Architectures et protocoles pair-à-pair décentralisés et structurés

La grande majorité des architectures décentralisées structurées est basée sur des tables de hachage distribuées : les DHT (*Distributed Hash Tables*).

Ce paragraphe étudie les DHT et explique les principales caractéristiques de plusieurs modèles où elles sont utilisées.

Il existe toutefois des architectures décentralisées structurées n'utilisant pas de DHT ; par exemple : les *skip graphs* ou graphes à enjambement (cf. paragraphe 1.4.9 page 58).

1.4.1 - Les tables de hachage distribuées

Une fonction f d'un ensemble E vers un ensemble F est dite injective si : $\forall (x, y) \in E^2, (x \neq y) \Rightarrow (f(x) \neq f(y))$ Ceci garantit donc que : $\forall (x, y) \in E^2, (f(x) = f(y)) \Rightarrow (x = y)$

Une fonction de hachage est une fonction injective d'un ensemble de clés vers un ensemble de valeurs, qui à une chaîne d'octets de longueur quelconque associe une valeur unique (de taille fixe).

Une table de hachage représente un tableau de correspondance entre des clés et des valeurs liées par une fonction de hachage, *h*.

Une table de hachage distribuée (ou DHT pour *Distributed Hash Table*) est une table de hachage partagée entre tous les éléments actifs d'un système réparti.

Appliquée à une majorité de réseau P2P, une DHT partitionne un index global de ressources (ou objet) et distribue ces parties d'index sur plusieurs entités différentes, elles-mêmes indexées. Ces entités sont les pairs, et sont aussi considérées comme des ressources (puisque chaque pair met à disposition une partie de ses propres ressources).

Par ailleurs, chaque pair reçoit un identifiant unique, dit *nodeId*. Il en est de même pour chaque ressource, dont l'identifiant est dit *objectId*. Lorsqu'on parle d'une requête, l'*objectId* recherché

est dit *key*. Dans la littérature, on dit souvent, par exemple, « *le pair 3* », au lieu de dire « *le pair d'identifiant (ou de nodeId) 3* ». Il en est de même pour l'identification des objets.

Les *nodeId* et *objectId* sont dans un même espace logique *K* qui correspond à l'index global de la DHT. Chaque pair est responsable d'une partie de cet espace. Cette partie sera dite tout court : la DHT du pair.

Dépendamment de l'implémentation de la DHT, l'espace de hachage correspondant (et donc la DHT) peut être visualisé(e) comme une grille, un cercle (ou anneau), ou une ligne. L'implémentation de la DHT est spécifique à l'algorithme de routage du protocole mis en œuvre. Cet algorithme va permettre de construire la DHT en associant chaque *objectId* au pair responsable de l'objet correspondant.

En d'autres termes, étant donné une fonction de hachage *h* et un pair ayant une adresse IP *@IP* et voulant partager sur le réseau une ressource *r*, on a :

$$h(r) = k \in K \text{ ; où } k \text{ est l'} objectId$$

$$\text{et } h(@IP) = i \in K \text{ ; où } i \text{ le } nodeId$$

La ressource *r*, ou un lien vers cette ressource, est alors stockée sur la DHT du pair d'adresse *@IP* tel que dist(*k, i*) est minimale ; dist(*k, i*) étant la distance entre *k* et *i*, telle que définie dans l'espace logique *K*. Autrement dit, dans la DHT, *r* est indexée par le pair d'identifiant *i*. De même, tout pair d'identifiant *j* quelconque connaît de par sa DHT tous les pairs d'identifiant *m* quelconque, tel que la distance entre *j* et *m* est faible. Si *m* n'est pas « proche » de *k*, le pair d'identifiant *j* a dans sa DHT au moins un pair *n* strictement plus proche de *k*. Ainsi, par itération, *j* trouve *k* et peut récupérer la ressource *r*. [Vie05]

La figure 1.7 donne un exemple de mécanisme de routage dans un réseau *overlay* basé sur une DHT. Le casier jaune représente pratiquement une ligne de la DHT du nœud pointé.

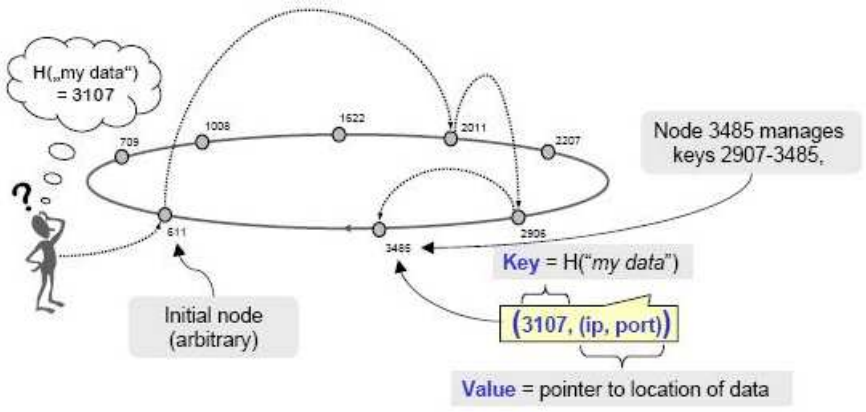

Figure 1.7 : *DHT et mécanisme de routage overlay*[1]

En utilisant les services de sa DHT chaque pair peut donc à tout moment :

- sur la base du *nodeId*, établir une communication directe ou indirecte avec un autre pair ;
- et grâce à l'*objectId*, localiser une ressource.

