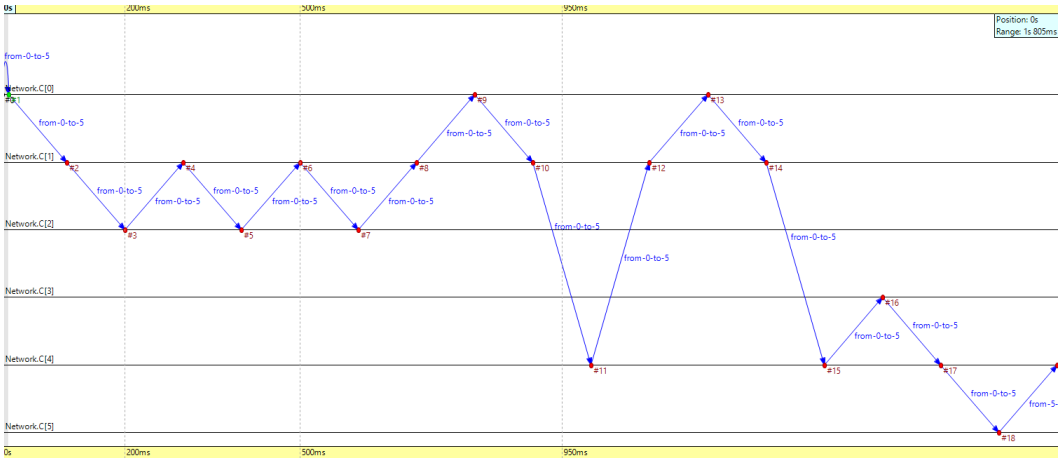


FIGURE 10 – Diagramme espace-temps RR_v0

Statistiques

Le Tableau 10 illustre la simulation et les différentes statistiques des messages envoyées dans le réseau.

Hop_Moy : C'est le nombre moyen de sauts qu'un message doit parcourir avant d'atteindre sa destination.

Hop_Min : C'est le nombre minimum de sauts qu'un message doit parcourir avant d'atteindre sa destination.

Hop_Max : C'est le nombre maximum de sauts qu'un message doit parcourir avant d'atteindre sa destination.

Hop_Ecart : C'est l'écart type de hopCount.

M_Recived : C'est le nombre de messages recevait par le nœud.

Nœud	$M_{Recived}$	Hop_Min	Hop_Max	Hop_Moy	Hop_Ecart
0	12	2	31	9.91667	7.899083
1	13	1	13	3.15385	3.73823
2	4	4	13	9	4.69042
3	6	1	45	18	16.8167
4	10	1	19	4	5.61743
5	11	1	29	9.36364	8.94732

TABLE 10 – Table de Statistique RR_v0

17 RR_v1

Il est clair que *RR-v0* n'est pas très efficace : souvent, le paquet rebondit entre deux nœuds pendant un certain temps avant d'être envoyé dans une autre direction. Cela peut être amélioré quelque peu si les nœuds ne renvoient pas le paquet à l'expéditeur.

Simulation

La Figure 11 montre les statistiques de hopCount des messages lors l'exécution de protocole *RR_v1*.

