ROSA: Un Réseau de Recouvrement Adaptable, Auto-Organisant et Extensible

Baud Loic

Résumé

Les travaux présentés dans cette thèse se déroulent dans le cadre des réseaux de recouvrement (overlay networks). Un réseau de recouvrement est un réseau construit au-dessus d'un autre réseau. A chaque nœud du réseau de recouvrement correspond un nœud du réseau recouvert. Les réseaux de recouvrement ont été popularisés par l'expension des réseaux pair-à-pair (peer-to-peer networks) dans les années 2000. Il existe de nombreux types de réseaux de recouvrement, certains sont extensibles d'autres non, certains ont pour rôle d'assurer une topologie résiliente, certain offre un service de routage fiable, etc. Mais aucun des réseaux de recouvrement n'est adaptable à différents types de réseaux recouvert et ne propose un large éventail de services. Tout réseau de recouvrement existant est uniquement dédié à une tâche particulière. On peut imaginer un réseau qui pourrait être déployé à cheval sur de nombreux types différents de réseau et ne se consacrant pas seulement à une tâche particulière. Ma thèse a pour objectif de définir les bases et de développer un tel réseau de recouvrement.

Dans ces travaux de recherches, nous proposons un nouveau réseau de recouvrement appelé ROSA. ROSA est conçu de façon à pouvoir facilement être adapté aux différents réseaux physiques et d'être en mesure de fournir un large éventail de services différents. Les nœuds de ROSA sont organisés en cluster appelé grumeaux (lump) et ROSA peut être considéré comme un enchevêtrement de grumeaux. Les nœuds organisent leurs ensembles de voisins en fonction des densités associées à ces grumeaux. ROSA est extensible car le nombre maximal de voisins qu'un noeud peut avoir est borné, et cette limite ne dépend pas du nombre total de nœuds participant au réseau. ROSA est adaptable car la définition de la densité des grumeaux peut être modifiée et adaptée aux propriétés du réseau recouvert. La densité définit le comportement de ROSA.

Afin d'offrir un routage efficace sur ROSA, nous développons une table de hachage distribuée qui peut être déployée sur ROSA. Cette DHT peut être ajoutée à ROSA, sans avoir à modifier le protocole initial. Puisque les services complexes qui sont proposés sur ROSA sont basés sur cette DHT, la DHT doit posséder un service de recherche efficace. Ce service de recherche doît être par conséquent extensible.

Cette thèse propose également un service de stockage de fichiers fiable et un service de routage résilient sur ROSA. Le stockage de fichiers fiable utilise la DHT et son service de recherche pour permettre aux utilisateurs de stocker des fichiers sur l'ensemble des grumeaux de ROSA. Le service de routage résilient est également basé sur la DHT, elle permet à chaque nœud de ROSA de communiquer de façon directe avec un autre noeud. Ces deux services montrent qu'il est possible de proposer des services complexes sur ROSA.

Keywords: *Overlay network, network virtualization, Distributed HashTable, Resiliency, Routing, File Storage*

Abstract

The works presented in this thesis take place within the context of the overlay network. An overlay network is a network built on top another network. To each node of the overlay network corresponds a node of the underlying network. These overlay networks have been popularized with the rise of the peer-to-peer networks (that belong to a sub-class of the overlay networks) in the years 2000. There exists many kinds of overlay networks, some are scalable, some ensure a resilient topology, some offers a reliable routing service, etc. But none of the existing overlay network are adaptable to different kind of covered network and propose a large set of services. Any existing overlay network is only dedicated to a particular task. One can imagine an overlay network that could be deployed astride many different kind of underlying network and not dedicating to only a particular task. My thesis aims to define the basis and develop such an overlay network.

In this PhD research we propose an new overlay network called ROSA. ROSA is designed to be used easily **adaptable to different physical networks** and to be able **to provide a large set of different services**. The nodes of ROSA are organized in cluster called lumps and ROSA can be seen as an entanglement of lumps. The nodes organize their neighbor sets according to the lumps and to the densities associated to these lumps. ROSA is scalable because the maximum number of neighbors that a node can have is bounded, and this bound does not depend on the total number of nodes participating to the network. ROSA is adaptable because the definition of the density of the lumps can be adapted to the properties of the underlying network. The density defines the behavior of ROSA.

In order to provide an efficient routing over ROSA, we develop a Distributed HashTable that can be deployed over ROSA. This DHT can be added to ROSA without modifying the initial protocol. Since some efficient complex services must be built over ROSA using this DHT, the DHT proposed provides an efficient lookup service. This lookup service is scalable.

This thesis also proposes a reliable file storage service and a resilient routing service over ROSA endowed with the DHT. The reliable file storage uses the DHT and its lookup service to allow users to store file over the set of lumps of ROSA. The resilient routing service is also based of the DHT, it allows a node of ROSA to communicate in a direct way to any other node. These two services demonstrate that it is possible to propose complex services over ROSA.

Keywords: *Overlay network, network virtualization, Distributed HashTable, Resiliency, Routing, File Storage*

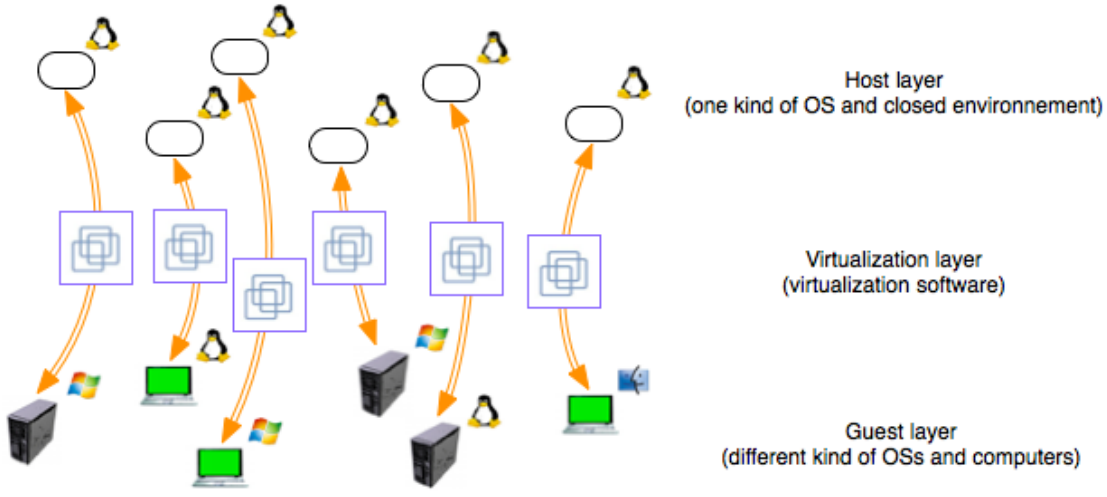
Contents

Chapter 0

Résumé

Contexte

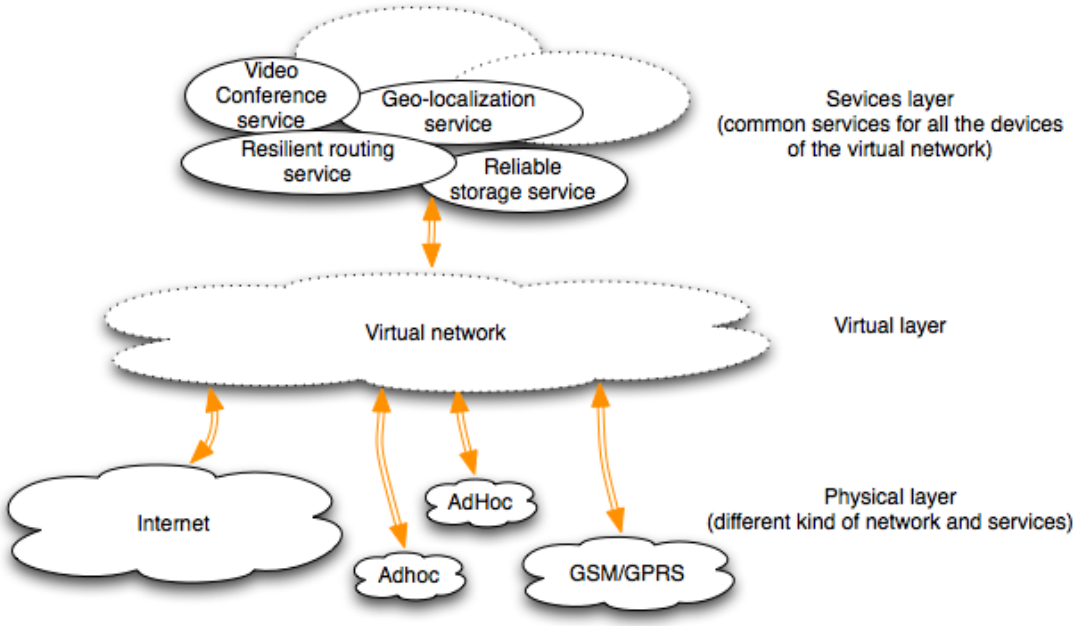
Récemment, il est devenu possible de faire tourner un système d'exploitation sur une architecture matérielle pour laquelle il n'était pas destiné. Cette technologie s'appelle la virtualisation. La virtualisation fait tourner chacun des OS invités dans un environnement fermé. Ces environnements virtualisés sont gérés par la couche de virtualisation qui doit être implémentée pour beaucoup de systèmes hôtes. Il doit également être en mesure d'exécuter un grand nombre d'OS. Par conséquent, la virtualisation peut se décomposer en trois couches, la couche host (ou la couche physique), la couche de virtualisation et de la couche guest. Ceci est schématisé dans la figure ci-dessous.



Ce concept de virtualisation peut également être appliqué aux réseaux. Aujourd'hui, il existe une grande disparité dans les réseaux: Internet, le réseau GSM, les réseaux ad hoc, etc. Chacun de ces réseaux est basé sur différents supports physisques et différents protocoles. Il serait intéressant d'avoir un réseau virtuel qui unifie tous ces réseaux. En outre, la convergence des appareils électroniques numériques facilite le développement d'un tel réseau virtuel. En effet, les appareils, tels que les téléphones de nouvelle génération,

sont à cheval sur plusieurs réseaux physiques (IP, WiFi et GSM/GPRS) et peuvent servir de passerelles entre ces différents réseaux physiques.

Le modèle de ce réseau virtuel global est definit en trois couches: la couche physique, la couche virtuelle, et la couche de services. La couche physique est un ensemble de réseaux physiques et les différents dispositifs qui les composent. La couche virtuelle est un réseau virtuel construit au-dessus de ces réseaux physiques ainsi que l'interfaçage avec chacun de ces réseaux. La couche de services consiste en un ensemble de services fournis au dessus du réseau virtuel. Les services peuvent être des services de routage résilient, services de stockage fiable, etc Ce modèle est schématisé dans la figure ci-dessous.



Une des principale conditions qu'un tel réseau virtuel doit respecter est l'extensibilité. Si l'on considère que beaucoup de dispositifs sont susceptibles de participer, il faut que le trafic supplémentaire généré soit le plus bas possible et si possible que ce traffic ne dépende pas du nombre de participants. Le réseau virtuel doit également avoir les mêmes qualités (résistance, etc.) que les réseaux sur lesquels il est déployé. Pour finir, ce réseau virtuel doit être facilement adaptable à un grand nombre de réseaux physiques différents.

Les réseaux de recouvrement sont de très bons candidats pour être utilisé comme un réseau virtuel. Un réseau overlay est un réseau construit sur un autre réseau. Pour chaque nœud du réseau overlay correspond un nœud du réseau sous-jacent. Ces réseaux de recouvrement ont été popularisés par les réseaux pair-à-pair dans les années 2000. Il existe de nombreux types de réseaux de recouvrement, certains sont extensibles, certains assurent une topologie résiliente, certains offrent un service d'acheminement fiable, etc. Néanmoins, aucun de ces réseaux existants ne peut être utilisé pour être le réseau virtuel, puisque ces réseaux sont généralement dédiés à un seul service et ne peuvent être déployées que sur un seul type de réseau physique. Afin d'utiliser un réseau de recouvrement en tant

que réseau virtuel, celui-ci doit être facilement adaptable à un grand nombre de réseaux physiques différents et proposer de nombreux services. C'est suite à ces observations, que les objectifs de cette thèse ont été définis:

- concevoir et construire un réseau de recouvrement adaptable et facile à déployer sur de nombreux types des réseaux physiques;
- proposer un ensemble de services au dessus de ce réseau de recouvrement.

Contributions majeures

Cette recherche de doctorat contribue à l'élaboration du réseau recouvrement de la couche virtuelle défini dans le modèle présenté précédemment par:

Premièrement, la conception d'un réseau de recouvrement appelé ROSA. Les nœuds de ROSA sont organisés en cluster appelé grumeaux et ROSA peut être considéré comme un enchevêtrement de grumeaux. Les nœuds organisent leur ensemble de voisins selon les densités associées à ces grumeaux. ROSA est extensible car le nombre maximal de voisins qu'un noeud peut avoir est borné, et cette limite ne dépend pas du nombre total de noeuds participant au réseau. ROSA est adaptable car la définition de la densité des grumeaux peut être adaptée aux propriétés du réseau sous-jacent. ROSA remplit les conditions nécessaires pour être le réseau virtuel.

Deuxièmement, la conception d'une table de hachage distribuée sur le réseau de recouvrement ROSA. Cette DHT peut être ajouté à ROSA sans modifier le protocole initial. Etant donné que certain des services proposés au dessus de ROSA sont complexes et doivent utiliser cette DHT, la DHT doit fournir un service de recherche efficace. Ce service de recherche doit aussi être extensible.

Troisièmement, la conception d'un service de stockage de fichiers fiable et d'un service de routage résilient. Le stockage de fichiers fiable utilise la DHT et son service de recherche pour permettre aux utilisateurs de stocker des fichiers sur l'ensemble des grumeaux de ROSA. Le service de routage résilient est également basé sur la DHT et permet à un nœud de ROSA de communiquer de façon directe avec un autre noeud. Ces deux services montrent qu'il est possible de proposer des services complexes sur ROSA.

Le protocole de ROSA

0.1 Le principe de ROSA

Les nœuds de ROSA sont organisés en clusters appelés grumeaux. Un grumeau est un ensemble de noeuds entièrement connectés. En théorie des graphes de tels objets sont appelés cliques. Une clique dans un graphe non orienté est un sous-ensemble de l'ensemble des sommets, tel que pour chaque pair de sommets du sous-ensemble, il existe une arête reliant les deux sommets. ROSA peut être représenté par un enchevêtrement de grumeaux. Chaque noeud de ROSA appartient à au moins un des grumeaux. A chacun des grumeaux de ROSA est associé une mesure appelée densité. Il y a plusieurs types de densités et c'est le choix de cette densité qui définit les caractéristiques de ROSA.

Nous appelons la gestion de la topologie la façon dont le graphe composé des nœuds et des liens et l'ensemble des grumeaux de ROSA évolue au cours du temps et selon les événements.