D'ailleurs, chaque DHT implémente des fonctionnalités de recherche (*lookup*) et de récupération (*retrieval*), assurant deux fonctions principales [Rhe05] :

- une fonction *put(key, data)* qui remplit la DHT ;
- et une fonction *get(key)* qui permet de localiser un objet à partir de son *objectId* (qui est la clé de la requête) en retournant le *nodeId* du pair responsable de l'objet. Ce pair-ci va alors fournir l'objet soit directement soit indirectement en indiquant où il peut être trouvé.

Les DHTs garantissent donc l'unicité de la clé et l'aboutissement des requêtes en un nombre minimal de sauts. Il s'agit là de sauts logiques, par opposition aux sauts physiques que sont les sauts IP. Au pire des cas, la localisation d'un objet nécessitera l'échange d'un nombre de messages logarithmique du nombre de pairs actifs.

[1] **In :** [BS06b]

La particularité des DHTs utilisées dans les architectures P2P est la fonction de hachage, qui est dite consistante (*consistent hashing* [KLL+97]). Elle garantit avec une probabilité élevée une distribution uniforme des ressources sur l'ensemble des nœuds et un déplacement de *O(1/N)* objets lors de l'arrivée ou du départ d'un *N*ième pair du réseau. En effet, la dynamique des pairs induit l'échange de messages et le déplacement d'objets pour préserver la structure et la cohérence de la DHT. Les architectures P2P à base de DHTs représentent alors de faibles coûts de maintenance et d'utilisation.

L'objectif principal des DHTs est le passage à l'échelle. Appliquées aux réseaux P2P, elles garantissent en plus les quatre propriétés suivantes [RM06] :

- *un degré de connexion faible* :
  chaque pair garde en mémoire (dans une liste ou table) un nombre restreint de connexions actives ;
- *un diamètre petit* :
  le nombre maximal de sauts nécessaires pour atteindre n'importe quel autre pair du réseau est minimisé ;
- *un routage "gourmand"* :
  chaque pair calcule tout seul le plus court chemin pour atteindre la destination ;
- *la robustesse* :
  un chemin vers la destination peut être trouvé même en cas de rupture de liens ou disparition de pairs.

Dans la dynamique des réseaux P2P, les pairs vont constamment s'organiser pour former un graphe de communication conforme avec l'implémentation de la DHT et ayant un rapport diamètre/degré optimal.

Les protocoles de routage dans les réseaux P2P qui utilisent des DHTs, et souvent appelés DHTs tout court, ont vu le jour dans les milieux de recherche académique. Les quatre premiers protocoles

sont : Chord [SML+01], Pastry [RD01], CAN (*Content-Addressable Network*) [RFH+01, Rat02] et Tapestry [ZKJ01, ZHS+04]. Depuis, de nombreuses DHTs ont émergés. Parmi les plus connues, citons : Kademlia [MM02], Viceroy [MNR02], Bamboo [RGR+04], et D2B [FG06]. Et la liste continue à s'allonger sans cesse bien qu'assez peu nombreuses sont les DHTs publiées avec une implémentation robuste. Mais Kademlia est d'ores et déjà intégrée dans les deux applications P2P de partage de fichiers, les plus populaires [Loe08, eMu04] : BitTorrent et eMule.

1.4.2 - Architecture en anneau

L'architecture en anneau est la plus simple qui puisse venir à l'esprit et la plus simple à gérer. Une DHT à géométrie circulaire permet la meilleure flexibilité et offre les meilleures performances de résilience et de proximité [GGG+03]. Dans une telle architecture l'espace d'identification est circulaire et la proximité est numérique. Chord [SML+01] en est un exemple. Il constitue aussi l'exemple type du routage P2P avec les DHTs.

Dans Chord, le réseau logique est un anneau orienté de 2^m pairs, où *m* est le nombre de bits de la fonction de hachage utilisée (souvent SHA-1 à 160 bits). Chaque pair est lié à un prédécesseur et plusieurs successeurs. Chaque objet est stocké sur le pair le plus proche dont le *nodeId* est supérieur ou égal à son *objectId*.

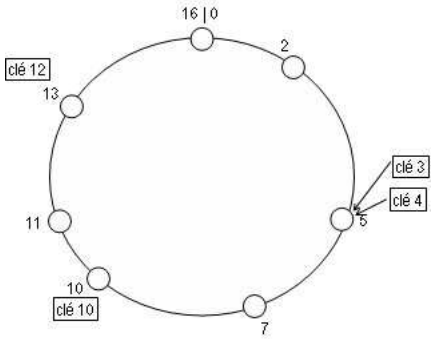

Figure 1.8 : *Un exemple d'anneau Chord*

La figure 1.8 montre la répartition de quatre objets sur les sept pairs actifs d'un réseau avec $m = 4$. Chaque objet y est représenté par le mot 'clé' suivi de l'*objectId*, et chaque pair actif y est représenté par son *nodeId*.

Une réplique de chaque objet est insérée sur chacun des successeurs du pair où il est stocké. La liste des successeurs est de taille fixe. Si tous les successeurs d'un pair disparaissent du réseau simultanément, l'anneau se scinde. Ce qui est cependant assez peu probable, même avec un petit nombre de successeurs par pair. Enfin chaque pair lance périodiquement une fonction de mise à jour de ses listes.

Chaque pair maintient une *finger table* qui joue le rôle de table de routage. Le premier pair de cette table est le premier nœud qui suit sur le cercle, c'est aussi un successeur. La figure 1.9 représente une *finger table* avec les détails de sa construction.