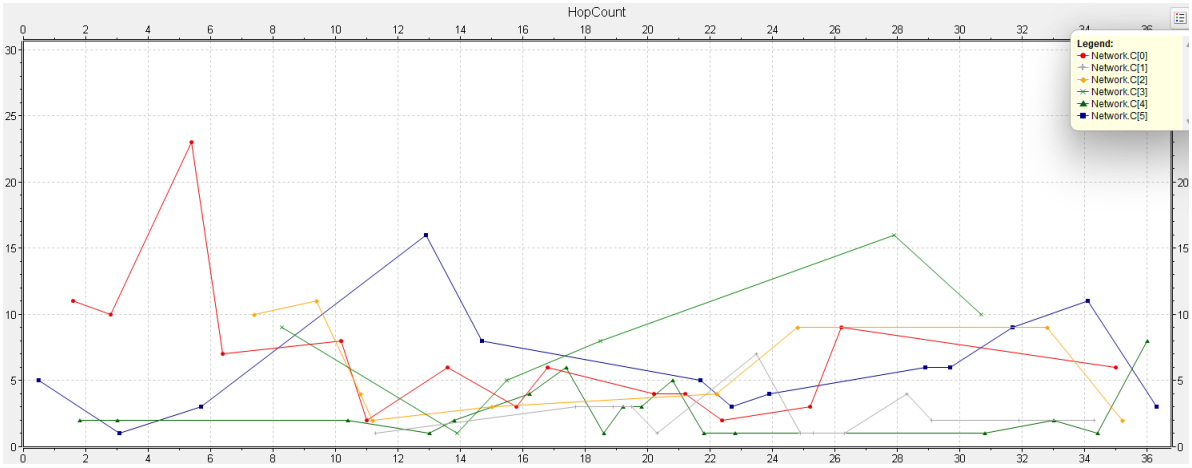


FIGURE 11 – Statistiques de hopCount des messages avec RR_v1

Statistiques

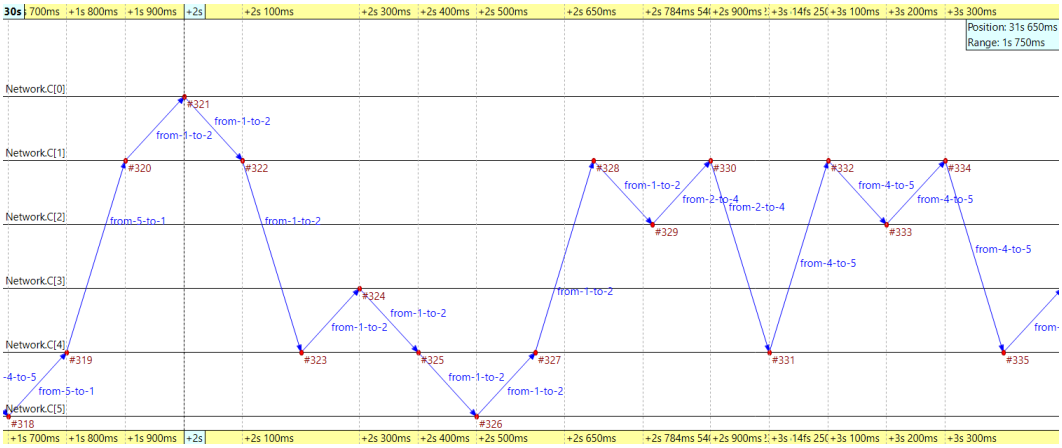


FIGURE 12 – Diagramme espace-temps RR_v1

La Table 11 et la Figure 12 résument les simulations de protocole *RR_v1*.

Nœud	$M_{Recived}$	Hop_Min	Hop_Max	Hop_Moy	Hop_Ecart
0	15	2	23	6.93333	5.27076
1	13	1	7	2.38462	1.7097
2	9	2	11	6	3.67423
3	6	1	16	8.16667	5.03653
4	17	1	8	2.64706	2.0292
5	13	1	16	6.15385	4.03828

TABLE 11 – Table de Statistique RR_v1

Remarque

La remarque notifiée est que le protocole *RR_v1* est plus efficace en termes de nombre de messages envoyés dans le réseau, comme exemple le premier message envoyé par le nœud 0 à 5 il prend 17 unités de temps avec le protocole *RR_v0* par contre avec le protocole *RR_v1* il prend que cinq unités de temps.

18 RT_v0

Pour améliorer encore *RR_v1*, nous allons exploiter le concept des tables de routage. Dans un premier temps, nous allons construire une table de routage très simple, où chaque nœud stocke ses voisins directs (à un saut).

Simulation

Dans une durée de 36 unités de temps, nous voyons bien que le nombre de messages envoyés dans le cas de *RT_v0* est plus grand que dans le cas de *RR_v1*, d'où l'efficacité de *RT_v0* par rapport au *RR_v1* (voir la Figure 13).