Le principe de gestion de la topologie de ROSA est semblable à une recette de pâte à crêpe. Dans une recette d'une pâte à crêpe, on doit diluer les grumeaux avec de fortes densités afin d'augmenter la densité des zones avec moins de farine. Dans ROSA, les nœuds calculent périodiquement la densité des grumeaux auxquels ils appartiennent, partagent leur connaissance à propos des grumeaux de faibles densités avec leurs voisins et quittent les grumeaux ayant des densités élevées pour joindre les grumeaux avec de faibles densités et ainsi augmenter la densité de ces grumeaux.

0.2 Representation d'un nœud, d'un voisin et d'un grumeau

Chaque nœud de ROSA a un identifiant, une liste des voisins et une liste de grumeaux. L'identifiant d'un nœud est un entier appartenant à $[1, 2^{128} - 1]$. Nous considérerons pour l'instant, que l'identificateur d'un nœud est choisi au hasard. La liste des grumeaux d'un nœud est une liste de grumeaux auquel le nœud appartient. La liste des voisins d'un nœud est une liste des identifiants et des adresses physiques des noeuds auxquels il est lié. Chaque noeud possède aussi un indicateur 'connecté'. Cet indicateur est défini à true si le noeud est connecté à ROSA et mis à false sinon. Pour compléter la représentation d'un nœud nous ajoutons une liste de grumeaux de faible densité et une liste des identificateurs des derniers messages reçus. Ces deux listes sont initialisées à vide. La premiere sera utilisée pour stocker la connaissance que chaque noeud a à propos des grumeaux de faibles densités dans son entourage. La seconde sera utilisée pour permettre à tous les nœuds d'un grumeau de recevoir un message, malgré les échecs de transmission.

Un voisin d'un nœud est représenté par le couple (id, phy) où id est l'identifiant de ce voisin dans ROSA et phy son adresse physique sur le réseau sur lequel ROSA est déployé. Un indicateur 'vivant' complète cette représentation. Cet indicateur est défini sur true par défaut et sera utilisé pour la détection des pannes.

Un grumeaux possède un identifiant, la liste des identifiants des noeuds qui le composent et toutes les informations permettant à chacun de ces nœud de calculer sa densité. Les informations nécessaires pour calculer la densité dépendent de la définition de la densité choisie. L'identifiant d'un grumeau est un entier appartenant à $[1, 2^{128} - 1]$. Un lump possède aussi une liste des adresses physiques des noeuds qui le composent.

0.3 Protocole

Dans cette section, nous présentons la base du protocole de ROSA. Nous nous concentrerons sur quelques fonctions primitives utilisées par les nœuds de ROSA, comme par exemple la fonction qui permet à un noeud de se connecter, de quitter ROSA, etc.

0.3.1 Connexion et initialisation de ROSA

Pour se connecter à ROSA un nœud doit connaître un noeud déjà connecté à ROSA, ce nœud est appelé bootstrap_node. Si le noeud ne peut pas trouver un tel bootstrap_node, il doit initialiser le réseau.

Dans l'implémentation actuelle, la découverte d'un bootstrap_node se fait en utilisant un serveur central appelé bootserver. Ce serveur maintient une liste de nœuds actifs sur ROSA. Pour se connecter à ROSA, un nœud doit communiquer avec le bootserver. Celui-ci répond avec l'adresse physique d'un noeud déjà connecté. Le bootserver sélectionne ce nœud au hasard dans sa liste. Si la liste est vide, le bootserver réponses avec l'adresse null. Cette adresse null indique au nœud qui veut se connecter à ROSA qu'il est le premier nœud et qu'il doit initialiser ROSA.

Afin de maintenir à jour la liste des noeuds connectés à ROSA, tous les noeuds connectés doivent périodiquement donner des signes de vie au bootserver. Le bootserver supprime de sa liste de tous les nœuds qui ne font pas cela.

Si le nœud désirant se connecter à ROSA est le premier il doit initialiser ROSA. Pour cela, le nœud crée un grumeau uniquement composé de lui-même. Ensuite le noeud ajoute le grumeau nouvellement créé à sa liste de grumeaux. Nous verrons plus tard que ce grumeau est le seul qui sera créé à partir de rien. Tous les autres grumeaux de ROSA résultent d'une scission d'un grumeau déjà existant. Pour terminer le noeud positionne son indicateur 'connecté' à true.

0.3.2 Quitter ROSA

Deux méthodes s'offrent à un nœud qui veut quitter ROSA. La première est la bonne méthode et doit être utilisée chaque fois que cela est possible. L'autre méthode est beaucoup moins élégante.

La première méthode consiste pour un nœud à appeler la fonction LeaveROSA. Un nœud utilisant la première méthode se contente d'envoyer un message *LeaveROSA* à tous ses voisins. Un noeud qui reçoit un message *LeaveROSA* supprime de sa liste de voisins

le nœud ayant envoyer le message. Il modifie également sa liste de grumeaux afin qu'elle reflète ce changement.

L'autre méthode n'est appliquée que lorsqu'un nœud ne donne aucun signe de vie à ses voisins. Comme il est décrit dans la section à la Section 0.3.7, quand un nœud ne donne plus signe de vie à ses voisins, ces voisins suppriment progressivement ce nœud de leur liste de voisins. Normalement, la première méthode devrait être privilégiée car elle génère moins de bande passante et utilise moins de cycles de CPU. La seconde méthode devrait être utilisée que quand un nœud tombe en panne ou quand il est dans l'incapacité d'appliquer la première méthode.

0.3.3 Joindre un grumeau

Un nœud joint un grumeau signifie que ce noeud crée des liens virtuels avec les nœuds qui composent ce grumeau. Hors il n'est pas toujours possible de créer un lien virtuel entre deux noeuds. Dans ce cas, un noeud peut joindre partiellement un grumeau. Dans cette partie nous allons voir comment un noeud joint un grumeau.

Pour joindre un grumeau, un nœud doit connaître la représentation de ce grumeau. Une fois en possession de cette représentation, le nœud doit d'abord vérifier s'il peut créer des liens virtuels avec tous les nœuds qui composent le grumeau. Cela se fait avec l'aide de l'utilitaire ping (Muuss [1983]) sur les réseaux IP ou avec tout autre utilitaire qui effectue la même action sur les réseaux non-IP. Une fois la vérification terminée le nœud détermine la joinRatio. Le joinRatio est égal à 1 moins le ratio entre le nombre de liens virtuels que le noeud peut créer et le nombre de nœuds qui composent le grumeau. Si le joinRatio est égale à zéro, nous sommes dans le cas d'une pleine adhésion. Si elle excède la limite fixée par joinRationLimit il est alors impossible pour le nœud de joindre le grumeau. Et dans les autres cas, le nœud effectue une jonction partielle.

Après le calcul du joinRatio et s'il est possible pour ce nœud de joindre le grumeau, le noeud modifie la représentation du grumeau afin de refléter le fait que le noeud l'ait joint. Il ajoute son identifiant dans la liste des identifiants des nœuds et son adresse physique dans la liste des adresses physiques. Le noeud modifie également toutes les informations nécessaires pour calculer la densité du grumeau. Enfin, le noeud envoie par l'intermédiaire d'un message *UpdateLump*, la nouvelle représentation du grumeau à tous les nœuds qui le composent. Les nœuds qui reçoivent le message, mettent à jour leur liste de grumeaux avec l'aide de la représentation du grumeau contenu dans le message *UpdateLump*. La différence entre une jonction complète et une jonction partielle et réside dans le fait que des gestions de liens brisés survienent après une jonction partielle puisque lors d'une jonction partielle, il a été impossible d'établir des liens virtuels avec certains des nœuds du grumeau.

0.3.4 Quitter un grumeau

Quitter un grumeau consiste pour un nœud à modifier la représentation du grumeau que le nœud veut quiter. La représentation doit refléter le fait que le nœud ne compose plus le grumeau. Les modifications que le noeud doit accomplir sont les inverses de ceux que doit effectuer noeud quand il veut joindre un grumeau. Une fois la représentation du grumeau modifiée, le nœud l'envoie à l'aide d'un message *UpdateLump* à tous les nœuds qui composent le grumeau. Chaque nœud qui reçoit la nouvelle représentation met à jour

sa liste de grumeaux.

0.3.5 Scission d'un grumeau

La scission d'un grumeau peut se produire lors de deux occasions, quand un lien virtuel est cassé ou au cours de la limitation du nombre de noeuds par grumeaux. La scission d'un grumeau est organisée autour de deux des nœuds du grumeau. Il consiste à diviser le grumeau en vue d'obtenir deux autres grumeaux. Les nœuds qui composent le premier des deux grumeaux qui résultent d'une scission sont les nœuds du grumeau scindé moins un des deux nœuds autour de laquelle la scission est organisée. Les nøeuds qui composent le second grumeau résultant de la scission sont ceux du grumeau scindé moins l'autre noeud autour de laquelle la scission est organisée.

Quand un nœud veut scinder un grumeau autour de lui et l'un des autres nœuds qui composent le grumeau, il construit la représentation des deux grumeaux qui résultent de la scission. Ensuite, le nœud envoie ces représentations des grumeaux résultant de la scission aux nœuds concernés par celle-ci à l'aide de messages *SplitLump*. Ces messages contiennent, en plus de la représentation des grumeaux, l'identifiant de grumeau qui est scindé. Un nœud qui reçoit un message *SplitLump*, recherche dans sa liste de grumeaux celui qui possède l'identifiant contenu dans le message et remplace dans sa liste de grumeaux, le grumeau qui est scindé par les deux grumeaux contenus dans le message.

0.3.6 Boucle principale de ROSA

La boucle principale exécutée par un nœud de ROSA est répétée à chaque ńIntervalleż de temps. Ce paramètre doit être défini de façon expérimentale en fonction des capacités du réseau sur lequel ROSA est utilisé. La boucle principale de ROSA est composée de 5 fonctions: checkFailure, checkAbsorption, checkMemberLimit, checkLumpLimit et enhanceROSA. La première fonction a pour but de détecter la panne survenant sur les liens virtuels et à réagir à celle-ci. Le rôle de la fonction checkAbsorption est de détecter et de gérer l'absorption d'un grumeau de la liste de grumeaux par un autre grumeau de cette liste. Les objectifs des fonctions checkMemberLimit et checkLumpLimit sont de limiter le nombre de voisins par nœud. Le fonction enhanceROSA est responsable de l'optimisation de ROSA. Cette fonction décide dans quels cas le nœud doit quitter un grumeau de haute densité afin d'augmenter la densité d'un grumeau avec une faible densité.

0.3.7 Gestion des liens virtuels brisés

Un avantage majeur de ROSA est que son protocole permet de détecter rapidement les défaillances des liens virtuels entre les noeuds et de modifier la topologie du réseau afin de refléter de telles pannes. La gestion des liens virtuels brisés est réalisée par la fonction checkFailure. La gestion des liens brisés se déroule en deux étapes, la détection des pannes et la modification de topologie.

La première étape consiste pour chaque nœud à donner signe de vie à ses voisins en leur envoyant un message *Alive*. Si un lien entre deux nœuds est brisés, l'envoi de ces messages *Alive* est interrompu. Les messages *Alive* sont également utilisés pour diffuser les connaissances sur les grumeaux ayant une faible densité. Dans chaque message *Alive* envoyé par un noeud est encapsulé un grumeau. Ce grumeau est le grumeau de plus faible

densité dans sa liste des grumeaux. Quand un noeud reçoit un message *Alive*, il extrait le grumeau contenu dans le message et l'ajoute à sa liste de grumeaux connus. Ainsi, chaque noeud connaît le grumeau avec la plus faible densité à laquelle appartient chacun de ses voisins.

Une fois qu'un noeud détecte un lien brisé par l'absence de message *Alive* de la part de l'un de ses voisins, la deuxième étape consiste pour ce nœud à modifier sa liste de grumeaux de façon à prendre en compte la panne sur ce lien. Quand un nœud détecte que la lien virtuel entre lui et un voisin est brisé, il recherche dans sa liste de grumeaux ceux qui contiennent le voisin défaillant. Quand un nœud rencontre un tel grumeau, il le scinde autour de lui et de son voisin. La scission d'un grumeau est décrite dans la Section 0.3.5.

0.3.8 Limitation du nombre de voisins par nœuds

Pour que ROSA soit extensible, le nombre de voisins par nœud est borné. Plutôt que de fixer une taille maximale sur la liste des voisins et d'imposer à un nœud de se séparer d'un de ses voisins lorsque la capacité de la liste des voisins est atteint, c'est en imposant un nombre maximum de noeuds par grumeaux et un nombre maximal de grumeaux par nœuds que le protocole de ROSA limite le nombre de voisins. La limitation du nombre de nœuds par grumeaux est faîte par la fonction checkNpL et la limitation du nombre des grumeaux par nœuds est faîte par la fonction checkLpN.

Pour limiter le nombre de nœuds qui composent un grumeau, un nœud cherche dans sa liste de grumeaux ceux dont le nombre de nœuds dépasse la limite. Quand un tel grumeau est trouvé, le nœud peut quitter ce grumeau ou bien le scinder . Un nœud peut quitter un grumeau qui a atteint le nombre maximal de nœud si le nœud appartient à au moins un autre grumeau et si le fait de quitter le grumeau n'entraine pas une perte de densité. Si une de ces conditions n'est pas remplie, le nœud doit sélectionner une paire de nœuds dans le but de scinder le grumeau autour de ces deux noeuds. Cette sélection se fait comme suit, le nœud parcourt la liste des nœuds qui composent le grumeau, pour chaque paire d'éléments de cette liste, le nœud calcule ce que serait la densité des deux grumeaux qui résulteraient de la scission du grumeau si il était scindé autour de ces éléments. Enfin, le noeud choisit la paire qui maximise la densité de grumeaux qui résulteront de la scission.

La limitation du nombre de noeuds par grumeaux est la première des deux étapes de la limitation du nombre de voisins par nœuds. La deuxième étape consiste à limiter le nombre de grumeaux auxquels un noeud peut appartenir. Pour limiter le nombre de grumeaux auxquels il appartient, un nœud regarde la taille de sa liste de grumeaux. Si cette liste excède la taille limite, le nœud sélectionne un grumeau à quitter. Ce choix est fondé sur la densité hypothétique qu'aurait le grumeau si le noeud le quittait. Le nœud sélectionne le grumeau qui aura la plus grande densité théorique et le quitte.

0.3.9 Gestion des absorptions des grumeaux

Lorsque l'ensemble des noeuds qui composent un grumeau contient tous les nœuds qui composent un autre grumeau, il se produit une absorption de second grumeau par le premier. Il en résulte la disparition du grumeau absordé des listes de grumeaux de tous les noeuds concernés. La gestion de l'absorption ne peut se faire sans l'échange de messages entre les nœuds qui composent le grumeau absorbé. Puisque l'ensemble des noeuds qui

composent un grumeau peut être inclus dans les ensembles de nœuds de plusieurs grumeaux différents, plusieurs grumeaux sont susceptibles d'absorber un même grumeau. Ce choix doit être la même pour tous les nœuds qui composent le grumeau absorbé.