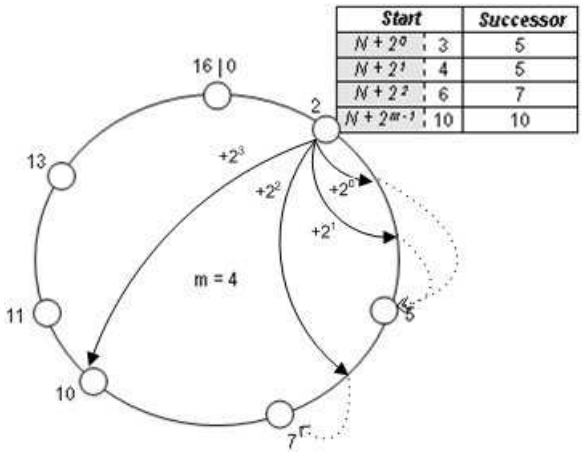

Figure 1.9 : *Un exemple de 'finger table'*

La figure 1.10 représente les sauts effectués par une requête consistant à trouver la clé 12 (l'objet d'identifiant 12) à partir du pair 2. À chaque saut l'on est au moins à mi-distance entre le pair émetteur de la requête et le pair cible, responsable de l'objet recherché. L'opération de recherche dans Chord est donc une fonction logarithmique du nombre total de nœuds pouvant adhérer à l'anneau.

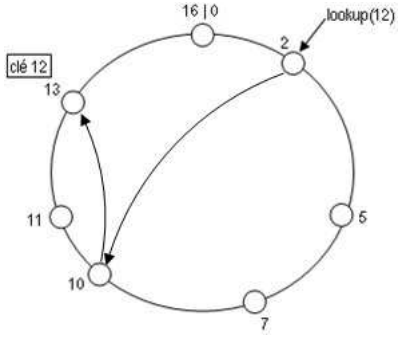

Figure 1.10 : *Un exemple du parcours d'une requête dans Chord*

Les *finger table*s sont aussi consultées pour l'insertion d'un nouveau nœud afin de lui indiquer ses prédécesseur et premier successeur, et donc son emplacement logique dans le réseau. Suite à cela, les *finger table*s de tous les nœuds sont mises à jour. (Préalablement à toute insertion, le nouveau nœud de *nodeId n*, par un mécanisme externe à Chord, prend connaissance d'un pair à proximité de lui et actif dans le réseau P2P sous le *nodeId* n_1. *n* envoie alors une demande d'adhésion à n_1. L'adhésion se fait à travers la fonction $n.join(n_1)$.)

Pour se retirer, un nœud peut avertir les nœuds présents dans sa *finger table*, afin de faciliter les mises à jour, qui seront à défaut faites à l'expiration d'un temporisateur.

La simplicité de Chord continue à inspirer plusieurs nouvelles DHTs : DKS (*Distributed K-ary Search*) [EAB+02] qui améliore les performances de Chord ; une hiérarchisation d'anneaux Chord [AS04, GGG04] ; Chord on Demand [MJB05] ; etc.

Chord est déployé dans de nombreuses applications, dont :

- CFS (*Collaborative File System*) [DKK+01], un système de fichiers distribués à l'échelle de l'Internet, et où chaque fichier est partagé en blocs ;
- ConChord [ACM+02], une infrastructure distribuée qui utilise CFS et délivre des certificats de sécurité SDSI (*Simple Distributed Security Infrastructure*) ;

– Chord-based DNS [CMM02] qui propose une implémentation P2P du DNS (*Domain Name Service*).

1.4.3 - Architecture maillée de Plaxton

L'algorithme de Plaxton [PRR97] a été développé pour les systèmes statiques distribués, formant une maille quelconque. Il peut donc localiser les objets en utilisant des tables de routage de taille fixe. C'est le premier algorithme applicable aux DHTs.

Les identifiants des nœuds et des objets sont numériques et indépendants de toute sémantique. Chaque nœud est une racine d'un arbre de recouvrement. Le routage en soi est de type *hyper-cube*, c'est-à-dire qu'il se fait par rapprochement successif d'un seul digit à la fois dans l'identifiant numérique jusqu'à atteindre l'objectif. Il s'agit d'un routage analogue au CIDR [RFC4632], le routage inter-domaine sans classe dans les réseaux IP. Il peut être basé soit sur le préfixe (comme dans Pastry [RD01]), soit sur le suffixe (comme dans Tapestry [ZKJ01, ZHS+04]).

1.4.3.1 - Tapestry

Dans Tapestry, l'espace d'identification est une maille quelconque et la fonction de hachage est fixée à 160 bits. L'identifiant de l'objet est désigné par GUID (*Global Unique IDentifier*) au lieu d'*objectId*. Des répliques de l'objet sont insérées avec des clés différentes le long du chemin menant du pair au sommet de l'arbre de recouvrement. Ce nœud racine identifie l'objet de manière unique. Le pair responsable de l'objet garde des pointeurs vers les pairs supports des répliques.

Le routage *hyper-cube* est basé sur le suffixe et la recherche est récursive. À défaut, le routage se fait vers le pair le plus proche numériquement dans le niveau hiérarchique supérieur ; on parle alors de s*urrogate routing*.

La figure 1.11 illustre les niveaux hiérarchiques dans Tapestry.

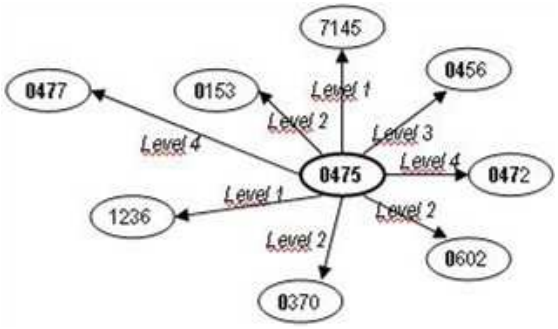

Figure 1.11 : *Exemple des niveaux hiérarchiques dans Tapestry*

La table de routage dans Tapestry contient les pairs les plus proches numériquement. Elle est formée de $b.Log_bN$ lignes et Log_bN colonnes, où b est la base d'identification des clés. Chaque colonne représente un niveau de suffixe commun avec le pair. En fin de colonne, des *back pointers* pointent vers les pairs reconnaissant le pair courant comme un de leurs voisins. Dans chaque ligne, chaque pair pointe vers des pairs voisins de même suffixe. Ces pointeurs sont classés par ordre de proximité physique (délai réseau).