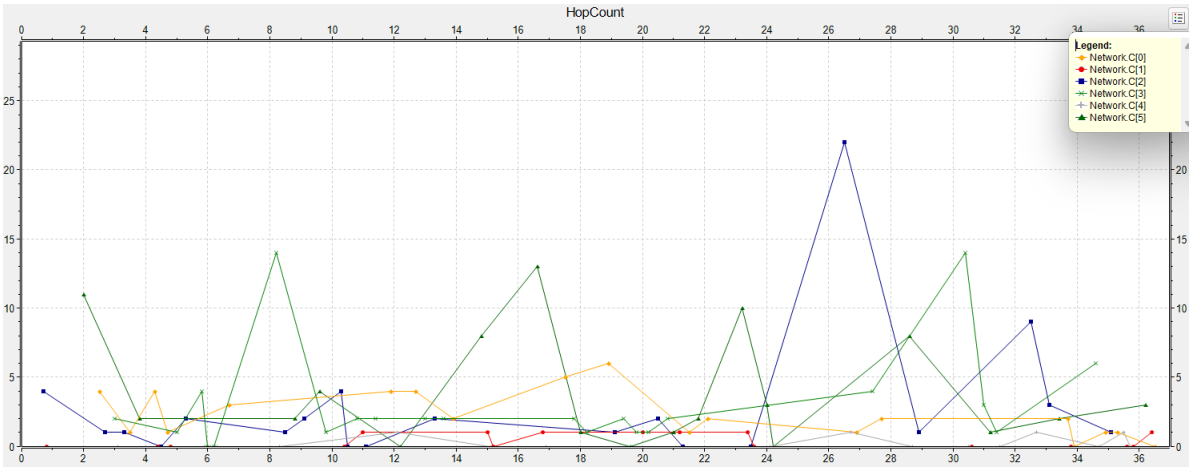


FIGURE 13 – Statistiques de hopCount des messages avec RT_v0

Statistiques

Le diagramme espace de temps dans la Figure 14, montre l'acheminement des messages entre les nœuds, nous voyons que le *hopCount* est de moyenne égale a 2 qu'est plus efficace dans le cas de *RR_v1*, la Table 12 résume les différents calcules on applique le protocole *RT_v0*.

