Modélisation formelle d'algorithmes pour les systèmes répartis

Par

Antoine Toullalan, Rosa Mendas

Encadrant

Fabrice Kordon

Rapport intermédiaire soumis dans le cadre de l'UE LU3IN013
Sorbonne Université
15 Mars 2021

Table des matières

Table des figures

Chapitre 1

Introduction

En ce 9 mars 2021, le leader européen du Cloud OVH est victime d'un incendie. Des données des data-centers ont été perdues et plus de 3 millions de serveurs HTTP se sont retrouvés hors ligne [1]. Cet incident nous prouve que la conception actuelle des zones de stockage atteint ses limites et que des progrès sont à faire.
L'évolution d'Internet et la possession d'ordinateurs personnels permettent un accès facile aux réseaux favorisant l'augmentation du nombre d'utilisateurs. Cette évolution technologique et celle des besoins ont donné naissance aux applications distribuées : sur deux ordinateurs différents ces applications coopèrent pour effectuer des tâches coordonnées entre un ou plusieurs client(s) et serveur(s).
Ces applications (magasins virtuels par exemple) utilisent les services fournis par un système réparti qui est un ensemble d'ordinateurs indépendants reliés par un réseau de communication et apparaissant comme un système unique et cohérent [2](WWW ou DNS par exemple) [2].
Ainsi, des utilisateurs du monde entier stockent et partagent des documents qui se retrouvent dispersés sur plusieurs sites distincts. Il est alors nécessaire de disposer d'un système de stockage permettant un accès rapide et sécurisé aux données.
On s'intéresse au réseau pair-à-pair qui est une infrastructure où tous les ordinateurs -appelés nœuds- sont directement connectés les uns aux autres : on ne passe pas par un serveur central lors de la transmission d'informations, les machines sont à la fois client et serveur.
Afin de faciliter la coopération entre les nœuds, on organise l'espace de sorte à localiser les nœuds simplement : par exemple attribuer la gestion des ressources à un groupe de noeuds précis en utilisant une Table de Hachage Distribuée (DHT pour Distributed Hash Table).

1. Chiffres donnés par Netcraft
2. Définition d'un système réparti d'après Tanenbaum

1.1 Objectifs

Le but de cette recherche est de construire des modèles d'algorithmes régissant le comportement des DHT en utilisant la notation des réseaux de Petri (qu'on notera aussi RdP).
Autrement dit, nous tenterons -grâce au langage de modélisation que sont les réseaux de Petri- de donner une visualisation globale de la structure et de la dynamique des DHT, plus précisement d'algorithmes décrivant leur fonctionnement.
Il sera important de comprendre et assimiler les notions des DHT et RdP dans un premier temps et comprendre pour quelles raisons nous faisons le choix de ces structures.
D'autre part, il faudra produire un modèle en PNML qui alimentera le «model cheking contest» :concours international de référence pour les logiciels de vérification de systèmes asynchrones par model checking.
Ainsi, dans cette partie du rapport, nous définissons les réseaux de Petri et DHT et abordons leurs principes de fonctionnement. Nous notons qu'il faudra par la suite modéliser des algorithmes de fonctionnement des DHT puis vérifier des propriétés basiques sur les modèles pour nous assurer de leur pertinence.

Chapitre 2

Réseaux de Petri

Apparus en 1962 dans la thèse de doctorat de Carl Adam Petri, les réseaux de Petri (RDP) sont un langage de modélisation graphique permettant de décrire des relations entre des conditions et évènements [14].

Pourquoi utiliser la notation de Petri dans la modélisation d'algorithmes de DHT ?

— Visualisation graphique de la conception : on décompose en éléments simples les éléments constitutifs du système.

— Description précise de la structure du système et de sa dynamique sur un même support.

On s'intéresse aux réseaux de Petri de places et de transitions.

2.1 Les composants élementaires

Un réseau de Petri est un graphe biparti puisqu'il est constitué de deux composants graphiques élémentaires : les places et les transitions.

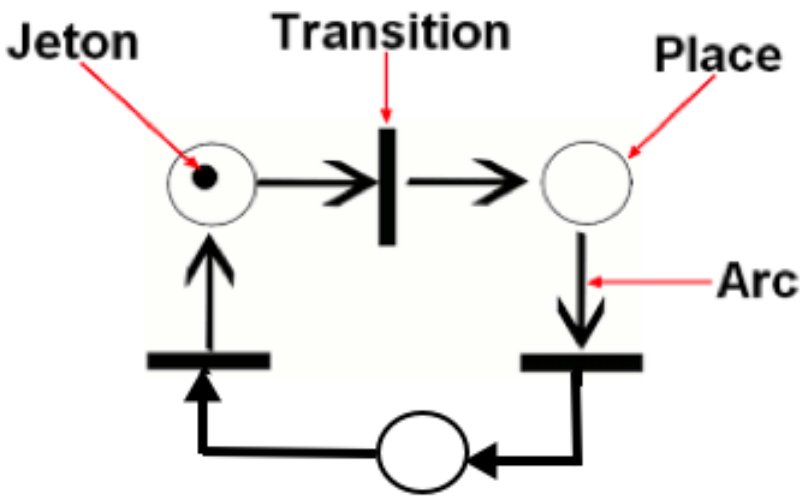

FIGURE 2.1 – Représentation d'un réseau de Petri [7]