Tapestry a servi de réseau de base à plusieurs applications, dont :

- OceanStore [Oce, KBC+00], une application de stockage massif déployée sur PlanetLab [Pla, PAC+02] ;
- SpamWatch [ZZZ+03], un système collaboratif de filtrage du pourriel.

1.4.3.2 - Pastry

Dans Pastry [RD01], l'espace d'identification est circulaire non orienté (à la différence de Chord [SML+01]). Les recherches peuvent donc se faire dans un sens ou dans

l'autre. La fonction de hachage est fixée à 128 bits. Les *nodeId*s sont en base 2^b, où b vaut souvent 2 ou 4. Ils peuvent aussi résulter d'un hachage de la clé publique du pair. Pastry se base donc sur une proximité numérique, mais prend aussi en considération la proximité réseau pour gérer les arrivées et départs des pairs. Sa méthode de routage hyper-cube se base sur les préfixes (cf. fig. 1.12).

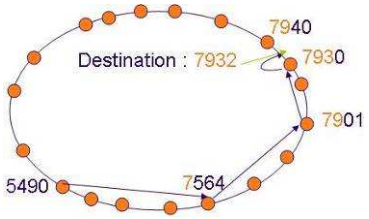

Figure 1.12 : *Exemple de routage dans un réseau Pastry*

Dans Pastry, chaque pair maintient en plus de la table de routage, un *leaf set* et un *neighborhood set*. (cf. fig. 1.13)

Exemple pour un nodeId 30230312

30230302	30230300	30230313	30230322	Leaf set
30230223	30230212	30230323	30230331	
-0-1321312	-1-1321300	-2-0311230	3	Table de routage
0	3-1-302210	3-2-230312	3-3-021031	
30-0-01120	30-1-21001	2	30-3-32112	
302-0-3213	302-1-1320	302-2-2203	3	
0	3023-1-210	3023-2-321	3023-3-312	
	30230-1-32	30230-2-12	3	
	1	302303-2-2		
		2		
32032312	01321312	32031302	11321300	Neighborhood set
21331201	13232010	20311230	31302210	

Figure 1.13 : *Exemple de table de routage dans Pastry*

La table de routage, proprement dite, contient $log_2{}^b N$ lignes. La $i^{ème}$ ligne contient des pairs ayant un préfixe de i digits communs avec le pair en question. Le nombre de colonnes contenant des *nodeId*s est de 2^b-1. Il reste une colonne qui contient un digit du *nodeId* du pair courant et

correspondant au numéro de la colonne. Le format des *nodeId*s dans la table de routage est donc :

préfixe commun – numéro de la colonne – suite du nodeId.

Le *leaf set* contient les *L* pairs les plus proches numériquement du pair en question ; *L/2* ayant un *nodeId* inférieur et autant ayant un *nodeId* supérieur. Les répliques des objets sont insérées sur chacun de ces *L* éléments, sans aucun contrôle de la part du pair détenteur de l'objet. Pastry diffère donc de Tapestry dans sa gestion de la proximité et des réplications.

Quant au *neighborhood set*, il contient *M* pairs proches du pair en question, au sens de la proximité réseau. Ils servent au maintient des propriétés locales.

Pour le routage, le pair consulte d'abord son *leaf set*, s'il y trouve la clé, il route directement vers ce pair-là. Sinon, il consulte sa table de routage, proprement dite, pour opérer un routage par saut selon Plaxton. Si à une des étapes, il n'y a pas de pair avec le *nodeId* duquel la clé partage un préfixe d'un digit plus long que celui partagé avec le *nodeId* courant, le routage s'opère vers un pair du *leaf set* qui serait numériquement plus proche de la clé.

Pastry a été avantagé par le développement de FreePastry [Fre] en code source libre et a servi de réseau de base à de nombreuses applications, dont :

- SCRIBE [RKC+01, CDK+02], une application de diffusion multi-groupes, de grande taille chacun ;
- PAST [DR01], un utilitaire de stockage massif ;
- Pastiche [CMN02], un système de sauvegarde qui exploite les capacités disponibles des antémémoires réparties dans le réseau.

1.4.4 - Architecture torique

Dans l'architecture en tore, l'espace d'identification est de dimension *d* à coordonnées cartésiennes. Il est uniformément partagé en zones ; chacune sous la responsabilité d'un pair. Les objets sont stockés en des points de l'espace. Chaque pair est responsable des objets de sa zone L'exemple type d'une DHT avec une structure torique est CAN (*Content-Addressable Network*) [RFH+01, Rat02].

Dans CAN, chaque zone peut être assimilée à un nœud d'un arbre binaire. La proximité des nœuds dans cet arbre est définie par la notion de voisins. Des zones voisines étant celles dont les coordonnées se chevauchent sur *d-1* dimensions et sont contiguës sur la dimension restante. Les pairs responsables de zones voisines sont alors des pairs voisins ; chacun connaissant l'adresse IP et les coordonnées de ses voisins.

Pour rechercher une clé, CAN hache cette clé suivant les différentes dimensions, localise le point résultat et repère le nœud responsable de la zone d'appartenance de ce point-là. Le routage s'effectue alors de proche en proche en passant par tous les voisins jusqu'à arriver au pair cible.

La figure 1.14 illustre un exemple de répartition et de routage dans un réseau CAN de dimension 2.