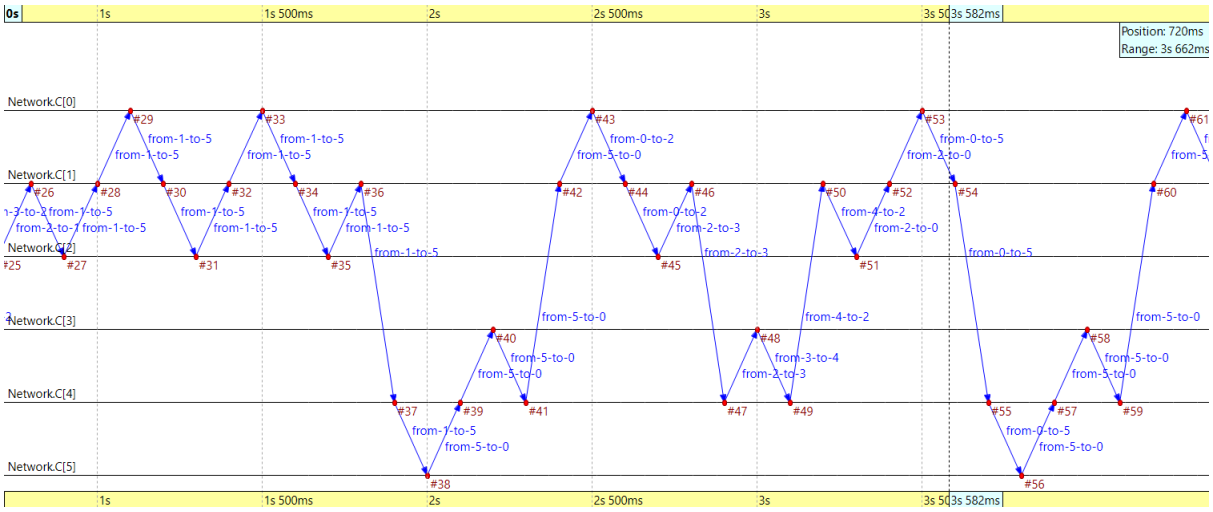


FIGURE 14 – Diagramme espace-temps RT_v0

Nœud	M_{Rec}	Hop_Min	Hop_Max	Hop_Moy	Hop_Ecart
0	19	1	6	2.31579	1.73374
1	17	1	1	0.411765	0.5073
2	20	1	22	2.8	4.97996
3	22	1	14	3.04545	3.79821
4	18	1	1	0.222222	0.427793
5	18	1	13	3.94444	4.13695

TABLE 12 – Table de Statistique RT_v0

Note :
M_Rec : le nombre de messages reçus.

19 RT_v1

Pour améliorer encore RT_v0 nous allons construire une table de routage, où chaque nœud stocke ses voisins directs et les voisins indirects (à deux sauts).

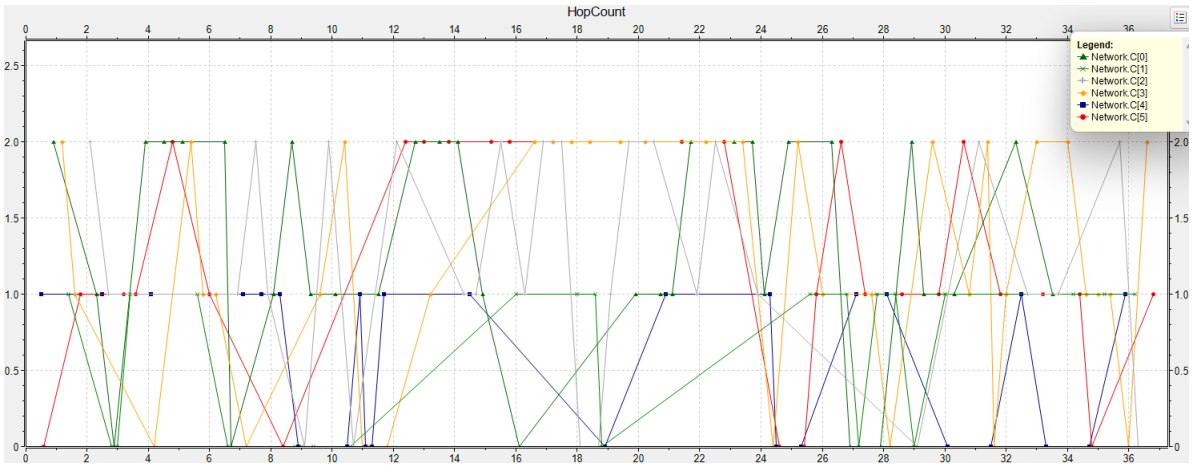


FIGURE 15 – Statistiques de hopCount des messages avec RT_v1

Simulation

Statistiques

Comme le diamètre de la topologie utilisé est égale à 4, le protocole *RT_v1* est plus efficace que les protocoles vu précédemment, nous voyons que le *hopCount* est de maximum égale a 2 (= d/2), comme la montre Figure 15, donc en termes de routage des messages le protocole *RT_v1* est efficace. La Table 13 et la Figure 16 résument les résultats obtenus l'or de la simulation de protocole *RT_v1*.