La détection et la gestion des absorptions de grumeaux se fait par les nœuds à l'aide de la fonction checkAbsoprtion. Un noeud détecte l'absorption d'un grumeau en parcourant sa liste de grumeaux. Pour chaque grumeau de cette liste, il vérifie s'il y a un autre grumeau de la liste qui peut l'absorber. Si un tel couple de grumeaux est trouvé, le noeud modifie sa liste des grumeaux afin de refléter l'absorption et avertis les autres nœuds de ces grumeaux de propos de cette absorption. Ceci est fait à l'aide de messages *AbsorbLump*. Ces messages contiennent l'identifiant du grumeau absorbé et l'identifiant du grumeau absorbant. Un noeud qui reçoit un message *AbsorbLump* supprime de sa liste de grumeaux le grumeau correspondant référencé par l'identifiant du grumeau absorbé.

0.3.10 Augmentation de la densité globale

Nous avons vu dans la Section 0.3.7 traîtant de la gestion des liens brisés que chaque noeud reçoit périodiquement des messages *Alive* de ses voisins. Ces messages contiennent des grumeaux avec des densités faibles. Nous allons voir dans cette section comment, à partir de cette liste de la grumeaux reçus, chaque nœud augmente localement la densité de ROSA.

La première étape consiste pour un nœud à parcourir la liste des grumeaux reçus de ses voisins. Pour chacun de ces grumeaux, le nœud compare la densité actuelle de ce grumeau à la densité hypothétique que le grumeau aura si le nœud le joint. Le calcul de cette densité est effectué avec la fonction getRMDJ. Cette fonction effectue une copie de la représentation du grumeau et simule les changements causés par la jonction du nœud. Dans le cas où le grumeau a atteint le nombre maximum de noeuds, la fonction getRMDJ simule également la scission résultant de la jonction. Ensuite, la fonction renvoie la densité hypothétique du grumeau obtenue lors de la simulation. Après avoir comparé les densités actuelles des grumeaux reçus à leurs densités hypothétiques, le nœud retire de cette liste chacun des grumeaux dont la densité hypothétique est égale ou inférieure à sa densité actuelle. Le noeud possède désormais une liste des grumeaux dont il peut augmenter la densité. Si cette liste est vide, le processus s'arrête, sinon il se poursuit.

Considérons que la liste des grumeaux dont la densité peut être augmentée par le noeud n'est pas vide. Le noeud essaye de joindre le grumeau de cette liste qui a la plus faible densité. Si le nombre maximal de grumeaux auxquels un noeud peut appartenir n'est pas atteint, le nœud joint le grumeau. Dans le cas où la limite de grumeaux est atteinte, le nœud doit quitter un grumeau s'il veut en joindre un nouveau. Cela signifie que, pour augmenter la densité d'un grumeau, le noeud peut diminuer la densité d'un autre grumeau. Par conséquent, avant de joindre le grumeau le nœud s'assure que la jonction augmentera effectivement la densité locale du nœud et ne la diminuera pas. Le noeud parcours donc sa liste de grumeaux, et pour chacun de ces grumeaux il compare la densité à la densité hypothétique que le grumeau aura si le nœud le quitte. Ces comparaisons sont effectuées par la fonction getRMDL. La fonction getRMDL, d'une façon similaire à la fonction getRMDJ, simule les grumeaux obtenus si le nœud quitte un grumeau. Le nœud peut joindre le grumeau et accroître sa densité que si la densité actuelle de ce grumeau est inférieure à la densité hypothétique qu'aurait le grumeau que le nœud aurait à quitter.

Une table de hashage distribuée pour ROSA

0.4 La chaîne de grumeaux

Dans ROSA, il est possible d'organiser l'enchevêtrement de grumeaux en une DHT. Cette DHT est appelée la 'chaîne de grumeaux'. La 'chaîne de grumeaux' évolue en fonction des modifications qui se produisent dans ROSA. Dans cette section, nous allons d'abord décrire la 'chaîne de grumeaux'. En second, nous allons voir comment les modifications de l'ensemble des grumeaux affectent la 'chaîne de grumeaux' et la façon dont les nœuds réagissent aux changements afin de maintenir la 'chaîne de grumeaux'. Pour terminer nous allons montrer comment la 'chaîne de grumeaux' peut être utilisée pour acheminer les paquets de données d'un noeud à un grumeau.

0.4.1 Description

La 'chaîne de grumeaux' est une DHT construite sur ROSA. L'espace des clés de cette DHT est un intervalle entier appelé I_{init}. I_{init} et définie comme: $I_{init} = [0, 2^{128} - 1]$. Chaque grumeau de ROSA gère une partie de cet espace de clé et I_{init} est projeté sur tous l'enchevêtrement des grumeaux afin de former une chaîne. La projection est réalisée en partitionant I_{init} en sous-intervalles. Ces sous-intervalles sont attribués aux grumeaux afin de satisfaire les conditions suivantes:

- Tous les sous-intervalles sont attribués (1) ;
- Deux grumeaux qui possèdent des sous-intervalles contigus partagent au moins un nœud (2);
- Chaque grumeau gère au moins un sous-intervalle (3).

Afin d'être effective la 'chaîne de grumeaux' doit impérativement satisfaire les conditions (1) et (2) à tout moment. La condition (3) assure seulement un bon rendement et un équilibrage de charge des grumeaux.

Considérons un grumeau l qui gère un sous-intervalle I de I_{init}. Le grumeau l est appelé le grumeau propriétaire de I. Le grumeau qui gère le sous-intervalle juste avant est appelé le prédécesseur de l. Un grumeau et son successeur sont des grumeaux consécutifs. Le grumeau qui possède le sous-intervalle juste après I est appelé le successeur l. Un lump a autant de prédécesseurs et successeurs comme de sous-intervalles qu'il gère. Afin d'être

plus efficaces le premier et dernier grumeau de la chaîne sont respectivement prédécesseur et successeur l'un de l'autre. De cette façon, les grumeaux forment une chaîne circulaire. Comme un noeud peut appartenir à plusieurs grumeaux, il existe des raccourcis dans la 'chaîne de grumeaux', c'est à dire une manière de passer d'un grumeau à l'autre sans passer par les grumeaux intermédiaires qui sont situés entre eux dans la 'chaîne de grumeaux'. La figure 2.1 montre la projection de l'enchevêtrement de blocs de I_{init}. Sur cette figure, les nœuds sont représentés par les cercles verts, les grumeaux par des formes en pointillés violets et les raccourcis par des courbes rouges.

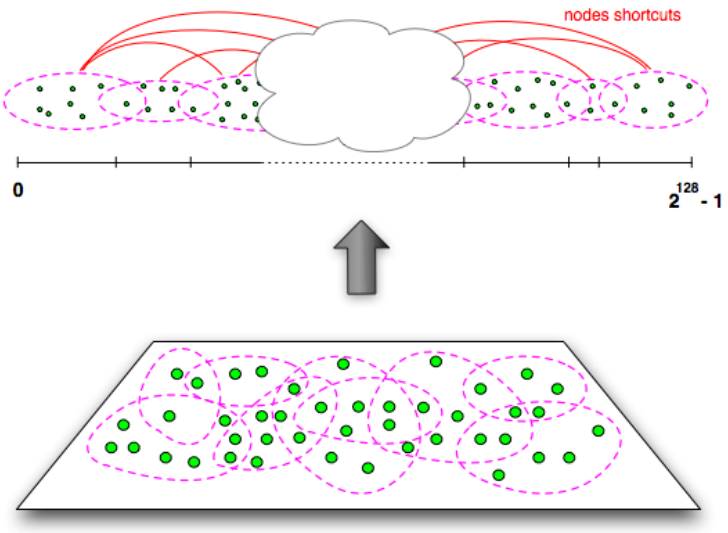


Figure 1: The transformation of the entanglement of lumps into a 'chain of lumps'.

Il est nécessaire d'ajouter la liste des sous-intervalles gérés par un grumeau à la représentation de celui-ci. Nous avons également à présenter la représentation d'un sous-intervalle. La représentation d'un sous-intervalle est composée de la borne inférieure et la borne supérieure de ce sous-intervalle. Cette représentation est complétée par l'identifiant du grumeau propriétaire, les listes des noeuds qui composent le prédécesseur et le successeur du lump propriétaire et une liste des entrées de DHT. Une entrée de DHT est une valeur qu'un nœud a stocké sur la 'chaîne de grumeaux' ainsi que la clé qui lui est associée.

Dans le reste de ce chapitre, le sous-intervalle qui précède le sous-intervalle I est appelé le prédecesseur de I et le sous-intervalle succède I s'appelle le successeur de I. Le prédécesseur et le successeur de I sont respectivement notés $I-$ et $I+$. Le grumeau qui gère un sous-intervalle I est noté l_I. Nous appellerons, le grumeau prédécesseur et le grumeau successeur d'un sous-intervalle I, respectivement, le grumeau gérant $I-$ et le grumeau gérant $I+$ En conséquence, en ce qui concerne un sous-intervalle I, nous noterons le grumeau prédécesseur de l_I l_{I-} et le grumeau successeur de l_I, l_{I+}. La liste des identifiants des nœuds qui composent l_{I-}, sera nommée la liste des prédécesseurs et la liste des identifiants des nœuds qui composent l_{I+} sera nommée dans la liste des successeurs.

0.4.2 Construction et maintenance de la 'chaîne de grumeaux'

0.4.2.1 Initialisation de la 'chaîne de grumeaux'

Lors de l'initialisation de ROSA, le premier nœud crée le premier grumeau. Le premier grumeau doit gérer l'intervalle initial I_{init}. Comme il est expliqué dans la suite de ce document les sous-intervalles gérés par les grumeaux peuvent être scindés lors de la scission des grumeaux, mais ni les absorptions, ni les scissions des grumeaux ne causent la perte de sous-intervalles. Par conséquent, cela implique que la condition (1) est satisfaitt. L'initialisation de la 'chaîne de grumeaux' contraint à modifier la fonction initROSA().

Ces modifications consistent à confier I_{init} au premier grumeau. Comme il n'existe, pour l'instant qu'un unique grumeau, selon la définition de la 'chaîne du grumeau' ce premier grumeau est son propre prédécesseur et successeur. C'est pourquoi les listes des identifiants des nœuds qui composent le prédécesseur et le successeur est égal à la liste des identifiants des nœuds du premier grumeau. Pour finir le nœud qui initie ROSA ajoute la représentation de I_{init} à la liste des sous-intervalles gérés par le premier grumeau.

0.4.2.2 Réaction aux absorptions de grumeaux

Quand un grumeau est absorbé par un autre, les sous-intervalles gérés par le grumeau absorbé doivent être confiés au grumeau absorbant. Ceci est schématisé dans la Figure 3.7. Dans la partie supérieure de la figure on peut voir deux grumeaux, l_{I-} et l_I, ainsi qu'une partie de 'la chaîne de grumeaux'. Le grumeau l_{I-} gère le sous-intervalle $I- = [a, b)$ et le grumeau l_I gère le sous-intervalle $I = [b, c)$. Le nœud n joint le grumeau l_I. Par conséquent l'ensemble des nœuds qui composent l_I comprend l'ensemble des noeuds qui composent l_{I-}, et le grumeau l_I absorbe le grumeau l_{I-}. Dans la partie inférieure de la figure on peut voir que le sous-intervalle gérés par l_{I-} est donnée au grumeau l_I. Après la jonction, le grumeau doit gérer le sous-intervalle $[a, c)$.

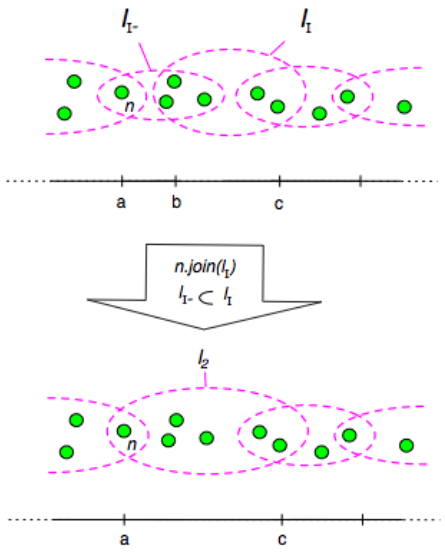


Figure 2: Une Absorption et son impact sur la 'chaîne de grumeaux'

La détection et la gestion des absorptions des grumeaux sont traitées dans la Section 0.3.9. Nous avons vu que la détection consiste en la vérifier si l'un des grumeaux de sa liste de grumeaux d'un nœud peut en absorber un autre. Si une telle paire de grumeaux est trouvée le nœud envoie des messages *AbsorbLump* aux nœuds qui composent la grumeau absorbant. La 'chaîne de grumeaux' oblige à modifier le contenu des messages *AbsorbLump* et la façon dont les nœuds traîtent ces messages.

0.4.2.3 Reaction aux scissions de grumeaux

Quand un grumeau est scindé en deux nouveaux grumeaux. Les sous-intervalles gérés par le grumeau scindé doivent être gérés par les deux grumeaux résultant de la scission. Certains de ces sous-intervalles doivent être scindés et distribués aux grumeaux qui résultent de la scission et certains doivent être confiés (sans être scindés) à un des grumeaux qui résulte de la scission.

Un grumeau l possède un prédecesseur et un successeur pour chaque sous-intevalle $I = [a, b)$ qu'il gère. Soit l_{I-} le grumeau prédecesseur et l_{I+} le grumeau successeur de l correspondant à I. Soit s_I, s_{I-} et s_{I+} l'ensemble des nœuds qui composent les grumeaux l_I, l_{I-} and l_{I+}. Si nous nous référons à la définition de la 'chaîne de grumeaux' nous avons $s_{I-} \cap s_I \neq \emptyset$ and $s_I \cap s_{I+} \neq \emptyset$. Cette propriété est vraie lors de la création de la 'chaîne de grumeaux'. De la scission du grumeau l_I résultent deux grumeau l_1 et l_2. L'ensemble des nœds de l_1 est s_1 et celui de l_2 est s_2. Si la taille de l_I est plus grande que 2 alors $s_1 \cap s_2 \neq \emptyset$. Le sous-intervalle $I = [a, b)$ peut être scindé en deux nouveaux sous-intervalles $I' = [a, \lfloor \frac{a+b}{2} \rfloor)$ et $I'+ = [\lfloor \frac{a+b}{2} \rfloor, b)$ si une de ces deux conditions:

$$s_{I-} \cap s_1 \neq \emptyset \text{ and } s_{I+} \cap s_2 \neq \emptyset \ (4)$$

$$s_{I-} \cap s_2 \neq \emptyset \text{ and } s_{I+} \cap s_1 \neq \emptyset \ (5)$$

est respectée.