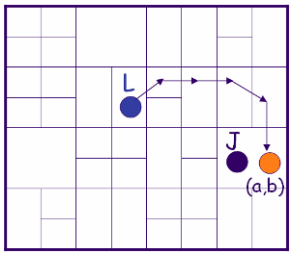

Figure 1.14 : *Exemple de routage dans CAN en dimension 2*

Afin d'améliorer les performances, notamment la latence du routage applicatif, l'espace entier avec sa DHT peut être dupliqué. On parle alors de « réalités » et un même pair occupera des zones différentes d'une *réalité* à l'autre.

1.4.5 - Architecture en papillon

L'architecture est dite papillon, ou *butterfly*, car les connexions établies entre les nœuds du réseau dessinent des formes en papillon. Ce type d'architecture a été initialement étudié pour les systèmes multiprocesseurs SIMD (*Single Instruction Multiple Data*) [Sie79].

Viceroy [MNR02] est un protocole de routage dans une architecture P2P *butterfly* (cf. fig. 1.15). En fait, Viceroy est une amélioration de Chord [SML+01] appliqué à une topologie annulaire à plusieurs niveaux, dite aussi multi-anneaux. De par l'architecture, chaque pair a une connectivité (ou degré) constante égale à 7. En effet, chaque pair maintient :

- *deux pointeurs généraux* : vers le nœud précédent et le nœud suivant sur le même niveau ;
- *deux pointeurs de niveau* : vers le prédécesseur et le successeur sur l'anneau de référence (généralement celui de niveau 1) ;
- *trois pointeurs 'butterfly'* : vers les nœuds à gauche et à droite sur le niveau d'indice supérieur et vers le nœud le plus proche sur le niveau d'indice inférieur. À noter que les indices des niveaux vont croissants vers le bas.

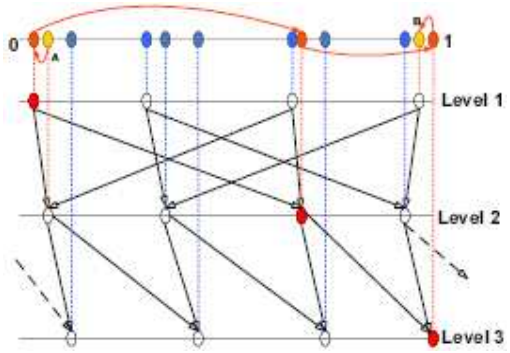

Figure 1.15 : *Illustration simplifiée de l'architecture de Viceroy, sans représentation des liens supérieurs et de niveau*[1]

Avec Viceroy, le routage se fait en commençant par le niveau du nœud qui lance sa requête, puis en remontant vers le niveau

[1] **In :** [LLX+04]

d'indice supérieur à l'aide des pointeurs de niveau, qui permettent par la suite de redescendre aux niveaux d'indices inférieurs, et ce récursivement jusqu'à satisfaction de la requête.

Ulysses [KMX+03] est un autre protocole de routage P2P utilisant une architecture en papillon. Basé sur Viceroy, il définit des liens supplémentaires entre un ensemble de pairs de niveaux différents. Ces liens servent de raccourcis en cas de congestion au niveau de certains pairs, due à la dynamique du réseau. Ils garantissent un système sans congestion et avec un trafic réparti équitablement.

1.4.6 - Architecture en arborescence

L'architecture en arborescence est une topologie plate où les nœuds sont les feuilles d'un arbre logique. Il n'y a pas d'hiérarchisation entre les nœuds. De par l'arborescence, l'espace d'identification est en base 2. Nous avons vu plus haut que CAN [RFH+01, Rat02] peut être transposé en une arborescence (cf. p. 51). Nous aborderons maintenant Kademlia [MM02] qui a déjà été introduit et fait ses preuves dans BitTorrent [Loe08] et dans eMule (sous le nom de Kad [eMu04, SR06]), les applications P2P les plus populaires pour le partage de fichiers.

À l'origine, Kademlia a été pensé comme une correction de certaines limites de Chord [SML+01]. L'espace d'identification de Kademlia est donc unidirectionnel, et la clé de hachage est sur 160 bits. Comme Pastry [RD01], Kademlia, utilise le routage hyper-cube basé sur le préfixe. Mais Kademlia a introduit une nouvelle métrique dans les réseaux P2P : le choix exclusif bit-à-bit (XOR) entre *n*, le *nodeId* courant, et *k*, la clé de l'objet à localiser. Plus *n* et *k* ont des bits en commun, plus *n XOR k* devient petit. À l'annulation, l'objet est localisé. Les requêtes contiennent le *nodeId* de l'initiateur. La figure 1.16 illustre un exemple de localisation de la clé1110, à partir du pair 0011.

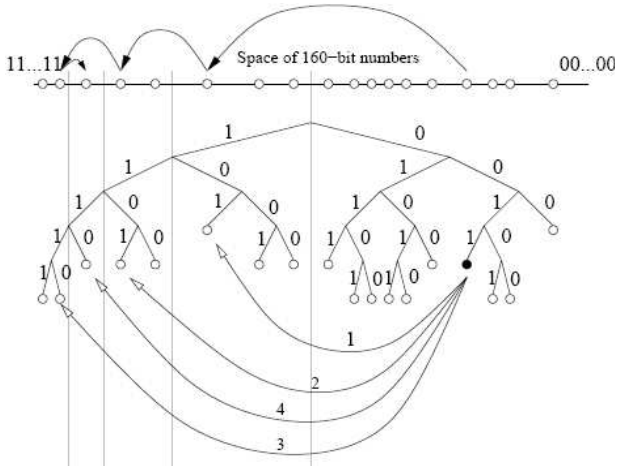

Figure 1.16 : *Localisation dans Kademlia de 1110 à partir de 0011*[1]

Dans Kademlia, les pairs communiquent grâce à UDP (*User Datagram Protocol*) et chaque pair maintient des listes finie de triplets {adresse IP, numéro de port UDP, *nodeId*} relatifs aux derniers pairs contactés. Chaque liste est régie par le mode d'ordonnancement LRU (*Least Recently Used*) qui avance en premier les éléments les moins utilisés. Kademlia avantage ainsi les pairs les plus anciens dans le réseau. Ceci assure une résistance active à l'intrusion de nœuds malicieux et à certaines attaques de type déni de service (DoS – *Denial of Service*).