Ce RdP est assez simple ici mais il n'y a pas de limite au nombre de places ou d'arcs d'un RdP. Une place est symbolisée par un cercle et représente une condition qui traduit l'état d'une ressource du système (par exemple une place peut traduire l'état "la machine est à l'arrêt" et une autre "la machine est en marche").
Une transition est symbolisée par un rectangle représentant un événement (une action se déroulant dans le système, par ex : "la machine tombe en panne") dont l'occurrence provoque la modification de l'état du système. Les arcs assurent la liaison d'une place vers une transition et d'une transition vers une place (par ex : de la place "la machine est en marche" vers la transition "la machine tombe en panne" vers la place "la machine est à l'arrêt"). Il ne peut donc pas y avoir d'arcs entre deux places ou deux transitions. Les jetons permettent de savoir si une place symbolise un état vrai ou non, une place avec un jeton symbolise donc un état vrai dans le système (par ex : si la place "la machine est en marche" contient un jeton, la machine est bien en marche dans notre système et si notre système est correcte, la place "la machine est à l'arrêt" ne contient pas de jetons). [4]

2.2 Dynamique d'un RDP

Un réseau de Petri représente un système dynamique, il doit donc évoluer de la même manière que le système qu'il modélise. En effet, la position des jetons change en fonction de l'évolution du système.

Pour qu'une place gagne un jeton, il faut qu'une des transitions sources de cette place soit validée, c'est à dire que toutes ses places d'entrée possèdent un jeton (nous simplifions ici, certains réseaux nécessitent plus d'un jeton à une place pour valider une transition). Lorsque la transition est validée, on enlève un jeton (ou plus) de chaque place d'entrée et on place un jeton (ou plus) dans chaque place de sortie.

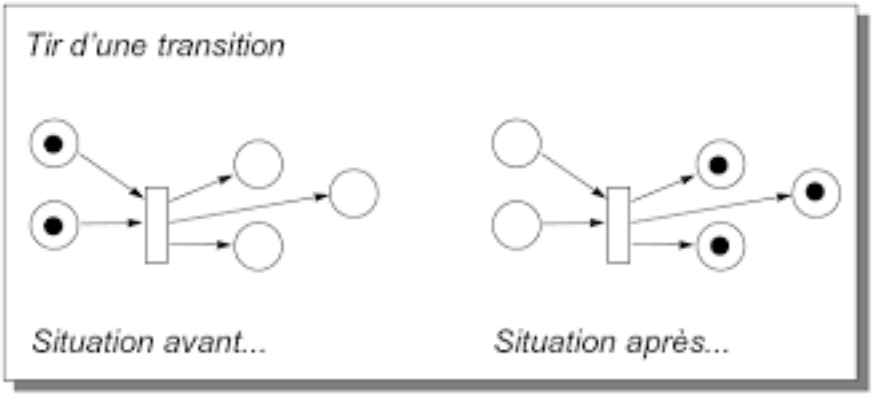

FIGURE 2.2 – Représentation de la dynamique d'un réseau de Petri [6]

On voit bien au sein de la Figure 2.2 que les deux places entrantes ont un jeton et qu'ensuite ce sont les trois places sortantes qui en ont chacune un. On a ainsi un réseau qui modélise bien un système dynamique. Ainsi grâce à un petit nombre de composants élémentaires et une dynamique d'éxécution assez simple on peut représenter des systèmes très complexes [9].

Un réseau de Petri se caractérise aussi par son graphe d'accessibilité qui représente les différents états à travers lesquels le RdP peut passer. À partir d'un état donné du graphe d'accessibilité, il est possible de définir les chemins de navigation possibles à travers le RdP.

La construction de ce type de graphe est un bon moyen de voir s'il existe des états défectueux, par exemple si dans un RdP il existe un état où les places "la machine marche" et "la machine est à l'arrêt" possèdent en même temps un jeton. On prouve avec le graphe d'accessibilité -comme en Figure 2.3- que notre réseau est correct. Néammoins, dans les cas pratiques la construction du graphe d'accessibilité est beaucoup trop complexe car les RdP ont souvent plus de trois transitions et trois places (Figure 2.4 notamment), on a donc des graphes avec beaucoup trop d'états