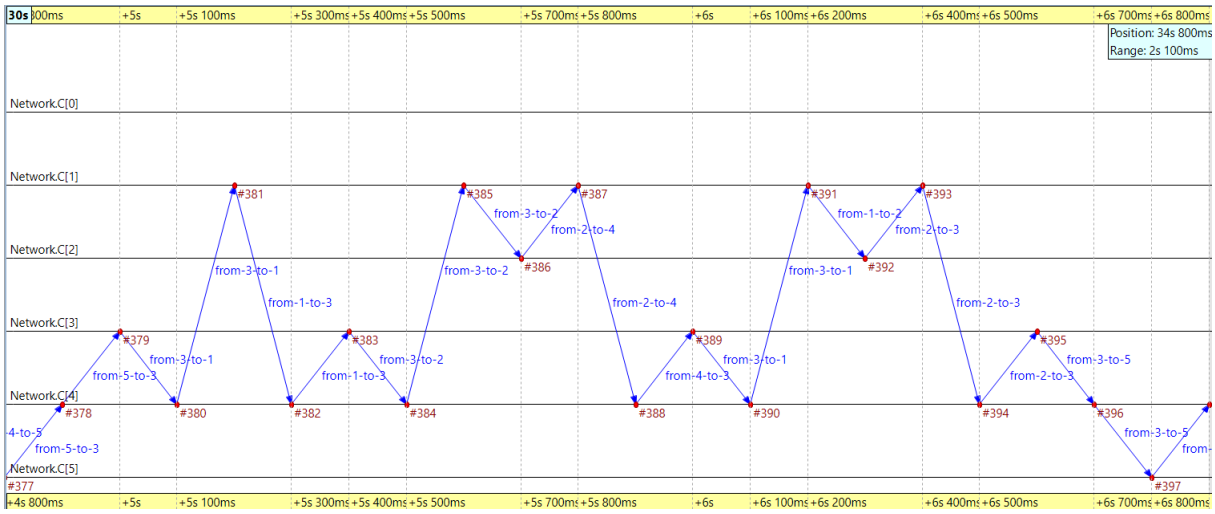


FIGURE 16 – Diagramme espace-temps RT_v1

Nœud	M_{Rec}	Hop_Min	Hop_Max	Hop_Moy	Hop_Ecart
0	34	1	2	1.32353	0.726994
1	24	1	1	0.583333	0.50361
2	29	1	2	1.2069	0.773642
3	38	1	2	1.21053	0.810669
4	26	1	1	0.576923	0.503831
5	27	1	2	1.18519	0.735738

TABLE 13 – Table de Statistique RT_v1

Sixième partie

20 Plus de Protocoles

20.1 Center Finding

La stratégie pour construire un arbre de diffusion de diamètre *d(BT)* est alors simple à énoncer :

— Déterminer un centre c de G

— Construit *BFT(c,G)* avec une racine en c.

Ces problèmes, comme nous le verrons, ne sont pas simples à résoudre efficacement, pour cela nous avons réduit le problème de recherche un centre dans un graphe à la recherche d'un centre dans un arbre.

Un centre est un nœud à partir duquel la distance maximale à tous les autres nœuds est minimisée. Un réseau peut avoir plus d'un centre. Le problème de recherche de centre (Center Finding) consiste à à faire en sorte que chaque entité sache si elle est ou non un centre. Pour résoudre la question du centre, nous pouvons utiliser le fait qu'un centre est exactement un nœud avec la plus petite excentricité. Ainsi, un protocole de solution consiste à trouver le minimum parmi toutes les excentricités, en combinant les protocoles :

— Excentricités

— Saturation et Résolution et Broadcast

L'excentricité d'un nœud x, notée *r(x)*, est la plus grande distance entre x et tout autre nœud de l'arbre : $r(x) = Max\{d(x, y) : y \in \text{V }\}$.

Pour calculer sa propre excentricité, un nœud x doit déterminer la distance maximale entre lui et tous les autres nœuds de l'arbre. La première étape consiste à utiliser la saturation pour calculer l'excentricité des deux nœuds saturés (La Saturation sera étudié prochainement). Ce qui est intéressant, c'est que l'information disponible à chaque entité à la fin de la l'étape de saturation est presque suffisante pour leur faire calculer leur propre excentricité. les nœuds saturés peuvent fournir les informations nécessaires à leurs voisins, qui peuvent alors calculer leur excentricité.

Ainsi, le coût du message des protocoles Excentricités et Recherche de centre seront :

$$M[Excentricits] = 3n + k - 4 \leq 4n - 4$$

$$M[CenterFinding] = 5n + k - 6$$

où k désigne le nombre d'initiateurs.