1.4.7 - Graphes de 'de Bruijn'

Les graphes de *de Bruijn* portent le nom du mathématicien qui les a introduits [deB46]. Il s'agit de graphes orientés $B(k,l)$ dont les sommets sont toutes les suites possibles de longueur l formées des symboles de l'alphabet $A=\{0, 1, \ldots, k-1\}$ et dont les arcs joignent deux sommets f et g, tel que $f=xh$ et $g=hy$, où x et y sont des symboles de A et h une suite quelconque de $l-1$ symboles de A.

Chaque nœud d'un graphe $B(k,l)$ de *de Bruijn* connaît k autres nœuds, avec k variable selon le degré de robustesse désiré. En passant d'un nœud à un autre, on parle de routage par décalage. Le

[1] **In :** [MM02]

décalage peut se faire vers la gauche ou la droite, selon l'orientation des arcs ; il est défini par l'algorithme du protocole.

La figure 1.17 illustre le graphe *B*(2,3) de *de Bruijn*.

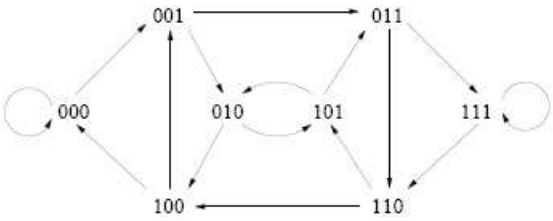

Figure 1.17 : *Exemple d'un graphe* B(2,3) *de 'de Bruijn'*[1]

Ces dernières années plusieurs systèmes P2P basés sur les graphes de '*de Bruijn*' et utilisant les DHTs ont vu le jour. Nous citons :

- Koorde [KK03] qui étend Chord [SML+01] pour atteindre les performances d'un graphe *B*(2,*l*) de *de Bruijn*. C'est l'une des DHTs qui représente le taux de latence le plus faible, en l'absence de congestion ;
- Broose [GV04], qui adapte Kademlia [MM02] à la topologie de *de Bruijn* ;
- D2B [FG06] qui adapte CAN [RFH+01, Rat02] aux graphes de *de Bruijn* ;
- Optimal Diameter Routing Infrastructure (ODRI) [LKR+03] pour les réseaux à degré constant.

1.4.8 - Analyse comparative des principales architectures avec DHT

Les performances des protocoles de routage utilisant des DHTs et dédiés à des architectures P2P décentralisées et structurées sont évaluées par le degré et le diamètre du protocole. Le tableau 1.1 donne les performances des principaux protocoles présentés, ainsi que celles intéressantes d'autres protocoles dont il a été fait mention plus haut dans ce chapitre.

[1] **In :** [FG06]

Tableau 1.1 : *Comparaison des degré et diamètre de quelques protocoles de routage P2P utilisant des DHTs*

Protocole	***Degré***	***Diamètre***
Chord et Kademlia	$O(\log N)$	$O(\log N)$
DKS	$O((k-1).\log_k N)$	$O(\log_k N)$
Tapestry et Pastry	$O(\log_2{}^b N)$	$O(\log_2{}^b N)$
CAN	$O(d)$	$O(dN^{1/d})$
Viceroy	$O(1)$	$O(\log N)$
Ulysses	$O(k)$	$O(\log N / \log k)$
Les graphes de 'de Bruijn'	k (variable)	$O(\log_k N)$
Koorde et Ulysses (avec $k \rightarrow \log N$)	$O(\log N)$	$O(\log N / (\log (\log N)))$

Ci-suivent quelques remarques d'analyse comparative.

- Malgré des performances bien intéressantes, Ulysses [KMX+03] reste lourd à mettre en œuvre, notamment à grande échelle, et ce, vu les nombreux liens croisés à gérer.
- Les architectures en anneau et celles utilisant la métrique du choix exclusif (XOR) permettent la meilleure flexibilité quant à la disposition des pairs voisins et au choix de routes alternatives (résilience) [GGG+03].
- Les graphes de '*de Bruijn*' sont plus fiables qu'une architecture en anneau simple ou une architecture torique, mais les routes alternatives sont indépendantes entre elles [LKR+03]. Ces graphes présentent d'ailleurs une connectivité intéressante, mais leur structure est assez rigide : pour une distribution aléatoire des clés, ils ne peuvent pas assurer à la fois un bon équilibrage de charge et une recherche efficace des objets [DGA04].

1.4.8.1 - Limites des tables de hachage distribuées

Les DHTs garantissent aux architectures P2P le passage à l'échelle d'un routage performant (évalué par le diamètre), tout en maintenant pour chaque pair une connectivité raisonnable (évaluée par le degré). Cependant, ces structures distribuées établissent certaines limites aux systèmes P2P qui les utilisent.