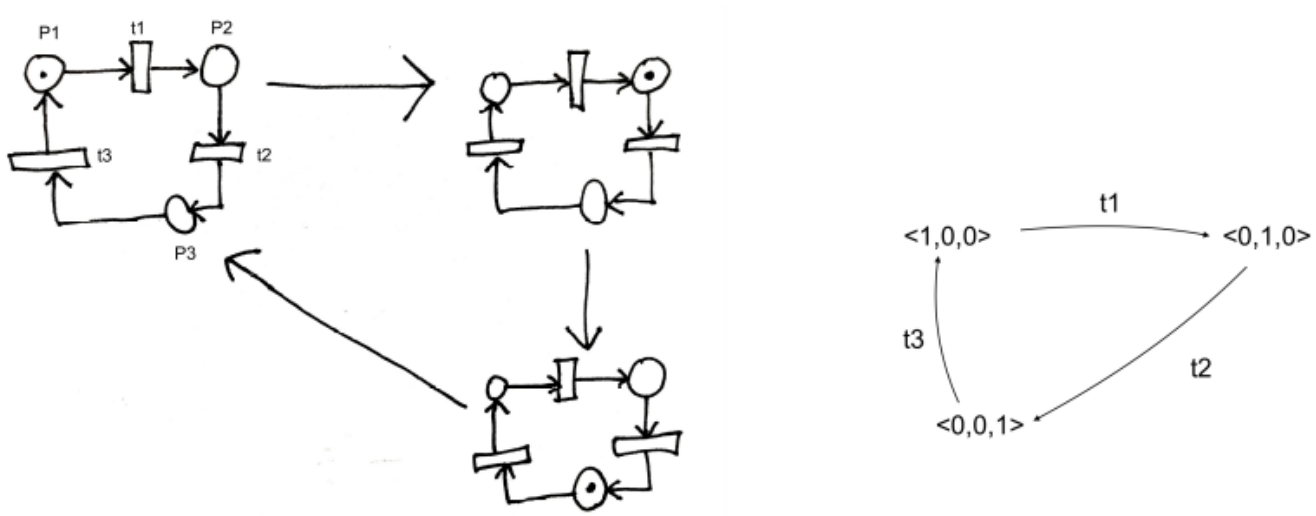

FIGURE 2.3 – Abstraction des représentations du système(graphe de gauche) et représentation du graphe d'accessibilité(graphe de droite) de la Figure 2.1

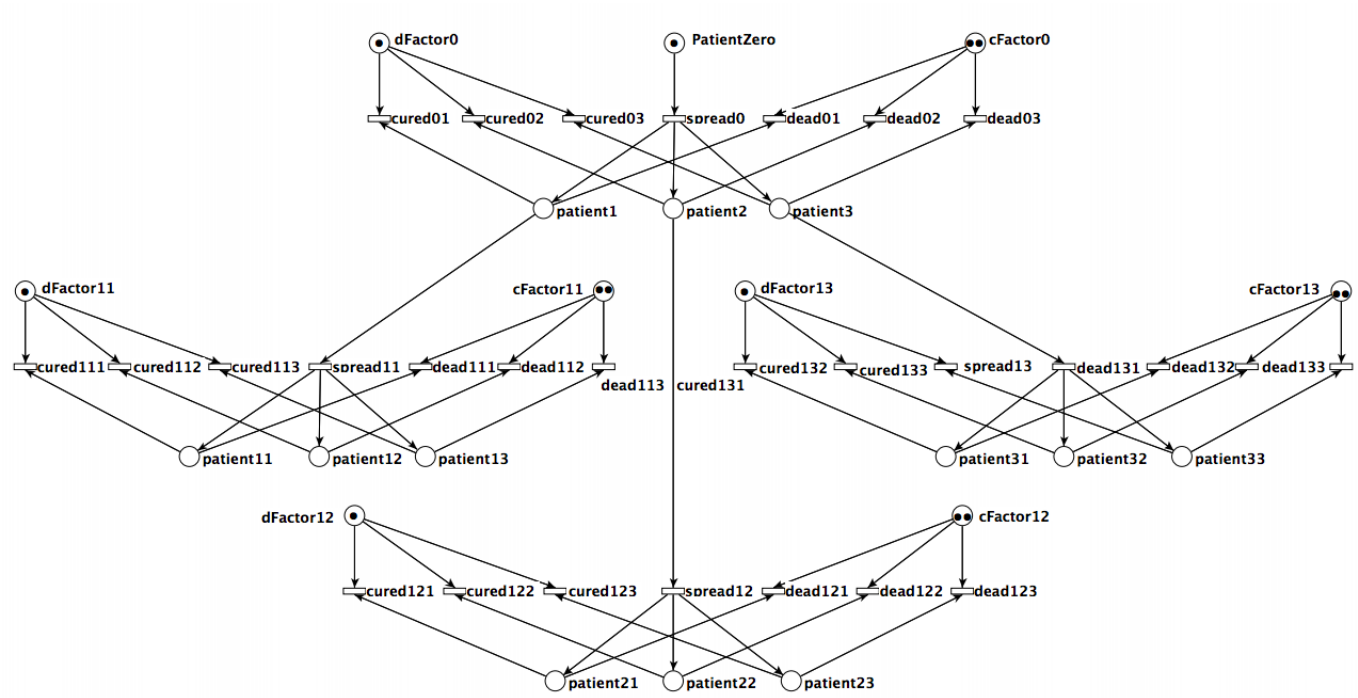

FIGURE 2.4 – Exemple d'un réseau modélisant une epidémie (à tout hasard) tiré du model checking contest de 2020 [8]

à représenter. Pour remédier à ce problème, il existe des solutions avec la logique temporelle linéaire que nous ne détaillerons pas ici. On utilisera donc les réseaux de Petri pour modéliser des tables de hachages distribuées.