Il peut ariver qu'aucune des conditions (4, 5) soit satisfaite. Dans ce cas I ne peut être scindé. Cependant, puisque $s_{I-} \cap s_I \neq \emptyset$ et $s_I \cap s_{I+} \neq \emptyset$ cela signifie que soit:

$$s_{I-} \cap s_1 \neq \emptyset \text{ and } s_{I+} \cap s_1 \neq \emptyset \ (6)$$

ou

$$s_{I-} \cap s_2 \neq \emptyset \text{ and } s_{I+} \cap s_2 \neq \emptyset \ (7)$$

Dans le premier cas I doit être confié à l_1 et sinon confié à l_2. Si une des conditions (4, 5) est satisfaite, I' et $I'+$ doivent être confiés à l_1 et l_2. Cela est schématisé dans la Figure 3.9.

Dans la partie supérieure de cette figure, la nœud n_1 détecte que le lien virtuel avec le nœud n_2 est brisé. Le grumeau l_I qui est composée de n_1, n_2 et d'autres nœuds doit être scindé autour n_1 et n_2. Il résulte de cette scission deux grumeaux l_1 et l_2. Dans la partie inférieure de la figure le sous-intervalle $[a, b)$ est scindé en deux sous-intervalles et celui de gauche est donné à l_1 tandis que la droite est donné à l_2.

0.4.3 Utilisation de la 'chaîne de grumeaux'

0.4.3.1 Stockage d'une paire <clé, valeur>

Dans cette section, nous allons voir comment un nœud peut stocker une paire <clé, valeur> sur la 'chaîne de grumeaux'. Pour stocker une valeur sur la 'chaîne de grumeaux', un nœud

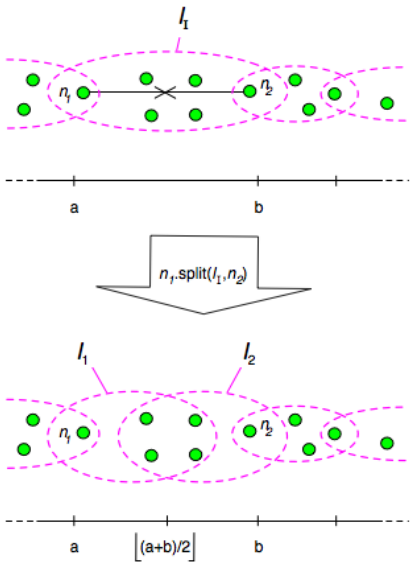


Figure 3: A split and its impact on the 'chain of lumps'

doit d'abord calculer la clé associée à la valeur. Cette clé est un entier qui doit appartenir à $[0, 2^{128} - 1)$. Afin de calculer cette clé, nous pouvons utiliser une fonction de hachage existantes sur les octets de la valeur. La clé est finalisé en effectuant un modulo 2^{128} sur la résultat donné par la fonction de hachage. Une fois que cette clé est calculée, le nœud peut démarrer le processus de stockage.

La première étape du processus de stockage par un nœud n consiste à savoir si le nœud gère la clé. Si le nœud gère cette clé, le nœud construit un message *StoreHere*. Soit I_{key} le sous-intevralle gérant la clé et l_{Ikey} le grumeau gérant I_{key}. Le message *StoreHere* contient la paire <clé, valeur> et la borne inférieure de I_{key}. Le nœud n envoie ce message à l_{ikey}. Un nœud qui reçoit un message *StoreHere*, cherche le sous-intervalle dont la borne inférieure est égale à celle contenue dans le message. Une fois ce sous-intervalle trouve, le nœud complète la liste des paires <clé, valeurs> de I_{key} avec la paire <clé, valeur> contenue dans le message.

Si le nœud n qui veut stocker la paire <clé, valeur> ne gère pas la clé, il envoie un message *GetDistance* à ses voisins. Ce message contient uniquement la clé. Un nœud qui reçoit un tel message, récupère la clé contenue dans le message et calcule la distance entre lui-même et la clé. La distance entre un nœud n et une clé k est définie comme la distance minimale entre la clé et les grumeaux de sa liste de grumeaux, la distance entre un grumeau l et d'une clé k est la distance minimale entre k et les sous-intervalles gérés par l et la distance entre un sous-intervalle I et la clé k est défini comme suit:

$$dist(I,k) = \begin{cases} min(|k - I.lowBound|,\ |k - I.upBound|) & \text{if} \quad k \notin I \\ 0 & \text{if} \quad k \in I \end{cases}$$

Une fois la distance calculée, un nœud qui a reçu un message *GetDistance* répond par un message *Distance*. Ce message contient la distance calculée. Le nœud n qui envoie les messages *GetDistance* reçoit éventuellement de ses voisins un ensemble de messages

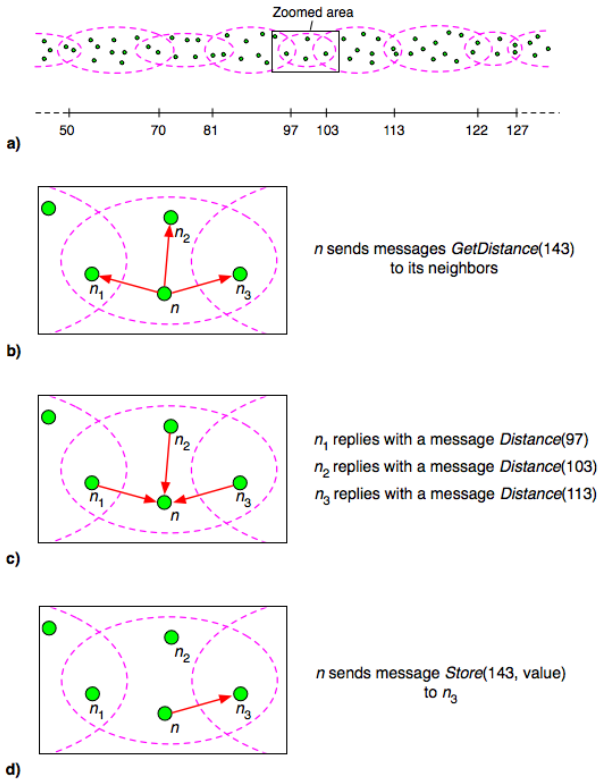


Figure 4: A step of the store process

Distance. Alors n sélectionne le voisin qui a envoyé le message qui contient la plus petite distance et construit un message *Store*. Ce message contient la paire <clé, valeur> que le nœud souhaite stocker. Enfin n envoie ce message au voisin sélectionné. Un nœud qui reçoit un message *Store* répète le processus et ce processus sera répété jusqu'à ce que la paire <clé, valeur> atteigne le grumeau appropriée. La Figure 3.12 montre une étape de ce processus.

0.4.3.2 Récupération d'une valeur

Pour récupérer une valeur stockée sur la 'chaîne de grumeaux' un nœud n doii connaître la clé k qui correspond à la valeur. Si n connait k, il peut démarrer le processus de recherche. D'abord n vérifie si il gère k. Si oui, le nœud peut récupérer la valeur désirée sur le grumeau qui gère k. Sinon le nœud construit et envoie un message *Lookup*. Ce message contient la clé k, l'identifiant et l'emplacement de n. L'emplacement d'un nœud est constitué par son identifiant et les bornes des sous-intervalles gérés par les grumeaux auxquels le nœud appartient.

Un nœud qui reçoit un message *Lookup* vérifie s'il gère k. Si le nœud gère cette clé, il récupère la valeur qui correspond à k, et construit un message *Datapacket* contenant la valeur désirée. Ensuite, le nœud envoie le message au nœud n. Afin d'être en mesure d'envoyer le message, le nœud doit trouver une clé qui est géré par n, avec l'aide de l'emplacement de n qui est contenu dans le message *Lookup*. Une fois qu'une clé est

trouvée, le nœud envoie la valeur récupérée à n comme dans le procédé décrit dans la Section 0.4.3.3.

Si un nœud qui reçoit un message *Lookup* ne gère pas la clé k, il envoie un message *GetDistance* à ses voisins. Ensuite, le nœud sélectionne le voisin qui a répondu avec la plus petite distance, et lui transmet le message *Lookup*. Ce processus sera répété jusqu'à ce que le message *Lookup* atteint un nœud qui gère la clé k.

0.4.3.3 Envoi d'un paquet de donnée à un grumeau donné

Il est possible d'étendre le processus de stockage afin de permettre à un nœud d'envoyer des paquets de données vers le grumeau gérant une clé donnée. Soit un nœud n qui veut envoyer des paquets de données aux nœuds du grumeau qui gère une clé k donnée. Le nœud n construit un message *Datapacket* message. Ce message contient la clé k et les données à transmettre. Ensuite, le nœud vérifie qu'il n'appartient pas au grumeau qui gère k. Si n appartient à un tel grumeau, il envoie le message *Datapacket* à tous les nœuds qui composent le grumeau. Si le nœud n'appartient pas à un tel grumeau, il envoie des messages *GetDistance* à ses voisins, reçoit un ensemble de messages *Distance*, sélectionne le voisin qui est le plus proche de k et transmet le message *Datapacket* au voisin selectioné.

Quand un nœud reçoit un message *Datapacket* il vérifie s'il gère la clé k contenue dans le message. Si le nœud gère cette clé, il extrait d'abord les données contenues dans le message. Ensuite, il vérifie si le nœud qui envoie le message gère également k. Cette vérification consiste à regarder si le nœud expéditeur est l'un des nœuds qui composent le grumeau qui gère k. Si le nœud expéditeur ne gère pas k, le nœud qui a reçu le message *Datapacket* le transmet à tous les nœuds du grumeau qui gère k. Si le nœud qui a reçu le message *Datapacket* ne gère pas de k, le nœud répète le procédé mentionné ci-dessus et ce processus est répété jusqu'à ce que le message *Datapacket* atteigne un nœud qui gère k.

Services sur ROSA

0.5 Un système résilient de stockage de fichiers

ROSA doté de la 'chaîne de grumeaux' offre un réseau et une table de hachage distribuée résistants. Ce système de stockage possède déjà les avantages de la résilience de la 'chaîne de grumeaux' et afin d'assurer une meilleure tolérance aux pannes chaque fichier stocké sur ROSA est stocké plusieurs fois. Chacune des répliques d'un fichier est stockée par un nœud. Le nombre de répliques est un paramètre qui doit être décidé lorsque le fichier est stocké. De cette façon, les fichiers importants peuvent être stockés avec de nombreuses répliques tandis que les fichiers les plus négligeables ne seront stockés qu'avec seulement quelques répliques. Afin de permettre à un nœud de savoir où les répliques d'un fichier donné sont situées, pour chaque fichier un index est également stocké sur ROSA. L'index d'un fichier contient l'identificateur de fichier, certaines données facultatives sur le fichier, un drapeau, les listes de l'emplacement des répliques et la liste des identifiants des nœuds qui est propriétaire de ce fichier.

0.5.1 Stockage d'un fichier

Un nœud qui souhaite stocker un fichier sur ROSA calcule l'identifiant du fichier. Cet identifiant est obtenu en utilisant une fonction de hachage sur le nom du fichier. Ensuite, le nœud détermine le nombre de répliques nécessaires et les nœuds qui sont autorisés à accéder au fichier. Pour terminer le nœud doit construire un message *StrFile*. Ce message contient l'identifiant de nœud d'envoi et son emplacement, l'identificateur de fichier, une description facultative du fichier, le nombre de répliques voulues, la liste des propriétaires et le fichier lui-même.

Une fois le message construit, le nœud l'envoie par la DHT au grumeau index de ce fichier. Le grumeau index de ce fichier est le grumeau qui gère le sous-intervalle qui contient l'identifiant du fichier. Quand un nœud du grumeau index reçoit un message *StrFile* il construit et envoie un message à *StrIdx* aux autres nœuds du grumeau index. Ces nœuds stockent l'index de ce fichier. Le message *StrIdx* contient l'index de ce fichier.

Le nœud qui reçoit le message *StrFile* sélectionne aléatoirement N éléments de I_{init}, où N est le nombre de répliques voulu. Nous désignerons ces éléments $k_i, i \in [1, N]$. Ensuite, le nœud construit un message *ScttrRep* qui contient l'identificateur de fichier et le fichier lui-même. Pour conclure, le nœud envoie une copie du message *ScttrRep* à chacun des grumeaux qui gèrent les k_i. Ces messages sont également envoyés en utilisant le procédé décrit dans la Section 0.4.3.3.

Ce processus garantit que les messages *ScttrRep* atteignent des nœuds ordinaires de

l'un des grumeaux qui gère l'un des k_i. Lorsque l'un de ces nœuds reçoit un tel message, il choisit au hasard un nœud du grumeau qui gère le k_i et envoie un message *StrRep* à ce nœud. Ce message informe le nœud de réception qu'il doit stocker une réplique du fichier. Un message *StrRep* contient l'identifiant et une copie du fichier.

0.5.2 Récupération d'un fichier

Un nœud qui veut récupérer un fichier doit connaître l'identifiant de fichier. Afin de récupérer un fichier, le nœud doit envoyer un message *RtrvFile* au grumeau du fichier en utilisant le procédé décrit dans la Section 0.4.3.3. Un message *RtrvFile* contient l'identifiant du fichier à récupérer, l'identifiant et l'emplacement du nœud qui veut récupérer le fichier. Quand un nœud du grumeau index reçoit un tel message, il répond avec un message *RepLoc*. Un message *RepLoc* doit contenir l'emplacement des répliques et la version actuelle du fichier.

Une fois que le nœud qui veut récupérer le fichier connaît l'emplacement des nœuds stockant les répliques, il sélectionne celui qui est le plus proche (dans le sens de la 'chaîne de grumeaux') et envoie un message *RtrvRep*. Ce message contient l'identifiant du fichier à récupérer. Un nœud qui reçoit un tel message doit répondre par un message *HereItIs* qui contient la réplique du fichier voulu.

0.5.3 Mettre à jour, modifier, effacer un fichier et quelques autres choses

Ce système résilient de stockage de fichiers permet aux utilisateurs de mettre à jour, modifier un fichier. Il est capable de gérer (de façon peu efficace) les modifications concurrentes d'un même fichier. Ce système permet aussi d'effacer un fichier. Il dispose d'un mécanisme permettant le remplacement d'un nœud réplique par un autre en cas de panne. Il dispose également d'un système de droit qui donne l'accès au fichier seulement aux nœuds autorisés.

0.6 Routage résilient de nœud à nœud

0.6.1 Construction et maintenance des tables de routage

Comme décrit dans le Chapitre 0.3.10, la 'chaîne de grumeaux' permet à n'importe quel nœud de ROSA d'envoyer des paquets de données vers les nœuds composant un grumeau gérant une clé donnée. La 'chaîne de grumeaux' peut être directement utilisée dans le cas où certains services de réseau sont confiés à des grumeaux. Les nœuds peuvent avoir accès à ces services en envoyant des requêtes aux grumeaux concernés. Quand un nœud veut communiquer avec le nœud qui possède un identificateur donné, il doit utiliser le service de routage. Ce service de routage utilise la 'chaîne de grumeaux' et se base sur des tables de routage. Ces tables de routage sont stockées par les gros grumeaux et sont accessibles via la chaîne de grumeaux.