- La répartition des éléments dans l'espace d'identification d'une DHT dépend à tout moment du nombre total de pairs présents dans le système. La dynamique continue des pairs amène donc un changement permanent dans la répartition et l'affectation des objets.
- Les DHTs ne peuvent pas traiter des requêtes complexes de type base de données ou intervalle. En effet, la fonction de hachage ne respecte ni la similitude ni l'ordonnancement des clés.
- Les identifiants référencés dans les DHTs (*nodeId*s et *objectId*s) n'ont aucune sémantique. En fait, la fonction de hachage broie toute sémantique et n'en crée aucune.
- Les DHTs font abstraction du réseau *underlay*, et leurs performances sont mesurées en sauts logiques. Elles supposent donc tous les pairs issus d'un même domaine administratif et les liens physiques les reliant homogènes et équitablement chargés.
- Les DHTs font aussi abstraction des différentes ressources réseaux disponibles (e.g. processeurs, capacités de stockage, etc.) et de leurs répartitions réelles (au niveau de chaque pair).
- Enfin, comme tout réseau, ceux à base de DHTs sont perméables aux différentes attaques et problèmes de sécurité (e.g. pair malicieux, faux-contenu si les objets

sont des fichiers, surcharge ciblée, messages parasites, etc.) [SM02].

1.4.9 - Graphes à enjambement

Les graphes à enjambement, ou *skip graph*, sont des architectures P2P décentralisées et structurées n'utilisant pas de DHT. Ils ont été développés sur base des listes à enjambement, ou *skip list*. Dans la suite sont utilisés les termes anglais : *skip list* et *skip graph*.

1.4.9.1 - Skip lists

Les *skip list*s ont été inventées en 1990 par W. Pugh [Pug90]. Il s'agit d'une structure de données probabiliste, organisée en plusieurs niveaux parallèles de listes chaînées. Le niveau le plus bas est une liste chaînée standard ; les niveaux supérieurs sont des listes dérivées de la liste de base suivant une certaine probabilité, permettant ainsi un parcours rapide de la liste de base. (cf. fig. 1.18)

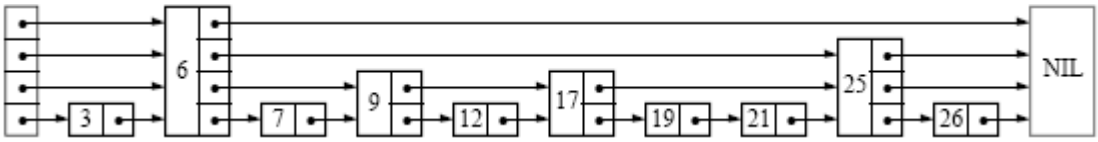

Figure 1.18 : *Exemple d'une 'skip list'*[1]

Le principal atout des *skip list*s est qu'elles ne nécessitent aucune connaissance préalable du nombre total de ses éléments (d'ailleurs indéfini). L'insertion et la suppression se passent comme dans une liste chaînée, sauf que les éléments présents à plus d'un niveau doivent être insérés et supprimés de tous ses niveaux. La recherche d'une clé est aussi similaire au mécanisme de recherche dans une liste chaînée. Elle s'opère successivement dans

[1] **In :** [Pug90]

chacune des listes, en commençant par la liste supérieure en premier. Une liste inférieure n'est parcourue que si la requête n'a pas été satisfaite dans les listes supérieures. La figure 1.19 illustre les modalités de recherche pour l'insertion d'un nouvel élément dans la *skip list*.

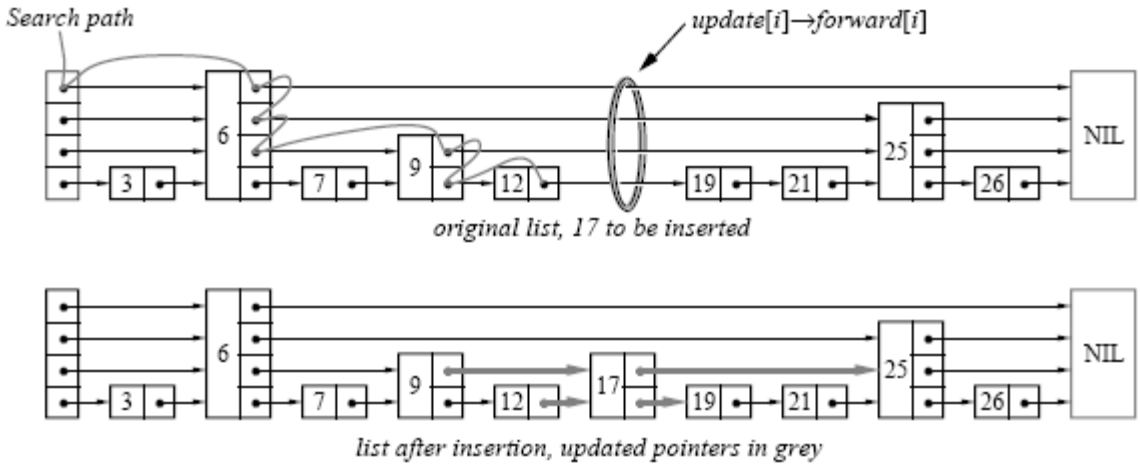

Figure 1.19 : *Illustration du parcours d'une* skip list *et de l'insertion d'un élément dans celle-ci*[1]

1.4.9.2 - <u>*Skip graphs*</u>

À la différence d'une *skip list*, un *skip graph* présente plusieurs listes par niveau afin d'assurer une certaine redondance et chaque nœud est présent à tous les niveaux.

Dans les *skip graph*, les nœuds sont identifiés par une chaîne aléatoire de bits, en base deux, et de longueur *l-1* ; *l* étant le nombre de niveaux. Chaque nœud indexe ses propres objets et choisit ses voisins en fonction des identifiants des objets qu'ils stockent.

Un vecteur d'appartenance, ou *membership vector*, est lié à chaque nœud pour lui permettre de reconnaître les listes chaînées auxquelles il appartient. Chaque liste chaînée est étiquetée par un mot de taille finie. Le vecteur

[1] **<u>In</u> :** [Pug90]

d'appartenance est un mot aléatoire de longueur indéfinie mais composé à partir d'un certain alphabet de taille finie. Un nœud appartient à toutes les listes chaînées dont l'étiquette est un préfixe de son vecteur d'appartenance.