Chapitre 3

Tables de hachage distribuées(DHT)

3.1 Description des DHT

Nous avons vu que les objets à modéliser sont des DHT. Pour commencer, une table de hachage est une structure de données, plus précisement un tableau ne comportant pas d'ordre(non indexé).Chaque élement du tableau est accessible par sa clé (key) comme nous pouvons l'observer en Figure 3.1. L'index du tableau -où la clé est mémorisée- est calculé à partir d'une fonction de hachage choisie préalablement pour éviter les collisions. En effet, plusieurs couples (clé,valeur) peuvent produire un même index suite à leur transformation. On retrouve alors plusieurs clés pour un seul index : sur la Figure 3.2 sont stockés deux couples (clé,valeur) en un même indice.
Un DHT est la mise en place d'une table de hachage dans un réseau distribué. Le réseau est constitué d'un ensemble de noeuds (qui peuvent être des ordinateurs par exemple) et le DHT permet de répartir le stockage des données du réseau : ces données peuvent être des films, des photos...
Chaque noeud est muni d'un identifiant clé nommé le NodeID, et à chaque donnée stockée sur ce noeud (s'il y en a) est associé un ObjID. On retrouve ainsi la paire (clé,valeur) précédemment énoncée [12].

Mais pourquoi les DHT ?
La force des DHT réside dans le fait que la perte d'un noeud ou d'un ensemble de noeuds ne signifie que très rarement la perte de données. En effet, chaque objet n'est pas stocké sur un unique noeud, mais est copié sur un ensemble de noeuds [10]. Les informations contenues dans un noeud sont copiées ailleurs sur le DHT pour de multiples raisons que nous aborderons (attaques sur le réseau, perte d'un noeud...)
De plus, le choix de la structure est lié à la rapidité d'accès, de recherche et d'inser-