Simulation

La Figure 17 er la Table 14 montre l'exécution de l'algorithme de recherche du centre.

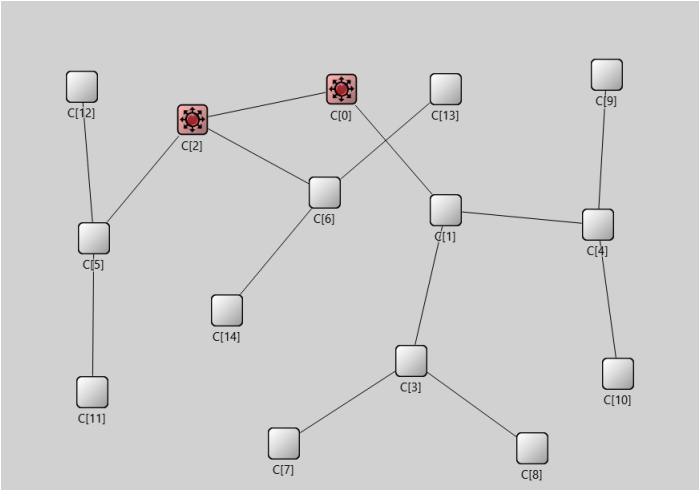


FIGURE 17 – Topologie de l'arbre

Nœud	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Excentricité	3	4	4	5	5	5	5	6	6	6	6	6	6	6	6

TABLE 14 – Table de Statistique Center Finding

Il est clair que le centre de l'arbre est le nœud 0. Dans la fin d'exécution de l'algorithme chaque nœud sache si elle est ou non le centre.

20.2 Full Saturation

La technique, que nous appellerons la saturation complète, est très simple et peut être lancée de manière autonome et indépendante par un nombre quelconque d'initiateurs. Il est composé de trois étapes :

1. la phase d'activation, lancée par les initiateurs, au cours de laquelle tous les nœuds sont activés.
2. la phase de saturation, entamée par les nœuds feuilles, au cours de laquelle un couple unique de nœuds voisins est sélectionné
3. la phase de résolution, entamée par le couple sélectionné

n'importe quelle paire de voisins pourrait être saturée. La phase de résolution entamée par le couple sélectionné est pour but de résoudre quelques problèmes comme la recherche du centre d'un arbre. Pour déterminer le nombre d'échanges de messages, on observe que l'étape

d'activation est un *Wake_ Up* dans un arbre et qu'elle utilisera donc n + k - 2 messages, où k désigne le nombre d'initiateurs. Pendant la phase de saturation, exactement un message est transmis sur chaque lien, sauf sur le lien reliant les deux nœuds saturés, Par conséquent la complexité de ce protocole est :

$$M[FullSaturation] = 2n + k - 2$$

20.3 Processus Noirs

Un réseau bidirectionnel asynchrone composé de n processus (avec ID) a un diamètre D et peut contenir zéro ou plusieurs processus noirs. Supposons que tous les processus se réveillent à l'instant 0 et commence l'exécution du code fournie à eux. Aussi, chaque processus sait initialement s'il est noir ou non, et connaît les identités de ses voisins, Néanmoins, les processus ne connaissent pas le nombre total de processus n ni le diamètre D.

Algorithme

Pour résoudre ce problème un algorithme basé sur la diffusion est nécessaire. Nous définissons 2 états : S_{init} = {IDLE} et S_{finale} = {DONE}