Chaque nœud de ROSA qui veut recevoir des paquets de données d'autres nœuds doit bâtir et maintenir sa table de routage. La table de routage d'un nœud est composée de son identifiant, son emplacement et un indicateur. Si l'indicateur est défini à TRUE cela signifie que le nœud gère une clé de la 'chaîne de grumeaux'. Si l'indicateur est défini à FALSE, cela signifie que le nœud ne gère pas de clé.

Quand un nœud se connecte à ROSA, il doit créer sa table de routage. Une fois construite, le nœud l'inclut dans un message *StrTable*. Le nœud envoie ce message, en utilisant la procédure décrite dans la Section 0.4.3.3, aux nœuds qui composent le grumeau qui gère la clé correspondant à l'identifiant de nœud. Le nœud garde aussi une copie de sa table de routage actuelle. Quand un nœud du grumeau concerné reçoit le message *StrTable* il extrait de la table du message et la stocke. Par conséquent, chacun des nœuds qui compose ce grumeau stocke la table. Un nœud qui veut recevoir des paquets de données doit maintenir sa table à jour.

0.6.2 Routage d'un message à un nœud

Soit id_1 l'identifiant d'un nœud n_1 de ROSA. Quand un nœud n_2 avec l'identifiant id_2 veut envoyer un paquet de données vers au nœud n_1, il encapsule le paquet de données dans un message *SendToTable*. Ce message, en plus de paquet de données contient les identifiants du nœud d'envoi id_2 et du nœud cible id_1. Ensuite, le nœud n_2 envoie le message *SendToTable* au grumeau qui gère la clé correspondant à id_1.

Les nœuds qui composent ce grumeau stockent la table de routage de n_1, si n_1 a voulu le publier. Quand un nœud qui compose le grumeau censé stocker la table de n_1 reçoit le message *SendToTable* elle vérifie que le nœud n1 a publié sa table de routage. Si le nœud a publié sa table de routage, le nœud qui a reçu le message *SendToTable* regarde la liste des emplacements figurant dans le table de routage et détermine celle qui est la plus proche en terme de 'chaîne de grumeaux'.

Une fois cet emplacement déterminé, le nœud qui a reçu le message *SendToTable*, extrait une clé de cet emplacement et envoie un message *SendToNode* au grumeau gérant cette clé. Le nœud n_1 est dans ce grumeau, car les emplacements qui sont dans la table de routage du n_1 correspondent aux bornes des sous-intervalles gérés par les grumeaux que n_1 compose. Le message *SendToNode* contient le paquet de données, les identifiants du nœud d'envoi id_2 et du nœud cible id_1.

Dès qu'un nœud du grumeau ciblé reçoit le message *SendToNode*, il l'envoie au nœud n_1. Puisque tous les nœuds qui composent un grumeau sont des voisins, la dernière étape n'est pas un problème sauf si la table n'est pas à jour ou si n_1 ne gère pas les clés. Ceci ne sera pas décrit dans ce document.

Densité

0.7 Densité

Nous avons vu dans le Chapitre 0, que la gestion de la topologie de la ROSA consiste en un calcul par nœuds de la densité des grumeaux auxquels ils appartiennent. Chaque nœud envoie à ses voisins de la représentation du grumeau qui a la plus faible densité dans sa liste de grumeaux. Par conséquent, tous les nœuds de ROSA ont connaissance de quelques grumeaux qui ont de faibles densités. Selon cette connaissance sur les densités des grumeaux, les nœuds quittent les grumeaux avec de fortes densités afin de joindre et d'augmenter la densité des grumeaux avec des densités faibles.

La densité est le paramètre qui définit le comportement de ROSA. Si deux instances de ROSA sont déployées sur le même réseau mais avec deux définitions différentes de la densité. Les nœuds de chaque instance qui choisissent les grumeaux à quitter et à joindre selon la définition de la densité de l'instance. Par conséquent, l'ensemble des grumeaux de ces deux instances de ROSA sera complètement différent. La densité est le paramètre le plus important de ROSA, la densité d'une instance de ROSA doit être choisie pour tenir compte de l'objectif que ROSA est censé atteindre.

0.7.1 Densité par défaut

La densité par défaut d'un grumeau est égale à la taille du grumeau. C'est la plus simple des densités. Sa première utilité est de tester la manière dont le processus de gestion de la topologie de ROSA se comporte. Néanmoins, cette densité peut être utilisée pour la maximiser la moyenne du nombre de nœuds par grumeau. Cette densité a trois avantages:

- Elle ne nécessite pas de données supplémentaires pour permettre aux nœuds de le calculer ;
- Elle est facilement calculable ;
- Elle peut être utilisée par ROSA sur tous types de réseaux.

Nous avons effectuer quelques simulations qui confirment que cette densité peut être utilisée pour maximiser le nombre de nœuds par grumeau et donc le nombre de voisins par nœud (le nombre maximal de voisin par nœud est égal au nombre maximal de nœuds par grumeaux multiplié par la nombre maximal de grumeaux par nœud).

0.7.2 Densité résiliente

Le but de la densité résiliente est de rendre ROSA extrêmement résistant aux pannes. La densité est définie comme le nombre minimal de pannes sur les éléments des liens virtuels du grumeau qui sont nécessaires pour isoler un nœud du grumeau. De telle manière que si le nombre de pannes est inférieur à la densité, nous pouvons affirmer qu'il existe un chemin entre deux nœuds du grumeau et que les nœuds sont encore en mesure de communiquer entre eux. On peut remarquer que si tous les liens entre les nœuds d'un capital sont disjoints, la densité de la masse est simplement égale au cardinal du grumeau moins un, à savoir #l - 1.

Un nœud qui veut calculer la densité d'un grumeau doit construire le graphe du réseau recouvert correspondant à l'aide d'informations sur la topologie des liens virtuels du grumeau. Si l'on considère que les sommets et les arêtes peuvent tomber en pannes, pour calculer la densité de l'un des grumeaux un nœud doit d'abord calculer le st-cutsets minimal séparant chaque paire de nœuds du grumeau. La densité est égale au cardinal du plus petit des cutsets calculés. Le st-cutset minimal séparant deux sommets source et terminal, est l'ensemble minimal des sommets et des arêtes dont la suppression déconnecte la source du terminal.

0.7.3 Densité mobile

La densité mobile a été créée en vue d'adapter ROSA à un réseau mobile AdHoc dense. La densité mobile est conçu pour examiner les caractéristiques de nœuds mobiles. Ces caractéristiques des nœuds mobiles sont leur position relative, la vitesse et la direction.

Pour proposer une définition de cette densité adaptée aux réseaux AdHoc dense réseau, nous introduisons le concept de viabilité du lien entre deux nœuds. La viabilité du lien entre deux nœuds mobiles est une mesure qui fournit des informations sur la qualité et la durabilité de la connexion entre ces deux nœuds. Par exemple, deux nœuds mobiles allant dans des directions opposées ne seront pas à portée de communication aussi longtemps que deux nœuds proches suivant la même trajectoire.

La viabilité d'un lien entre deux nœuds mobiles n_1 et n_2 est calculable avec la formule suivante:

$$viability(n_1, n_2) = 1 - \left(\frac{1 - cos(\widehat{\vec{s_1}, \vec{s_2}})}{2} + \frac{d^2}{d_{max}^2} + \frac{||\vec{s_1} - \vec{s_2}||^2}{(2.v_{max})^2} \right) / 3$$

Où $\vec{s_1}$ et $\vec{s_2}$ sont respectivement les vecteurs d'état représentant la direction, la vitesse des nœuds n_1 et n_2. La distance entre deux nœuds est notée d et d_{max} represente la distance au delà de laquelle deux nœuds ne peuvent communiquer et v_{max} represente la vitesse maximale qu'un nœud peut atteindre. L'utilisation de la densité mobile implique que les nœuds soient équipés d'un système de GSP embarqué.

La densité d'un lump est la somme des viabilités des liens entres les nœuds qui composent le grumeau. Soit l un grumeau, sa densité est égale à:

$$density(l) = \sum_{a_1}^{a_1 \in l} \sum_{a_2}^{a_2 \in l - \{a_1\}} viability(a_1, a_2)$$

Conclusion

0.8 ROSA dans un cas réel

0.8.1 Contexte

ROSA et la densité résiliente ont été utilisés dans dans le cadre du projet DESEREC. Ce projet vise à fournir des méthodes et des outils pour surveiller, analyser, concevoir, modéliser, simuler et optimiser le plan de configuration des communications et des systèmes d'information (CIS). Le cadre DESEREC doit effectuer 3 missions qui sont:

- Modélisation. Cette tâche consiste à planifier et à définir l'utilisation et la configuration optimale de fonctionnement du CIS. Cela permet de définir le mode cohérent et opérationnel du CIS. Cela permet la détection des attaques ou des défaillances par comparaison au mode optimal de fonctionnement du CIS.

- Détection et prévention. Pour obtenir le mode de fonctionnement actuel du CIS, certains capteurs doivent permettre de mesurer la caractéristique du CIS. Les capteurs doivent détecter les signes avant coureur d'une défaillance ou d'une attaque.

- Reaction. Certaines initiatives et contre-mesures assistées par ordinateur et automatisée doivent être prises en cas de défaillance ou d'une détection d'attaque. Ces réponses doivent être rapides et adaptées à la nature des incidents détectés.

L'outil de suivi proposé dans le cadre de DESEREC consiste en un ensemble de capteurs distribués et un ensemble de mécanismes de détection pour détecter les attaques, les défaillances ou les bogues des services qui peuvent se produire dans le système. Quand un tel incident est détecté, l'outil doit réagir de manière rapide et appropriée conformément à une politique de sécurité. La politique de sécurité peut impliquer le système et la reconfiguration des services. L'outil proposé au cours du projet DESEREC est en mesure d'identifier les événements de malveillance et d'isoler les entités soupçonnées afin d'éviter la propagation de menaces ou d'un effet en cascade.

Puisque cet outil est destiné à être déployé sur le réseau d'une entreprise et puisqu'un tel réseau pourrait être partagé sur de nombreux pays et donc sur de nombreux systèmes autonomes et de sous-réseaux, nous ne pouvons pas tolérer que des pannes empêchent les capteurs partagé par ces sous-réseaux différents de communiquer. L'outil proposé par DESEREC, doit avoir des garanties sur l'efficacité de l'acheminement de ses données. Les capteurs doivent communiquer à l'aide d'un réseau résilient. Pour cette raison, ROSA a été choisi pour supporter cet outil.

0.8.2 Description

Un ensemble de capteurs est installé sur toutes les entités du réseau mesurables. Les capteurs installés sur une entité donnée dépend du type de cette entité. Les capteurs peuvent mesurer la charge du processeur, l'utilisation du disque, l'utilisation des interfaces réseau, la production de logiciels anti-virus, les erreurs dans les services, les fichiers logs et ainsi de suite.

L'outil utilise les données de surveillance de sortie des capteurs pour calculer une valeur d'assurance de sécurité tel que décrit par Pham et Riguidel dans Pham and Riguidel [2007].

De nombreux types d'entités peuvent composer un réseau, les entités peuvent être des postes de travail, passerelles, imprimantes, routeurs, sous-réseaux et ainsi de suite. Chaque type d'entités a sa propre définition de la valeur SA. La valeur SA d'une entité peut être calculée au niveau local ou à distance en fonction de sa nature. La valeur SA d'une imprimante ne peut pas être calculée sur place puisqu' une imprimante ne dispose pas de capacités de calcul. La valeur SA de certaines entités, comme les sous-réseaux, doit être calculée de manière distribuée.

Une fois les valeurs SA calculées, la politique de sécurité doit être appliquée. De nos jours, la politique de sécurité se compose de deux règles qui sont:

- Si la valeur SA d'une entité de routage (routeur ou une passerelle) passe sous un seuil donné, l'outil essaie de trouver une entité de routage alternative pour tous les sous-réseaux qui dépend de l'entité ayant une faible valeur SA. Si aucune alternative de routage ne peut être trouvée, pour un sous-réseau donné, le sous-réseau est isolé jusqu'à ce que la valeur de l'entité SA soit au-dessus du seuil. Si un autre entité est trouveé la table de routage des entités concernées est modifiée afin d'utiliser l'entité de remplacement.

- Si une valeur d'un sous-réseau SA passe sous un seuil donné ce sous-réseau est isolé. Cela signifie que toutes les entités du réseau doivent ignorer les paquets de données émanant du sous-réseau avec la faible valeur SA.

Les seuils des valeurs SA en dessous desquels la politique de sécurité exige une réaction doivent être déterminés expérimentalement par l'étalonnage des capteurs du réseau surveillé pendant une période où aucune attaque et aucune panne n'est rencontré rencontré. Cette période doit être suffisamment longue afin d'obtenir des seuils significatifs.

Beaucoup d'autres règles pourraient être ajoutées à la politique de sécurité en fonction des services fonctionnant sur le réseau surveillé. Imaginons qu'un serveur Web est exécuté sur le réseau surveillé et que les serveurs web de substitution existent. Si la valeur de SA du serveur Web principal diminue trop, l'un des serveur web alternatif est activé par l'outil de surveillance tandis que celui avec la faible valeur SA est désactivé. On peut imaginer autant de règles que l'on veut.

La Figure 8.2 illustre l'application des règles de la politique de sécurité en fonction de la valeur SA calculée par l'outil de surveillance. On peut voir que c'est l'outil de surveillance qui est chargé à la propagation de la reconfiguration des commandes.

ROSA et la densité résiliente sont utilisés comme l'épine dorsale de cet outil. Sa topologie, son service de routage résilient et sa fiabilité apportent une bonne assurance que les différents dispositifs de l'outil de surveillance sont en mesure de communiquer malgré

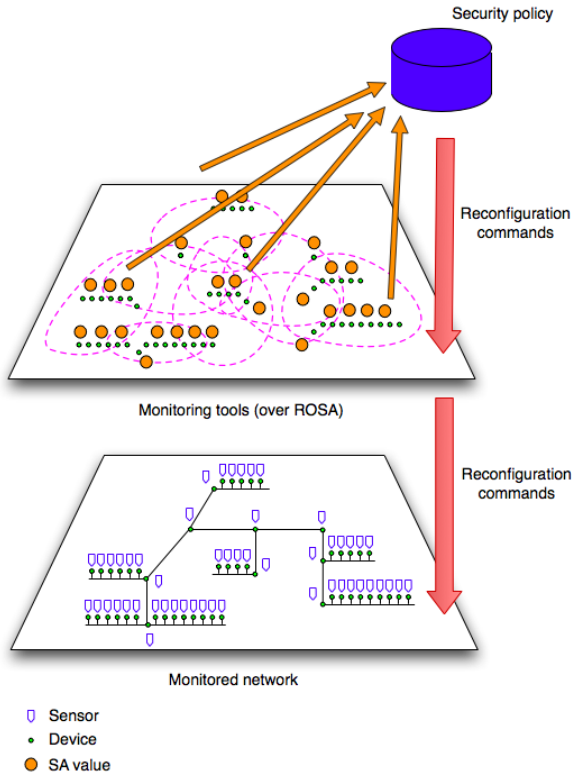


Figure 5: L'outil DESEREC interprétant les valeurs SA et appliquant la politique de sécurité

les pannes, les attaques et les modifications des tables de routage des entités du réseau par l'outil de surveillance.