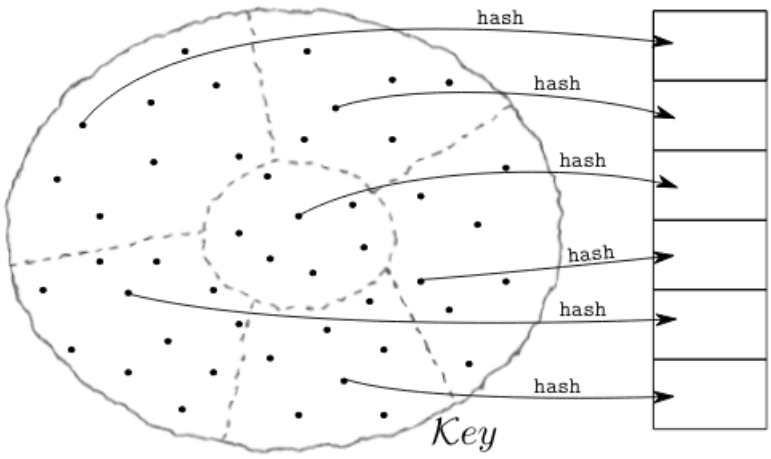

FIGURE 3.1 – Représentation de la construction d'une table de hachage

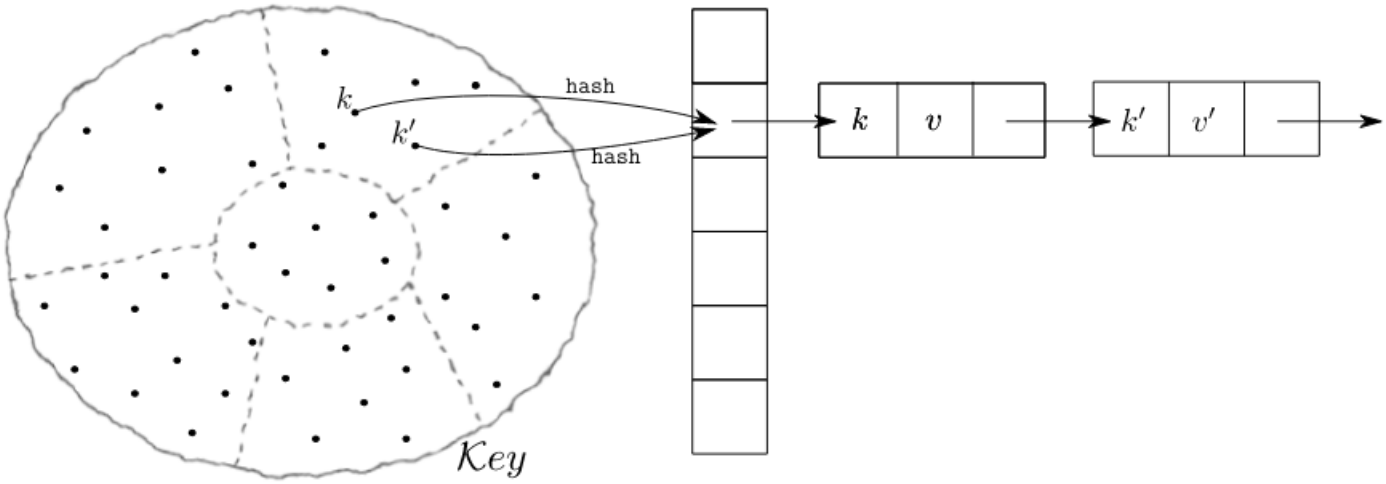

FIGURE 3.2 – Représentation de la construction d'une table de hachage avec collision [5]

tion/retrait des données.

Il existe différents types de DHT mais les principes de fonctionnement sont les mêmes. Nous retrouvons toujours 3 algorithmes importants qui sont l'insertion, la recherche, et la suppression d'un noeud. De plus, chaque DHT possède un dispositif de routage permettant aux noeuds de communiquer entre eux. Cette communication permet notamment de détecter toujours de la même façon la perte d'un noeud.

— Insertion : put(DHT,node)
— Recherche : get(DHT,node)

— Suppression : delete(DHT,node) [1]

Caractéristiques d'un noeud

En général, chaque noeud contient 3 structures de données qui constituent son **état** : Un **LeafSet**, un ensemble de **voisins** et une **table de routage**.
Le LeafSet d'un noeud c comprend d'autres noeuds où sont répliquées les données à stocker sur c. La table de routage sert lors du routage des messages vers un noeud et les voisins sont les noeuds qui sont proches géographiquement (dans les DHT et pas forcément en réalité) du noeud.

Perte d'un noeud

Dans les conditions normales, un nœud envoie des messages de mise à jour périodiques à chacun de ses voisins, donnant ses caractéristiques et une liste de ses voisins (on appelle ce système le "heart beat").
L'absence de message de mise à jour signale son absence. Une fois qu'un nœud décide que son voisin est mort, il lance une procédure de prise de contrôle (procédure qui sera lancée par chacun des voisins du nœud mort).

Conciliation des copies

Un des principaux problèmes dans un DHT est la fiabilité des noeuds : un noeud qui s'ajoute au réseau pourrait très bien avoir de mauvaises intentions (par exemple diffuser la mauvaise version du film "La petite Sirène"). Ainsi lorsqu'un utilisateur souhaite accéder à un objet, il va s'adresser à un noeud qui contient l'objet grâce au routage dans le réseau.
Ce noeud ne va pas tout de suite transmettre l'objet à l'utilisateur mais il va demander aux noeuds de son LeafSet (qui contiennent aussi l'objet comme vu précédemment) quelle est leur version de l'objet. Dans l'exemple de "La petite Sirène", si cinq noeuds présentent la version originale du film et un noeud (qui est le noeud mal intentionné) la version erronée, on va prendre alors la version majoritaire qui est la version originale. On appelle cet algorithme "Quorum-based".

On a présenté ici le fonctionnement général des DHT, nous allons à présent porter notre intérêt sur Pastry et CAN. Ces deux DHT ont une organisation assez différente : tandis que Pastry va baser son fonctionnement sur les NodeID du réseau qui sont

1. Il existe des variantes entre chaque algorithme, nous donnons en paramètre le noeud à gérer ainsi que le DHT.

indépendants de leur localisation géographique, CAN va utiliser des coordonnées spatiales pour son fonctionnement (structure fractale).

3.2 Réseaux de Pastry

Le fonctionnement de ce réseau est basé sur les NodeID des noeuds qui sont indépendants de leur localisation géographique.

Revenons un instant sur le Leaf Set : on a vu qu'un objet est stocké sur un noeud X au départ car son NodeID est le plus proche de l'ObjID dans le réseau. Il s'agira du "noeud maitre" pour l'objet. En effet, lorsque l'on va vouloir accéder à l'élement, on s'adressera obligatoirement à ce noeud. On a vu aussi qu'un objet était répliqué pour plus de fiabilité. Dans le réseau de Pastry, les noeuds vers lesquels on réplique l'objet sont ceux avec le NodeID le plus proche de celui du "noeud maitre", appellés ici les "noeuds valets". Le LeafSet va donc conserver l'ensemble des "noeuds valets" du noeud X.
Chacune de ces structures contient donc l'adresse IP et le NodeID d'autres noeuds du réseaux (cela ne constitue qu'une petite partie de l'ensemble des noeuds du réseau) [3].
Considérons un noeud A de NodeID **10233102**, les Figures 3.3, 3.4, 3.5 schématisent respectivement le Leafset, la table de routage et les voisins de A.