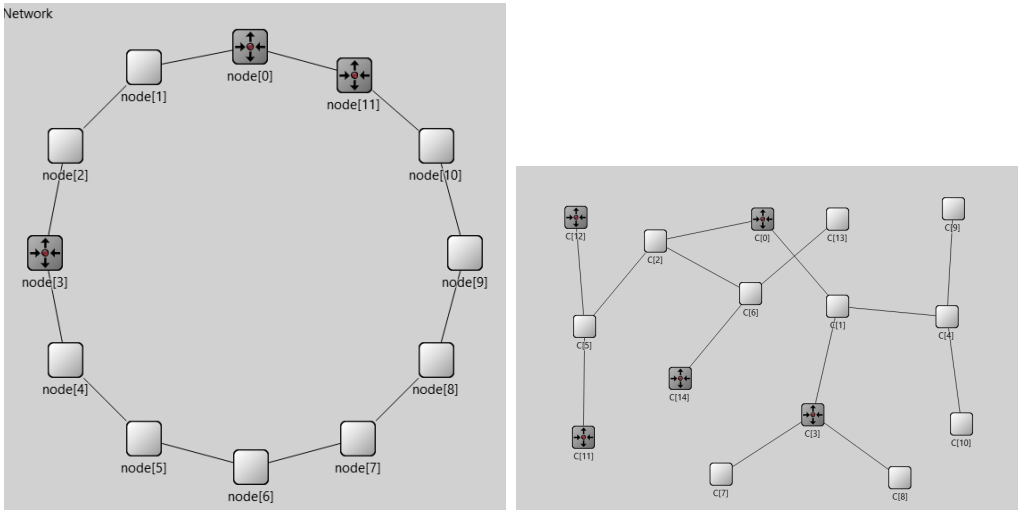
IDLE × S_p → {
Unknown ← $N(x)$
if (Black) {
Black_nodes ← x
}
else {
White_nodes ← x
}
diffuse() }

IDLE × $Receving_{(B,W,U)}$ → {
Black_nodes ← Black_nodes ∪ B
White_nodes ← White_nodes ∪ W
Unknown ← Unknown ∪ U - Black_nodes - White_nodes

if (Unknown == ∅){
Nmb ← | Blcak_nodes |
Become DONE
}
diffuse()
}
Procédure diffuse () { **send**(Black_nodes, White_nodes, Unknown) to $N(x)$}

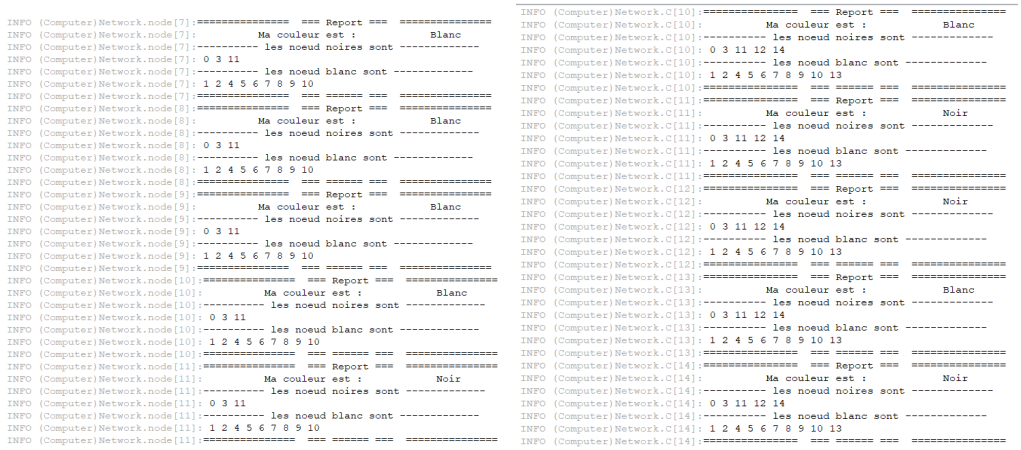
Simulation

Nous avons fait une simulation sur Omnet++, quelques résultats sont montrés dans la Figure 18 et 19.



(a) Topologie ring avec 12 nœuds (b) Topologie arbre avec 15 nœuds

FIGURE 18 – Topologies de simulation d'algorithme Processus Noirs



(a) Topologie ring avec 12 nœuds (b) Topologie arbre avec 15 nœuds

FIGURE 19 – Résultats de simulation d'algorithme Processus Noirs

Disccusion

L'algorithme proposé de Processus Noirs résoudre avec succès le problème.

21 Conclusion

Tous les protocoles cités dans ce rapport seront simulés avec l'outil Omnet++. Les codes de simulations sont commentés pour mieux comprendre.