0.8.3 Experimentation sur le réseau de Télécom ParisTech

L'outil de surveillance a été mis en œuvre et déployé sur le réseau du département InfRes de Télécom ParisTech. Ce réseau compte environ 50 postes de travail, plusieurs routeurs CISCO et est divisé en plusieurs sous-réseaux.

Dans cette expérimentation, nous avons utilisé de vrais capteurs sauf pour le capteur de virus car nous ne voulions pas vraiment propager un virus ou un ver sur ce réseau opérationnel. Nous sommes en mesure de simuler l'existence d'une attaque de ver ou la défaillance d'un élément du réseau. En ce qui concerne la politique de sécurité, seuls les deux règles déjà mentionnées dans les sections ci-dessus ont été mises en œuvre. L'application des règles de politique de sécurité ont été mises en œuvre virtuellement puisqu'une fois de plus, nous ne voulions pas interférer avec l'exploitation du réseau.

Chaque poste de travail et serveurs du réseau surveillé agissent en tant que nœud de ROSA et en tant que nœud de l'outil de surveillance. Toutes les communications entre les nœuds de l'outil de surveillance transitent à travers ROSA.

Afin de compléter l'outil de surveillance et de permettre des démonstrations un outil de contrôle a été conçu. Cet outil de contrôle est appelé le cockpit de sécurité. Le cockpit

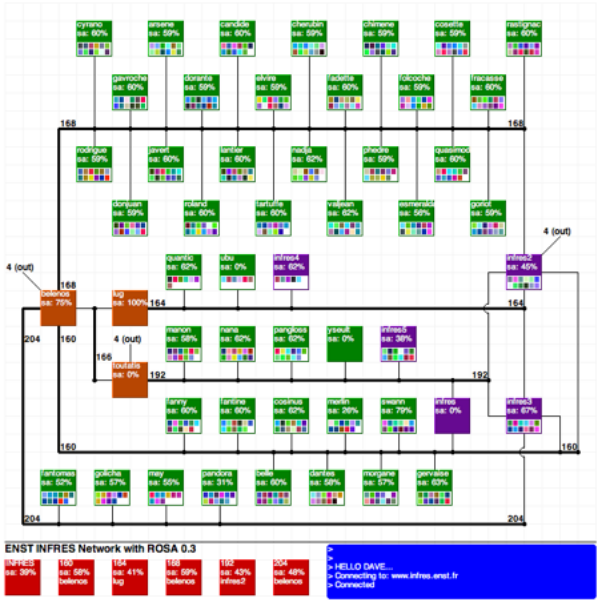


Figure 6: Le cockpit de contrôle de DESEREC

de sécurité consiste en une applet java qui affiche une représentation du réseau surveillé. Cette applet s'exécutait sur un serveur du réseau. Sur cette applet sont également affichés les liens virtuels de ROSA et les valeurs SA des différentes entités du réseau. La Figure 8.3 est une capture d'écran du cockpit de sécurité. On peut voir sur cette figure les postes de travail sous forme de carrés verts, les serveurs des carrés violets et les routeurs par des carrés oranges. Les carrés du bas en rouge correspond aux différents sous-réseaux du réseau surveillé.

0.9 Analyse

0.9.1 Extensibilité de ROSA

ROSA assure son extensibilité en limitant le nombre de voisins par nœud. Nous avons effectué quelques simulations afin de vérifier l'extensibilité de ROSA. Cette simulation a consisté à créer plusieurs instances différentes de ROSA. Chacune de ces instances de ROSA ont un nombre différent de nœuds. Puis, nous mesurons le nombre moyen de bits que les nœuds de ROSA envoient à chaque intervalle de temps. Le graphique 7.1 montre le résultat de l'une de ces simulations.

Pour la simulation correspondant à la figure, le nombre maximal de noeuds par forfaitaire est fixé à 10, le nombre maximal de grumeaux par nœud est fixé à 2 et la densité utilisé est la densité par défaut.

On peut voir dans la figure que la taille moyenne des données envoyées par un nœud pendant un intervalle de temps ne dépend pas du nombre de noeuds de ROSA. Les autres simulations effectuées montrent le même fait. On peut considérer ROSA comme extensible.

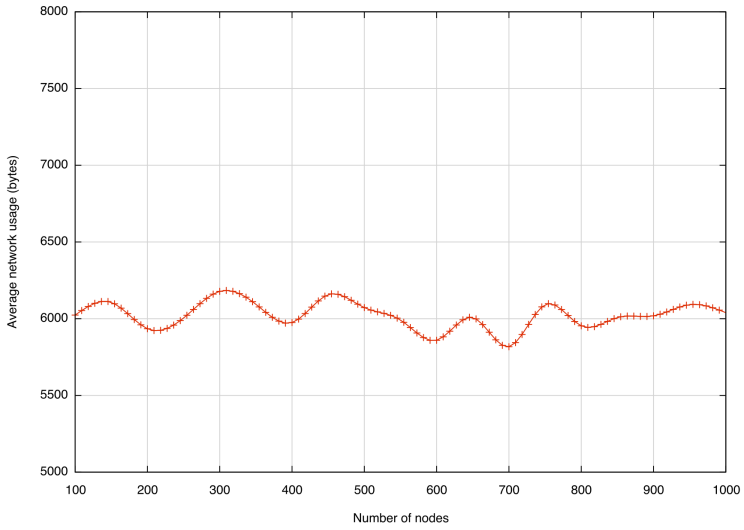


Figure 7: Nombre moyen de bytes envoyer par un nœud en fonction du nombre de nœuds du réseau

0.9.2 Efficacité du routage sur la 'chaîne de grumeaux'

Le pire des cas pour l'efficacité de la 'chaîne de grumeaux' est l'absence de raccourcis. Dans le pire des cas, nous avons démontré formellement que le nombre de sauts nécessaires pour acheminer un paquet de données à partir d'un nœud choisi aléatoirement à un grumeau choisi aléatoirement est inférieur ou égal à $N/4$ où N est le nombre de sous-intervalles qui composent la 'chaîne de grumeaux'. L'algorithme de routage utilisant la 'chaîne de grumeaux' est en $O(N)$.

Nous avons simulé le comportement de la 'chaîne de grumeaux' dans des conditions proches de celles rencontrées dans la réalité. Dans ces simulations, nous avons construit plusieurs instances de ROSA avec une 'chaîne de grumeaux' composée par de plus en plus de sous-intervalles. Le résultat d'une de ces simulations se trouve dans le graphique 7.5.

Chacun peut voir que le nombre de sauts ne croit pas linéairement en fonction du nombre de sous-intervalles et finit même par devenir constant. Ce résultat s'explique par un phénomène de 'petit monde'.

0.10 Conclusion

Les recherches de cette thèse apportent des contributions dans deux domaines. Ces domaines sont le domaine des réseaux de recouvrement et le domane des tables de hachage distribuées. Cette thèse a également proposé deux services qui peuvent être déployées sur ROSA. Ces services tirent avantage des propriétés de ROSA.

Dans le domaine des réseaux de recouvrement, cette thèse propose un nouveau réseau appelé ROSA. ROSA surpasse la vision des réseaux de recouvrement en graphe avec des nœuds et des liens à l'aide d'une couche d'abstraction: dans ROSA, les noeuds sont organisés en clusters appelés grumeaux. Un grumeau est un ensemble de nœuds entièrement connectés et ROSA peut alors être représentée par un enchevêtrement de grumeaux. ROSA

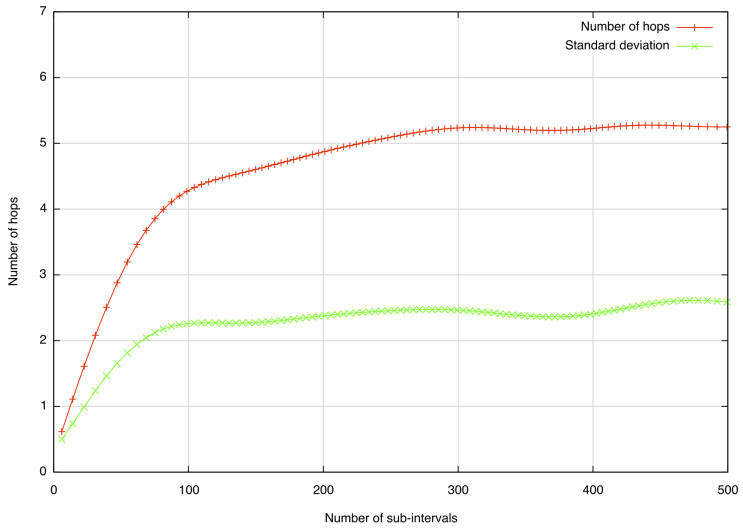


Figure 8: Nombre moyen de saut en fonction du nombre de sous-intervalles avec 2.5 raccourcis par nœud

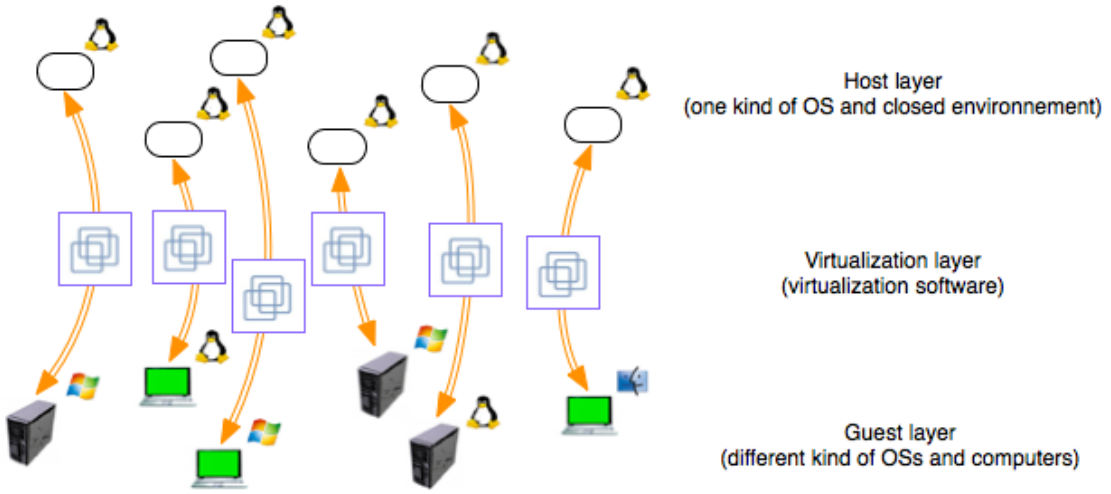
est un réseau auto-organisant, paramétrable, extensible et auto-guérissant.

Dans le domaine des DHTs cette thèse propose la 'chaîne de grumeaux'. La 'chaîne de grumeaux' est une DHT construite sur ROSA. Cette 'chaîne de grumeaux' doit remplir quelques conditions: la 'chaîne de grumeaux' ne doit pas réduire l'extensibilité de ROSA et ne doit pas modifier sa topologie non plus. La 'chaîne de grumeaux' est construite sur l'enchevêtrement des grumeaux. Les grumeaux sont organisés de façon à former une chaîne, cette chaîne est complétée par des raccourcis puisque les nœuds peuvent appartenir à plus d'un grumeau. Cette chaîne permet de router des paquets de données en un nombre de sauts bornés.

Introduction

Context of research

Recently, it has been possible to run an operating system on a hardware architecture for which the OS was not intended for. This technology is called virtualization. Virtualization runs each guest OS in a closed environment. These virtualized environments are managed by the virtualization layer that is implemented for a lot of host systems. It must also be able to run a large number of OS. Therefore, the virtualization is composed of three layers, the host layer (or physical layer), the virtualization layer and the guest layer. This is schematized in the Figure below.

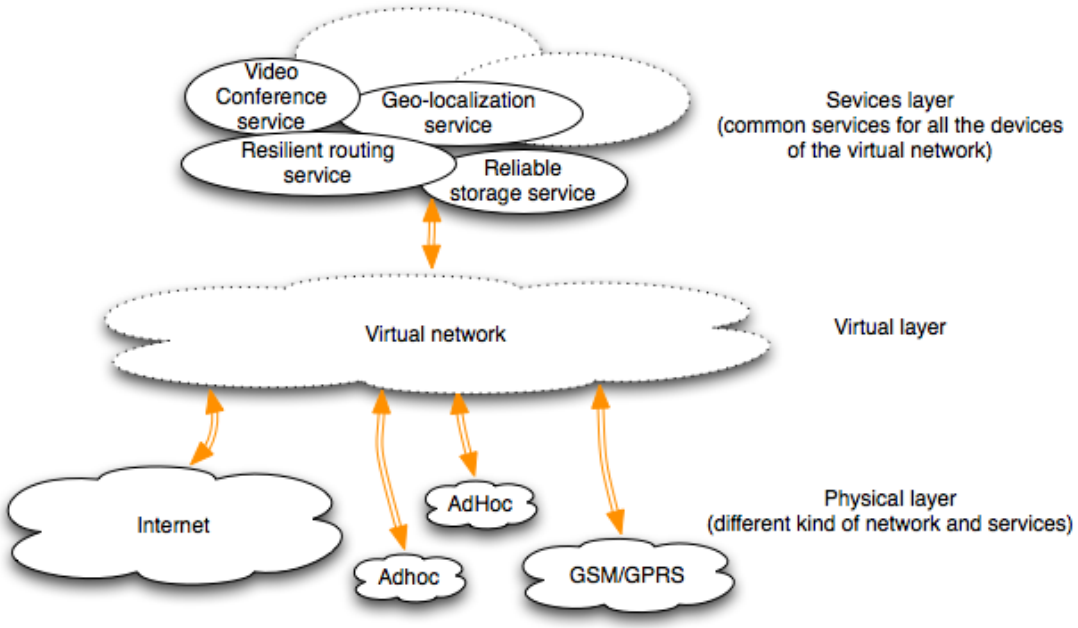


The Virtualization has allowed companies with a disparate set of computers to virtually have only a single type of computer. The benefits for a company to have its entire fleet computers identical are considerable. Moreover, all the effort needed is made by the designers of the virtualization layer. Therefore the installation of a new machine consists simply to copy a virtualized environment of an existing machine on the new one.

This concept of virtualization can also be applied to networks. Today there exists a great disparity of networks: Internet, the GSM network, ad hoc networks, etc. Each of these networks is based on different supports and protocols. It would be interesting to have a

virtual network to unify all these networks. Moreover, the convergence of digital electronic devices facilitates the development of such a virtual network. Indeed devices, such as new generation phones, spanning several physical network (IP, WiFi and GSM/GPRS) could serve as bridges between these physical networks.

The model for this global virtual network would be, by analogy with the virtualization of computers, as follows: the physical layer, the virtual layer, and the services layer. The physical layer is of a set of physical networks and the different devices that compose them. The virtual layer is a virtual network built over each of the physical network of the physical layer. The services layer consists in a set services provided on top of the virtual network of the virtual network. The services can be resilient routing services, reliable storage services, etc. This model is schematized in the Figure below.