Leaf set	SMALLER	LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

FIGURE 3.3 – Exemple de LeafSet pour un noeud A de NodeID 0233102

Routing table			
-0-2212102	**1**	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	**2**	10-3-23302
102-0-0230	102-1-1302	102-2-2302	**3**
1023-0-322	1023-1-000	1023-2-121	**3**
10233-0-01	**1**	10233-2-32	
0		102331-2-0	
		2	

FIGURE 3.4 – Version simplifiée de la table de routage du noeud A

Neighborhood set			
13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

FIGURE 3.5 – Noeuds voisins du noeud A

Routage dans le réseau

Mais comment un noeud qui ne contient qu'une petite partie des informations des noeuds du réseau, va envoyer et recevoir des messages ?

Un algorithme Greedy est utilisé par chaque Noeud qui reçoit une requête à destination d'un NodeID. La requête peut aussi être associée à l'ObjID d'une donnée, dans ce cas la destination est le NodeID le plus proche de l'ObjID dans le réseau. Cette donnée sera copiée par k noeuds de son Leaf Set [13].

Déroulement général de l'algorithme Greedy :
Un noeud A reçoit une requête destinée à un noeud X avec le NodeID I(ou à un ObjID comme vu précédemment). A regarde dans son état s'il possède un noeud avec le NodeID I. Si oui alors A envoie la requête à ce noeud, sinon A envoie la requête au noeud avec le NodeID le plus proche de I. Cet algorithme sera éxécuté pour tous les noeuds par lesquels passent la requête jusqu'à arriver au noeud X.
Ainsi lorsqu'on veut accéder à une donnée stockée dans le réseau, la table de hachage associée au réseau nous donne son ObjID et grâce à l'algorithme Greedy on contacte le noeud où est stocké la donnée pour qu'il nous la transmette.
De la même manière, pour stocker une donnée dans le réseau, un ObjID est associé à la donnée et on transmet une requête, grâce à l'algorithme Greedy, au noeud dont le NodeID est le plus proche de l'ObjID .

Ajout d'un noeud

Supposons que le noeud X veuille joindre un réseau de Pastry. L'état de X contient une table de routage, un leaf set et un ensemble de voisins vide.
1ere étape : Une clé de 128 bits est associée à X
2eme étape : X localise un noeud A qui va contacter d'autres noeuds.
3eme étape : Ces noeuds envoient leur état au noeud X qui va initialiser son propre état ainsi.
4ème étape : X transmet une copie de sa table de routage à tous les noeuds de son

état pour qu'ils sachent qu'il existe.

Ainsi les réseau de Pastry sont des DHT dont le fonctionnement repose sur les NodeID des noeuds du réseaux, et qui assure ainsi un stockage fiable et distribué des données.

3.3 Réseaux CAN

Description du réseau

CAN (content adressable network) est un réseau de recouvrement de type table de hachage basé sur des coordonnées spatiales : chaque nœud est identifié par un point P(de coordonnées x,y 2D) n'ayant pas de lien avec l'adresse physique du pair. L'espace est partagé en rectangles possédant chacun un noeud responsable, on parle de topologie hypercubique [11].

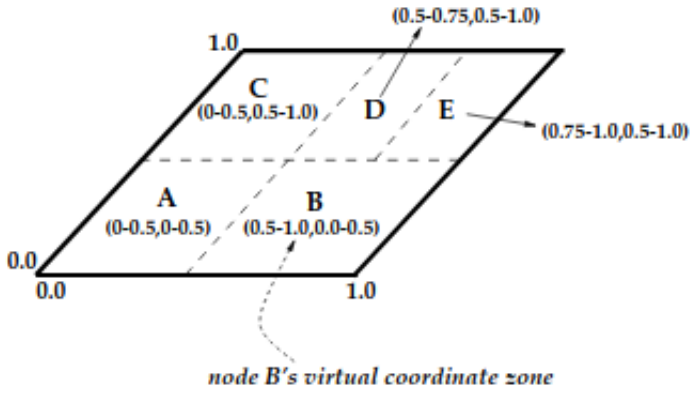

FIGURE 3.6 – Représentation spatiale d'un CAN composé de 5 nœuds

Chaque nœud possède sa propre zone de la table de hachage- qui est une portion de l'espace entourant le point P identifiant le nœud - et contient des informations sur ses zones adjacentes : permettant de router un message entre les nœuds.

Chaque CAN a un unique nom de domaine (DNS), permettant d'avoir accès à des adresses IP de nœuds au sein du réseau.
On peut également visualiser un CAN comme un arbre binaire : Les nœuds internes ont déjà possédé une zone qui a été partagée en deux avec les descendants. Les feuilles représentent donc les zones existantes à cet instant donné [1].