La figure 1.20 illustre un exemple de *skip graph* à trois niveaux avec six nœuds.

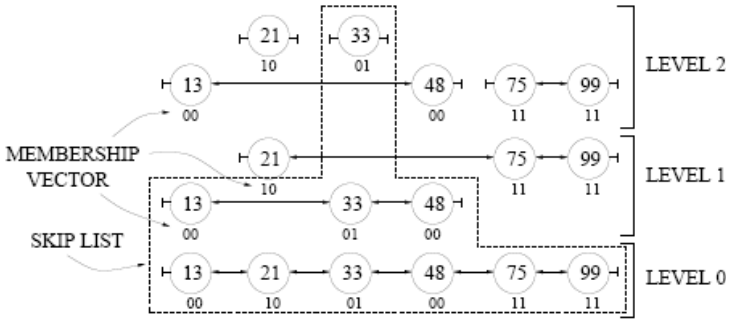

Figure 1.20 : *Un 'skip graph' de six nœuds à trois niveaux*[1]

1.4.9.3 - <u>*SkipNet*</u>

Parallèlement aux *skip graph* [AS03], l'idée d'utiliser les *skip lists* pour le routage P2P structuré a abouti aussi à SkipNet [HJS+03]. SkipNet se présente comme un cas d'usage des *skip graph*, garantissant la localité des chemins et des contenus. La figure 1.21 illustre une infrastructure SkipNet à huit nœuds.

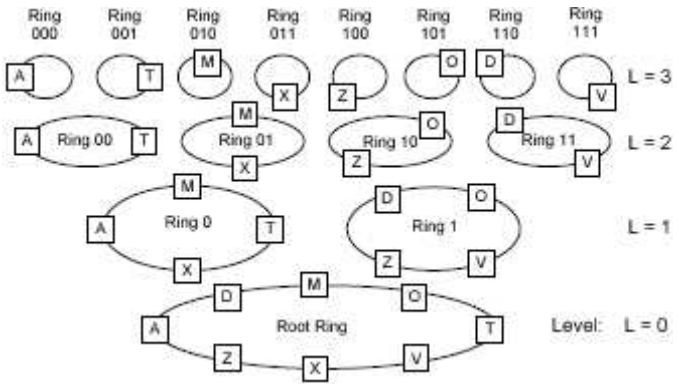

Figure 1.21 : *Une infrastructure SkipNet à huit nœuds*[2]

[1] **<u>In</u> :** [AS03]
[2] **<u>In</u> :** [HJS+03]

Dans SkipNet les nœuds ont deux identifiants : un numérique (*numericId*) et un nominal (*nameId*). La recherche d'objets peut alors se faire sur la base soit de l'identifiant nominal, soit de l'identifiant numérique. La recherche en mode nominal est une recherche par domaine (comme la recherche DNS). Un exemple clair est donné dans [RM04] : pour trouver le document *docname* se trouvant sur le nœud *user.compagny.com*, la requête parcourt les listes à la recherche du préfixe *com.company.user*. Quant à la recherche en mode numérique, elle se fait en remontant les niveaux à partir du niveau le plus bas, de sorte à avancer d'un digit par niveau jusqu'à destination. Dans les deux cas, à défaut de pouvoir avancer, le routage vers l'élément suivant se fait de façon aléatoire mais en se rapprochant de la destination.

1.4.9.4 - Analyse des graphes à enjambement

Dans un *skip graph*, les objets sont référencés par leurs identifiants logiques, sans passer par le service d'une fonction de hachage. Similitudes et ordonnancements des clés sont donc conservés. Les requêtes complexes de type intervalle deviennent alors possibles, sans qu'elles soient fractionnées ou répétées. Elles s'exécutent dans une sous-liste.

L'absence de fonction de hachage fait que chaque pair est seul responsable de ses objets. Ceci accroît la sécurité du système et diminue le temps de latence entre la localisation d'un objet et sa récupération. La gestion des objets est aussi simplifiée et leur emplacement plus flexible que dans une DHT.

La gestion du système est cependant moins équitablement répartie entre les pairs. En fait, la charge de

chaque pair est fonction du nombre d'objets qu'il partage. Chaque pair physique est représenté par autant de pairs logiques que d'objets qu'il met en partage, et avec *k* clés, il y a *O*(*log k*) niveaux dans le réseau. Le degré d'un pair est donc aussi en *O*(*log N*). Chaque nœud physique gère donc un nombre de liens différent et bien plus important que dans une DHT. L'importance des trafics de maintenance et de mise à jour est tout autant conséquente. Le diamètre reste pourtant en *O*(*log N*).

Comme les DHTs, les *skip graph*s font abstraction des ressources réseaux disponibles. Quant à la localité structurelle du réseau *underlay* (e.g. domaines administratifs), elle est conservée dans SkipNet. Aussi SkipNet, permet de par l'identifiant nominal de garder une sémantique, pour les objets uniquement.

CONCLUSION DU CHAPITRE –

Ce chapitre, après une présentation des réseaux P2P et de leurs différentes classes d'architecture, s'est consacré aux principaux protocoles P2P. Il s'est particulièrement étendu sur ceux à base de DHT et s'est terminé par ceux utilisant les graphes à enjambement. Les deux types de systèmes ont été analysés : il en ressort que les derniers pallient certaines limites inhérentes aux premiers, mais non sans introduire des surcharges. Les DHTs ont toutefois déjà fait leurs preuves dans les applications P2P les plus populaires. D'où leur attractivité pour être améliorées en vue de meilleures performances.