In this Figure are represented at the physical layer 4 networks: Internet, 2 ad hoc networks and the GSM/GPRS network. Above the physical layer is built the virtual layer. This layer consists in a virtual network deployed over each of the networks of the physical layer. This virtual network provides common services for all the devices of the physical networks, these services are represented on the top of the Figure. With such a model, one would have similar benefits for networks to those provided by virtualization to computers. The different devices using the virtual network could communicate without worrying about the physical networks. One could also able to change the physical support without worrying about routing. One could also migrate to another type of routing with a negligible cost compared with the migration from IPv4 to IPv6, for example. To perform such a migration it is only needed to change the routing service provided by the services layer. All the works and efforts would be made in the implementation of the virtual layer for the different networks and physical supports.

A prerequisite for such a virtual network is the scalability. If one considers that a lot of devices will use it, we need the extra traffic generated the lowest possible. An overhead that does not depend or grow linearly on the number of devices would be ideal. The virtual network would also have the same qualities (resilience, etc..) that the networks on which it is deployed. The virtual network must be easily adaptable to a lot of different physical networks.

A class of network are very good candidates to be used as a virtual network: the overlay networks. An overlay network is a network built on top another network. To each node of the overlay network corresponds a node of the underlying network. These overlay networks have been popularized with the rise of the peer-to-peer networks (that belong to a sub-class of the overlay networks) in the years 2000. There exists many kinds of overlay networks, some are scalable, some ensure a resilient topology, some offers a reliable routing service, etc. Nevertheless the existing overlay networks cannot be used to be the virtual network since these networks are usually dedicated to only one service and can only be deployed over a single type of physical network. In order to use an overlay network as the virtual network, this one must be easily adaptable to a lot of different physical networks and must propose a lot of services. This is, in regard with these observations, that the goals of this PhD Thesis was defined:

- designing and building an overlay network adaptable and deployable over many kinds of physical networks ;
- proposing a set of services over the overlay network.

Major contributions

This PhD research contributes to the elaboration of the virtual network of the virtual layer in the model defined above by:

Firstly, designing an overlay network called ROSA. The nodes of ROSA are organized in cluster called lumps and ROSA can be seen as an entanglement of lumps. The nodes organize their neighbor sets according to the lumps and to the densities associated to these lumps. ROSA is scalable because the maximum number of neighbors that a node can have is bounded, and this bound does not depend on the total number of nodes participating to the network. ROSA is adaptable because the definition of the density of the lumps can be adapted to the properties of the underlying network. ROSA fulfills the necessary conditions to be the virtual network. Secondly, designing a Distributed HashTable over the ROSA overlay network. This DHT can be added to ROSA without modifying the initial protocol. Since some efficient complex services must be built over ROSA using this DHT, the DHT must provide an efficient lookup service. This lookup service has to be scalable. Thirdly, designing a reliable file storage service and a resilient routing service over ROSA endowed with the DHT. The reliable file storage uses the DHT and its lookup service to allow users to store file over the set of lumps of ROSA. The resilient routing service is also based of the DHT, it allows a node of ROSA to communicate in a direct way to any other

node. These two services demonstrate that it is possible to propose complex services over ROSA.

Structure of the thesis

The thesis is organized in 8 chapters as following:

Chapter 1 presents the state of the art of the overlay networks. It introduces a classification of the different overlay networks and the different applications built over these networks.

Chapter 2 deals with the principle of ROSA. It introduces the basic notion allowing to understand how ROSA works. It also introduces the protocols in charge of initializing, building and maintaining the topology of ROSA.

Chapter 3 presents a Distributed HashTable builds over ROSA. It describes in detail the protocol used to build and maintain the DHT. One can see in this chapter that the implementation of the DHT does not imply to modify the protocol presented in the Chapter 3. This DHT allows to build sophisticated services over ROSA.

Chapter 4 presents a reliable file storage service developed over the ROSA overlay network endowed with the DHT. This service allows user to store and retrieve a file over the nodes of ROSA in a reliable way. This service include primitive methods to manage the access rights of the users.

Chapter 5 presents a resilient node to node routing service built on top of ROSA endowed with the DHT. This chapter also introduce a way of computing the identifiers of the nodes of ROSA based on a reversible mixing function.

Chapter 6 introduces the notion of density. The density is a flexible parameter that allows ROSA to be adaptable and deployable over a large set of different physical networks. This chapter presents three different definitions of the density.

Chapter 7 presents the analyses of ROSA. In this chapter we study the scalability of ROSA, the efficiency of the DHT built on top of ROSA and the average load of each nodes of ROSA according to the fluctuation of some parameters such: the number of nodes, the maximum number of neighbors per node, etc. Most of these analyses are based on simulations.

Chapter 8 presents an real experimentation of ROSA. In this experimentation, which took place in the network of the INFRES department of the Telecom ParisTech school, ROSA was used to support a distributed tool in charge of monitoring the network.

Chapter 1

State of the art

1.1 Introduction

Internet obeys to simple and well known architectural principles and protocol: intelligence (that is to say, computing and information) is rejected at the periphery (network intelligence is a flat electroencephalogram), separation between computation and communication is affirmed by the dogma of the OSI seven layer model, the best effort delivery of messages in an asynchronous mode, split into packets formatted according to a description of incompleteness. Incompleteness means that the you can add functions, over the developments in the IP format, such as security of IPSec or the MPLS component (see Rosen et al. [2001]).

Since its inception, the networks developed on the Internet adopted an architecture in which resources and information from these networks were stored on servers with large processing capabilities and storage. Clients then access these resources and information by sending queries to these servers. The servers treat each query and return the resources or information to customers who had requested. The most popular networks developed on the Internet is the World Wide Web, where websites are stored on Web servers, and accessed by clients via http requests.

The routing of messages on the Internet is organized around entities dedicated to it. These entities called routers are responsible to deliver or disseminate the packets from one source to one or more (in case of multicast IP), destination(s) following the IP protocol, possibly using different routes for packets of the same message. Today, IP provides the best effort routing of packets. This means that IP offers no guarantee that packets are properly routed to their destination, nor guarantees the performance or scheduling of the packages.

The sole warranty regarding the reliability of the IP routing is the ability to control the errors on the packet using a checksum. This lack of control and reliability was deliberate, because it has the advantage of reducing the level of complexity operations to be performed by the routers.

The choice of client-server architecture is obvious when the capabilities of the computers of the users are weak in terms of computing power and bandwidth. This choice of architecture remains valid today for a new generation of customers with limited resources, such as the RFID chips for example. However the difference between the capacity of the servers and those of older types of customers is becoming weaker and, consequently, many

servers are thus unable to effectively perform their tasks, because of the demand ever more important of the various customers.

One solution is to increase the capacities of the servers or at least their numbers. This is the solution chosen by Meebo (Meebo [2005]). Meebo offers its users a service to simultaneously connect to multiple instant messaging systems. Meebo is composed of a large number of servers and a Web interface. An user wishing to connect simultaneously to AIM and MSN, enters its MSN and AIM identifiers via the web interface. A MSN client and an AIM client are then launched on a server of Meebo. These clients use the identifiers entered by the user to connect to their respective messaging services. The user may then communicate with these clients through the Web interface. Meebo therefore gives the user the impression of being directly connected to MSN and AIM, but it is only connected to a server of Meebo. There are also now more and more services that use a large farm of servers behind a curtain. The search engines such Google (Google [1998]), the applications Web2 such SecondLife (SecondLife [2003]) or Wikipedia (Wikipedia [2001]) have an impressive number of servers in parallel.

The alternative is to break with the client-server architecture to move towards an architecture where information, resources and calculations that was confided to the servers are distributed across a set of network entities.

It is also possible to criticize the mode of routing proposed by IP. Nowadays, the capacities of the routers have greatly increased, and it does not seem absurd to make them run a routing algorithm more complex and more efficient. That is what the MPLS routers do (although the comparison between IP and MPLS is not entirely justified given that IP is a protocol of layer 3, while MPLS is a protocol astride layer 2 and 3 of the OSI model). It would be welcome to replace IP by another protocol. But the replacement of IP would be costly in term of finance and in terms of the duration of migration. This would require, for example, rewrite all the applications that use the network so that they take into account the new protocol. From these observations arose the overlay networks. They allow, as it will be described, to build networks architectures where the routing protocols perform more than simple best effort routing. Overlay networks can be deployed without having to change the algorithms used by Internet routers. Subsequently, the overlay networks have proven to be good candidates as an infrastructure for applications such as shared resources (data, CPU cycles, disk space), data dissemination (multicast distribution of content) and some others. Some applications running on overlay networks are designed to ensure a certain quality of service or to increase security covered of the covered networks, protecting them against and ensuring the detection of external attacks. But if these new architectures can provide solutions for security and dependability of networks covered, they also raise new issues about their own safety. In fact, the overlay networks are also targets of attacks exploiting their characteristics. In the remainder of this Chapter, we give a description and definition of the overlay networks, present various applications developed on the overlay networks and some attacks used against these networks.

1.2 The Overlay Networks

1.2.1 Definition

An overlay network is a virtual network built on top of another network, this other network may itself be virtual or not. A network dial-up (connection by modem), for example, an overlay network on the telephonic network. However, in the context of this Chapter, the networks studied are primarily those covering the IP network or other overlay networks for the simple and good reason that no or very few documents and research papers dealing with overlay networks on another network technology that IP, which is a shame because the need to create overlay networks on 3G networks or to create an overlay network astride the Internet and other communication networks is real.

In the current literature, an overlay network is defined as a set of nodes and a set of virtual links connecting these nodes. These links correspond to paths (one or several) at the covered network layer. The virtual links may be encrypted and therefore theoretically make it difficult to listen or falsification of communications between two network nodes of an overlay network. However, the issue of the availability is not solved by this encryption, and the denial of services can disrupt such overlay networks. Each node in an overlay network has an unique identifier, often different from its identifier in the underlying network. A node also has a table of other nodes, neighbors. In this table can be found the correspondence between the identifiers on the overlay network and the identifiers on the underlying network of these neighbors. The advantage of this architecture is to create a virtual topology of "neighbors".

It is possible to define an overlay network of as a set of entities, belonging to one or more networks, that share and respect the same initialization, maintenance and endogenous routing services. We shall see later in this chapter what these services are. This new definition has the advantage of simply considering what might be a coexistence, cooperation or competition between two overlay networks. For example, two overlay networks are said to coexisting if they have common entities. And we say that two overlay networks coexist and cooperate if they also have the same endogenous routing service.

When we deal with the overlay networks, we must make a distinction between an overlay network and its applications. This distinction is often ignored and even in this article that proposes a new overlay network. To give an example of this non-distinction, it is often mistakenly declare that Kademlia (Maymounkov and Mazières [2002]) is a file-sharing protocol, however Kademlia can be used as a botmaster during a botnet attack (Gunther et al. [2008]). It remains clear that the topology of overlay networks predisposes them to be carriers for certain types of applications: a topology tree is such a topology to build an efficient multicast service, this predisposition is not restrictive.

In this Chapter, an overlay network is characterized by a set of initialization services, a set of maintenance services and a set of routing endogenous services:

- Initialization services are services which allow the overlay network to acquire desired topology. If they are virtually nonexistent in the static or manually managed overlay networks, they gain importance in the self-organizing networks, they are particularly responsible for adding new node.

- Maintenance services are designed to maintain the properties and invariants of the

overlay networks. For example, when adding or removing nodes, in case of failure, attacks or other disturbances, it is sometimes necessary to reconfigure the virtual links of the network so that overlay network is still able to function properly .

- Endogenous routing services are in charge of exchanging information between the different nodes of the overlay networks

It is possible that overlay networks do not have initialization or maintenance services. However, there is no overlay network with no endogenous routing service. It is also important to understand the distinction between overlay and peer-to-peer networks. The set of the peer-to-peer networks is a subset of the overlay networks. The confusion between these two are very frequent. A peer-to-peer network are participative overlay network where each node has the same rights and duties. Often on the peer-to-peer network each new user brings a new node to the network and there is not (at least initially) distinction between the roles played by each of the nodes of the network. To illustrate this distinction, it is possible to take the example of the overlay network SOS (Secure Overlay Services in Keromytis et al. [2002]). SOS is a network that aims to protect a target of attacks like Denial of Service. It consists of several nodes of four different types (SOS will be described more fully later in this Chapter) around the target to protect, these nodes do not belongs to many users but only the one that deployed SOS. SOS absolutely does not have any common point with a peer-to-peer network, but is an overlay network. This distinction must be made when talking about grid computing and current applications of overlay networks to share computation cycle. Grids are typically composed of powerful machines always available connected together with high performance networks. The resource discovery is based on a centralized model. In Globus (Luis et al. [2003]), for example, a user or application may access to an information about the resources of a given node by sending a request to a server application running on him or server holding information on a set of nodes. This system is therefore valid only in the case of grids arranged where the servers are known in advance. While overlay networks dedicated to sharing computing cycles are generally composed of a large number of machines running without any guarantee of availability and efficiency. The table 1.1 summarizes the usual differences between grid computing and overlay networks dedicated to sharing computing cycles.

Now that the basic notions on overlay networks have been presented, a classification of different overlay networks will be discussed, followed by a classification of different applications of overlay networks, both illustrated by some examples.

1.2.2 Classification of overlay networks

Establishing a classification of the overlay networks is not easy as there are many different networks. There is still a commonly used classification which is to categorize the overlay networks according to the network architecture that is centralized (pseudo-centralized) or decentralized, and then determine if its topology is structured or unstructured.

This classification does not appear to cover all the varieties of the overlay networks, it is important to add the concepts previously mentioned the concept of participative networks (or non-participative) and the notion of self-organizing networks and those requiring organization "manual". These concepts are closely related to security.

Types	Computing grids	Overlay networks dedicated to CPU cycles sharing
Architecture and Connectivity	Static configuration, limited scalability	Flexibility of the overlay networks
Control and operation Model	Centralized Control, static set of known participants	May use centralized or decentralized control, dynamic set of participants
Security, privacy and reliability	Confidence guaranteed, more secure with known participants	Reputation system at the nodes level and hierarchical security topology

Table 1.1: Differences between grid computing and overlay networks dedicated to CPU cycles sharing

In the remainder of this section will be defined and explained these different concepts. These definitions will also be accompanied by examples of overlay networks that best illustrate each concept.

1.2.2.1 Participative / Non participative overlay networks

An overlay network is a participative one if its operation depends on the cooperation of all users and if each node of this network is controlled by a user. Peer-to-peer networks are good examples of participative networks. Indeed, each user brings a new node, new resources to the network and collaborates with the other users. These participative networks are by definition very vulnerable to attacks because any user can cooperate and there is no guarantee that these users are not malicious. The dangers caused by the participative overlay networks are discussed in the Section 1.2.4. The participative overlay networks are in opposition to the non participative overlay networks where all nodes belong to one individual or one organization.