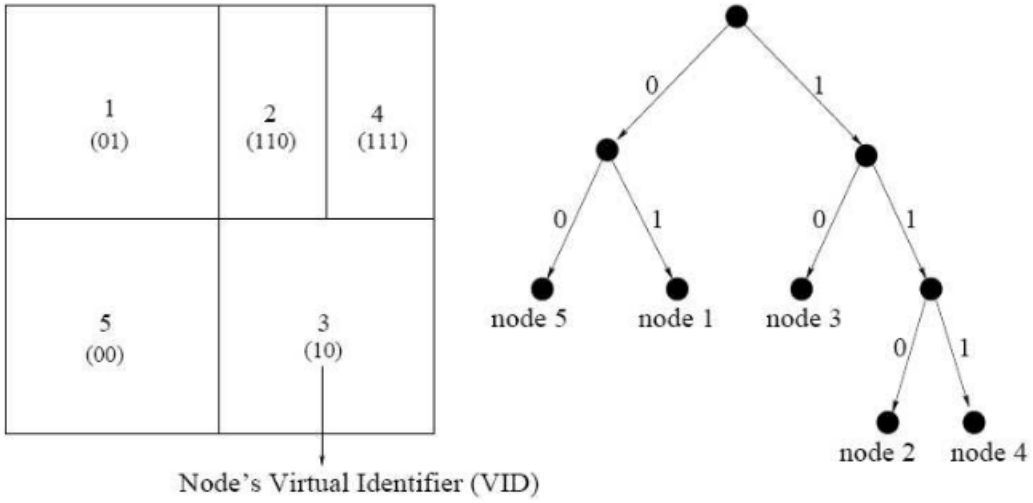

FIGURE 3.7 – Représentation d'un CAN sous forme d'arbre

Ajout d'un noeud

Dans un réseau CAN l'ajout d'un noeud consiste en 3 étapes :

— Amorçage(bootstrapping)

— Recherche de la zone

— Notifier les noeuds

Amorçage(bootstrapping)

La première étape consiste à trouver l'adresse IP d'un nœud existant au sein du réseau de recouvrement CAN grâce au protocole DNS : On parle de point d'entrée et de nœud bootstrap. Ce nœud ≪c≫ est ensuite utilisé pour déterminer une liste de nœuds actifs au sein du réseau.

Recherche de la zone

Le nœud à insérer choisit aléatoirement un point P dans l'espace qui sera son identifiant (NodeID).
Le nœud ≪c≫ tente de router un message d'adhésion (JOIN message) vers le point P. Le nœud responsable de la zone peut honorer la demande et scinder sa zone en deux pour attribuer une partie au nouveau nœud. Dans le cas où la demande est refusée, le nœud à insérer choisit de nouveau aléatoirement un nouveau point jusqu'à intégrer le réseau.

Algorithme de routage greedy sur x et y

Parmi les algorithmes de routage existants voici un algorithme glouton (greedy)

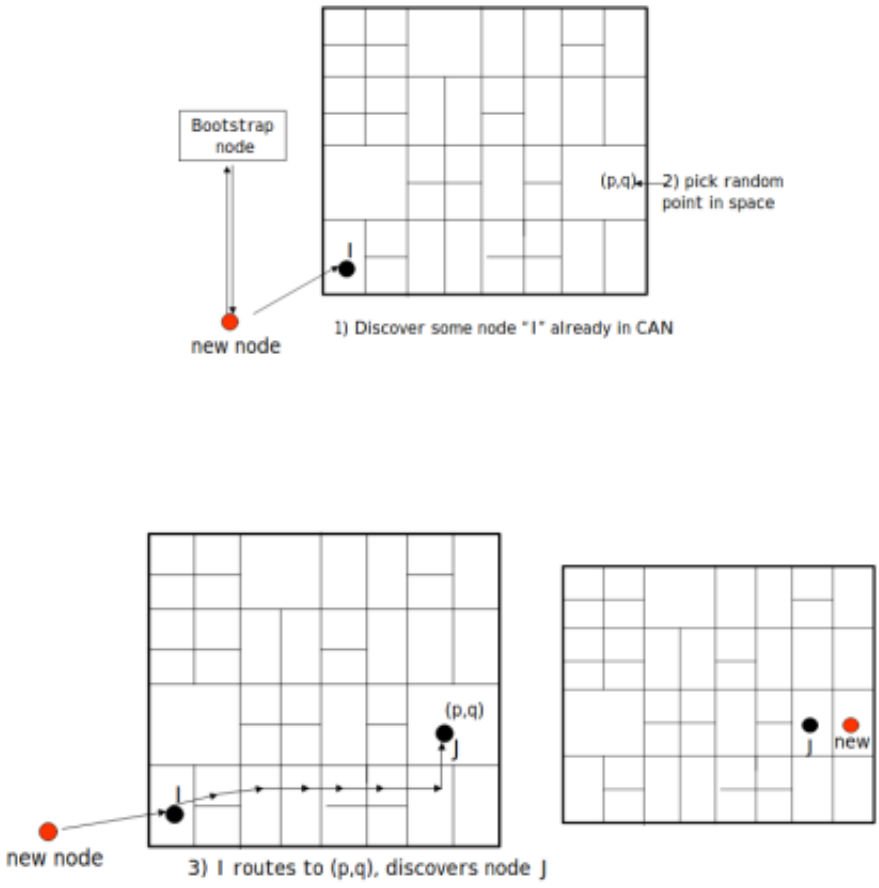

FIGURE 3.8 – Représentation de l'ajout d'un nœud dans un CAN

permettant de router un message dans un CAN.

Procédure ROUTE(c, P)

Route un message de c vers P et retourne le nœud p responsable de la zone contenant P

Si le point P est contenu par un voisin "n" de "c" alors on retourne directement n sinon on initialise p = c.

Tant que p ne contient pas P : 0n tente de se rapprocher le plus des coordonnées x et y de P :

Parmi les voisins de p on choisi le nœud d'abscisse la plus proche de x puis parmi les voisins de p on choisi le nœud d'ordonnée la plus proche de y.

Si le point P est trouvé ou si toutes les zones ont été visitées on retourne p.

Chapitre 4

Conclusion

Dans ce rapport nous avons présenté les réseaux de Petri et le fonctionnement de deux DHT : Pastry et CAN. De plus, nous avons tenté d'expliquer pourquoi ce projet se base sur ces deux notions spécifiquement (RdP et DHT).
Nous nous sommes intéressés à l'algorithme de construction d'un CAN (noeud qui rejoint le réseau) et l'algortihme d'extension du leafset pour Pastry en particulier pour la suite du projet.
A travers nos recherches, nous avons appris et assimilé de nouvelles informations que nous tenterons désormais d'appliquer sur des modélisations.
En réfléchissant à ces conceptions, nous sommes confrontés à une difficulté en particulier : la représentation peut facilement prendre des proportions conséquentes si aucune abstraction n'est faite.
Ainsi, la suite de ce travail consistera à chercher de nouveaux algorithmes et à les modéliser grâce aux réseaux de Petri. Nous tenterons ainsi de fournir un modèle pour l'édition du "model checking contest" de cette année.

Bibliographie

[1] Demetres Kouvatsos Alexandru Popescu Dragos Ilie. *A Scalable Content-Addressable Network*. URL : https://www.diva-portal.org/smash/get/diva2:836192/FULLTEXT01.pdf.

[2] Nada ALMASRI. *Modèle d'Administration des Systèmes Distribués à Base de Composants*. 2005. URL : https://tel.archives-ouvertes.fr/tel-00474407/document.

[3] Peter Druschel Antony Rowstron. *Pastry : Scalable, decentralized object location androuting for large-scale peer-to-peer systems*. URL : http://rowstron.azurewebsites.net/PAST/pastry.pdf.

[4] Vincent Augusto. *Cours Réseaux de Petri*. URL : https://www.emse.fr/~augusto/enseignement/icm/gis1/UP2-2-RdP-slides.pdf.

[5] Guillaume Burel. *Cours sur les tables de hachage*. URL : http://web4.ensiie.fr/~guillaume.burel/cours/IAP3/cours_hash.pdf.

[6] C.GiraultR.Valk. *Petri Nets for Systems Engineering*. 2001. URL : http://files.untiredwithloving.org/petribook.pdf.

[7] ENSIRAIL. *Réseaux de Petri ensirail*. URL : http://ensirail.free.fr/rdp.html.

[8] Fabrice Kordon. *Viral Epidemic Form, MCC 2020*. URL : https://mcc.lip6.fr/pdf/ViralEpidemic-form.pdf.

[9] Université de Montpellier. *Couverture RdP*. URL : http://www.lirmm.fr/~virazel/COURS/M1%20-%20HMEE111/HMEE111.pdf.

[10] Bassirou Ngom. *FreeCore : un système d'indexation de résumés dedocument sur une Table de Hachage Distribuée (DHT)*. 2020. URL : https://tel.archives-ouvertes.fr/tel-01921587v2/document.

[11] Scott Shenker Sylvia Ratnasamy Paul Francis. *A Scalable Content-Addressable Network*. 2001. URL : http://conferences.sigcomm.org/sigcomm/2001/p13-ratnasamy.pdf.

[12] Wikipedia. *Distributed hash table*. URL : https://en.wikipedia.org/wiki/Distributed_hash_table.

[13] Wikipedia. *Pastry (DHT)*. 2021. URL : https://en.wikipedia.org/wiki/Pastry_(DHT).

[14] Wikipedia. *Réseaux de Petri*. URL : https://en.wikipedia.org/wiki/Petri_net.