1.2.2.2 Manually organized / self-organizing overlay networks

Manually organized overlay networks

The popularity of peer-to-peer has largely helped spread the idea that the overlay networks do not require any human intervention during their initialization and configuration. The fact remains that this idea is false. In fact, some overlay networks do not have initialization and maintenance services and need to be configured manually. These networks are not scalable. There are tools facilitating the initialization and management of such overlay networks, the most famous of them is X-Bone (Touch and Hotz [1998] and Touch et al. [2005]).

X-Bone is a distributed system for the deployment and management of overlay networks over the IP network. It is composed of some Resource Daemons (RDs), some Overlay Man-

agers (OMs) and a GUI. A user wishing to create an overlay network sends a query through the GUI to a OM. The graphical interface allows you to specify the desired topology, the method of choice for identifying those nodes, the type of application to run on the network and other settings, but these options are limited. When OM receives a user query, it transforms the query into an invitation containing the necessary conditions to become a node of the overlay network in initialization. This invitation is subsequently broadcasted (using multicast UDP). The RDs, who are responsible for managing the resources of host on which they runs, listen to these invitations and respond positively to this invitation if the available resources meet the demands of the invitation. The OM, in cases where there is a sufficient number of favorable responses, selects an arbitrary subset of the replying RDs equals with a cardinal equals to the number of nodes wanted for the overlay network. Then TCP/SSL tunnels are drawn between these RDs in agreement with the topology desired by the user. In case of insufficient number of responses, a warning is sent through the GUI to the user, indicating the failure of the creation of the overlay network. Once the overlay network is created, the OM periodically sends reminder pulses to nodes in order to maintain a view of the status of these nodes. In case of failure or malfunction one of these nodes, the OM's replaces the failing node by another RD and reconfigures the network.

Self-organizing overlay networks

The majority of the overlay networks do not need to be configured manually. They have automated initialization services that handle the addition of nodes and the acquisition of the desired topology. These networks are therefore more scalable and easy to deploy as requiring no human intervention. There are two major groups of initialization services: services using third party and fully distributed services.

The initialization services using a third party have the advantage of avoiding any conflict because they have an overview of the entire overlay network. The main drawback is that the third party responsible for initialization of the overlay network is a single point of failure. Indeed, in case of failure of such third party, no more new node can be added. The third party is a very easy target in the case of participative overlay networks. Since the overlay network can be joined by anyone, the third party has to be known by everyone. Consequently, it is also known by a potential attacker. The overlay network Skype (Baset and Schulzrinne [2006]) uses a initialization service based on a third party. Every Skype user contacts the login server via its client. The role of the login server is to elect the machine of an user as a super-node if resources of the machine permits it, or redirect the machine to a super-node if its resources are not sufficient. The login server then manages the addition of node and the network topology. Generally, this type of initialization services is not used by the overlay networks with decentralized architectures. These overlay networks privileges the decentralized initialization services.

Most of these decentralized initialization services operate as follows: initially, the new node must contact a node already present in the overlay network. Then, a place in the network topology is allocated to the new node and the new node integrates the endogenous routing service. These decentralized services have the advantage of being a solution to the problem of the single point of failure of the centralized initializations services. As drawbacks, these decentralized services require a way for locating at least one node of the overlay network from the underlying network. Kademlia (Maymounkov and Mazières [2002]) uses this type of services. If the user connects to the Kademlia network for the first

time, it can download a list of active nodes of Kademlia through the World Wide Web (from http://download.overnet2000.de/nodes.dat for example). If the user has already connected to Kademlia, its client has stored a list of nodes that were present at its last connection. These nodes will eventually spread the existence of the new node to the other nodes in the overlay network, and so the new node could participate in the endogenous routing service. The identifier, and then the location in the topology of the new node, is calculated using a hash function on a set of private customer data (IP address, MAC, etc..). The workings of Kademlia will be discussed in more detail in this chapter.

1.2.2.3 Centralized / Decentralized Architectures

Centralized Architectures
An overlay network with a centralized architecture is a strong central node around which other nodes are grouped. This is the first architecture adopted by the overlay networks and is primarily a network architecture especially suitable for developing applications for indexing, locating and sharing resources. The server has the task of indexing resources and allows nodes to locate and access it.

It is the architecture adopted by the overlay networks dedicated to distributed computing such BOINC (Anderson [2004]) and XtremWeb (Fedak et al. [2001]) that are described in more detail later in this chapter. In these networks, the nodes, in order to perform the calculations, are organized around a task server which distributes the calculation to perform to them.

Napster (Napster [1999]) also adopted this architecture. Each user that wishes to join the Napster network must identify themselves to a central server. Once connected, the server sent information about the files it wishes to share. It stored these in its database. An user that wishes to find a file, queries the central server, which according to its database sends a list of nodes owning that file. Then, a series of transactions between the node requester and the node owner are performed. These transactions may lead to the download of the file. The main server of Napster was shut in 2001.

The developer of BitTorrent (BitTorrent [2005]) also chooses this architecture for its network. In BitTorrent, a user wishing to download a file must first find (usually on the Web) the torrent file corresponding to the requested file. Each torrent file contains the address of one or more trackers. These trackers have a list of nodes downloading the desired file. The user then contacts one of the trackers that the torrent file contains. The tracker adds the IP address of the new node to its database and returns the list of other nodes downloading this file to the node. The nodes downloading the same file, can connect with each other and exchange parts (called chunks) of this file.

The advantage of this architecture is that it allows any node to directly contacts the other nodes that own chunks of the requested file. This is made possible by the fact that the server has a view of the whole network. In general in the case of a centralized architecture routing messages between any two nodes in the overlay network has a complexity of $O(1)$.

The main drawback of this architecture is that the central server is a single point of failure of the network. In case of failure or attack against it, the whole network is decommissioned. Another shortcoming is that the central server has finite capacity, it is therefore quite conceivable that if too many nodes join the overlay network the central server is overloaded.

Pseudo-centralized architectures
In order to remedy to the defects of the overlay network with centralized architecture the pseudo-centralized architectures have emerged. In these architectures, the central server is replaced by a collection of smallest servers communicating with each other. This is the solution adopted by FastTrack and Skype whose authors are the same.

FastTrack is the an overlay network that supported the KaZaA application (Liang et al. [2005]). KaZaA was an application for indexing and sharing files between different nodes. In Skype and FastTrack, nodes with larger capacities are elected as super nodes, these super-nodes are interconnected and act as a central server. If the overlay networks with pseudo-centralized architecture have the same routing properties as the networks with centralized architecture, they are much more scalable. To give an order of magnitude, there was before the shut of KaZaA (in 2005 due to copyright infringement) over 3 million users simultaneously present. But be careful, if indeed FastTrack allows indexing the files of 3 million users, however it is unable to simultaneously process 3 million accesses to the index.

Decentralized architectures
An overlay network with a decentralized architecture is a network where routing does not require any central authority. The endogenous routing service is distributed. Each node then has a knowledge about the local network, i.e. it knows all other nodes to which it is connected. A message is then routed from source node to destination node by a series of successive hops. Each node is using local knowledge at its disposal to bring the message closer to the destination. The performance in terms of complexity of these protocols are lower than those of centralized protocols but these overlay networks are much more flexible and scalable. These overlay networks with decentralized architecture can be divided into two classes, structured overlay networks and unstructured overlay networks.

1.2.2.4 Structured / Unstructured overlay Networks

Unstructured overlay network
The unstructured overlay network are the first overlay networks with decentralized architectures that have emerged. Usually on these networks the set of neighbors of each node is randomly chosen. We can often see the unstructured overlay networks as random graphs. These overlay networks have the advantage of being easy to implement and easily deployable. The endogenous routing service in these networks consists most often in a flooding service. A node wishing to send a message to another node of the overlay network sends it to all its neighbors. The neighbors at first receipt of the message, also send to all their neighbors and ignores the message if it receives it again, and so on. This system ensures that the message reaches the destination node.

However, the routing by flood can lead to congestion of the network if it is composed of too many nodes. This type of network is not very scalable. A solution to avoid congestion is to assign each message a maximum number of hops (the equivalent of what may be the TTLs in IP). Whenever a copy of a message is received by a node, the field maximum hop count is decremented. When the field drops to zero, the copy is simply ignored. The disadvantage of this solution is that no longer any guarantee that the message reaches the destination node. Indeed, if the shorter distance (in number of virtual links) between

the source and destination nodes is larger than the maximum number of hops permitted, the message cannot be transmitted. A probabilistic calculation determining the minimum distance between any two nodes of the network is needed to determine the maximum number of hops needed. This is not an easy thing to do, especially if the network is very dynamic and that a large number of nodes leaves and enters the overlay network. Another solution is to assign each node a number of neighbors proportional to its capacity, CPU and bandwidth, therefore the network load is distributed according to the capabilities of each nodes. This solution, without completely eliminating the risk of overloading the network, greatly reduces it.

Now we will present two overlay networks, both unstructured and used for sharing files between users. The first one called Gnutella, avoid congestion using the first solution. The second one, Gia, is an improved version of Gnutella, it has opted for the second solution to decongest the network.

- Gnutella (Gnutella [2003]) is a participative self-organizing overlay network. It uses a decentralized initialization service and therefore it is necessary for a user that wants to join Gnutella to know at least one node on the network. In this purpose, it can contact a server responsible for maintaining a list of connected nodes. The new node signals its presence by flooding the network with a query containing the IP address, network nodes receiving this request reply with their IP addresses. The new node selects a predefined number of nodes that responded and establish virtual links with them. These nodes will be its neighbors. When a user wants to download a file on Gnutella, it sends a search query containing a description of the file desired and a maximum number of hops. This number is a parameter imposed by the network when connecting. Each node receiving this query searches through the files he wants to share if there is one who fits the description. If such file is found, it replies positively to the node that initiated the request. These responses, in early versions of Gnutella, follows the same path that the query took. Nowadays, the request contains the IP address of the node that initiate them and a tunnel is then drawn between the node that owns the file and the node searching the file. There then follows a series of transactions that lead eventually to the download of the file.

- The idea of Gia (Dufour and Trajković [2006]) has been inspired by the authors in observing the gain in terms of scalability has obtained FastTrack using the heterogeneous nodes in a overlay network. Its authors applied this same principle to Gnutella. In Gia, to avoid that the nodes be overloaded, a search query is propagated only to nodes allowing it. Each node of Gia, when joining, distributes tokens to its neighbors. Each token allows a neighbor of the node to send one search query to the node. These tokens are redistributed when the search query is performed. Each node also controls its bandwidth. Once it exceeds a certain level of use, the node will not redistribute the tokens to its neighbors, warning them that it does no more search queries for the moment. Once the use of its bandwidth is sufficiently low, the node starts again the distribution of tokens to its neighbors which can therefore forwards search queries to it. This solution is effective when all the nodes actually play the game, but a malicious node can abuse the system by keeping its token to save its bandwidth. This attack called free riding will be treated in the section on overlay network security. In order to consume less bandwidth, Gia uses a routing

algorithm based on a random walk rather than a flooding one. A copy of the message is propagated to only one neighbor randomly determined for each hop.

One of the shortcomings of these unstructured overlay networks is that they are starved of security (confidentiality, in particular). Indeed, whether in the case of a flood routing or by random walk, any node in the network may have at some time in hand a clear message that was not destined to it. These items are discussed in more detail in the sections dealing with the security of the overlay networks.

To finish with these unstructured overlay networks, it is important to note that most of these networks do not have complex maintenance service. This is due to the lack of problems when a node fails or leaves the network. Usually when a node does not receive no more signs of life from one of its neighbors, it replaces it by another node in the network and updates its neighborhood table.

To give an order of the scalability of such networks, there was at its apogee about 1 to 2 million users online and indexing files on Gnutella. And since an average user indexes between 5 and 50 files, one estimated the number of files indexed at 50 million.

Structured overlay networks

The use of the inefficient routing algorithms such flooding or random walk ones is the weak point of the unstructured overlay network. To allow the use of more clever routing algorithm, new overlay network topologies were studied. Structured overlay network emerged from this studies. The structured overlay networks have, as their name suggests, a structured topology. The nodes have a position in the network topology, and are distributed (with a very high probability) on a regular basis. These overlay networks are often organized as Distributed HashTable.

This allow to use efficient routing protocols with a bounded complexity. When a node wants to send a message to another node, it is possible, knowing the position of the destination node, to route the message to its neighbor closest to the destination node. This neighbor will do the same, and so on until the message reaches the destination node. We must pay attention to the meaning of distance, the distance between two nodes is by no means the geographical distance separating them, each network implements its own distance. A distance in a mathematical meaning is defined as $d : E \times E \rightarrow \mathbb{R}$ such that:

$$\forall u, v \in E : d(u, v) = d(v, u)$$

$$\forall u, v \in E : d(u, v) = 0 \Leftrightarrow u = v$$

$$\forall u, v, w \in E : d(u, w) \leq d(u, v) + d(v, w)$$

where E is the set of nodes in the network.

However most of the structured overlay networks use pseudo-distances where some of these three conditions are not always respected. The symmetry (first condition) as the triangle inequality (second condition) are often ignored in the definition of these networks. Let us consider the case of a network organized in a directed ring and composed by N nodes. The nodes are identified by their relative position to a given node with 0 as identifier. On this network, let u_i and u_j be two nodes, the distance between u_i and u_j is defined as: $d(u_i, u_j) = j - i \ mod(N)$. With this distance the symmetry is not respected since:

$$d(u_i, u_j) = d(u_j, u_i) \quad \Leftrightarrow \quad j - i \ mod(N) = i - j \ mod(N)$$

$$d(u_i, u_j) = d(u_j, u_i) \quad \Leftrightarrow \quad 2j = 2i \ mod(N)$$

$$d(u_i, u_j) = d(u_j, u_i) \quad \Leftrightarrow \quad j = i \ mod(N/2)$$

Similarly it is quite possible to consider distances that do not meet the last condition. Consider then the IP network and define the distance as the response time to a ping. Experience shows that this distance does not respect the triangle inequality. In the remainder of this section will be presented four structured overlay networks frequently encountered.

The Plaxton mesh (Plaxton et al. [1997]) is a distributed data structure optimized to support overlay networks. Pastry (Rowstron and Druschel [2001]) and Tapestry (Zhao et al. [2004]) are based on a Plaxton mesh.

In a Plaxton mesh network, each node has an unique identifier, this identifier is a sequence of m digits in base b. This identifier is obtained by using a hash function, SHA-1 for example, on specific data of the node. Using a hash function ensures with a high probability an homogeneous distribution of nodes on the space of identifiers.

Each node n also has a table of neighbors. This table has m levels and b entries at each level. The i^{th} entry of j^{th} level contains the identifier and the position of the nearest node whose id ends in suffix $i + (n, j - 1)$. To give an example, in a Plaxton mesh where the identifiers are sequences of 10 base 8 digits, the 6^{th} entry of 8^{th} level of the node whose identifier is 3426742075 is the node whose identifier is nearest and ends by in **6**6742075. The distance used in plaxton mesh network is the alphabetical distance.

The routing protocol in a Plaxton mesh consists in routing messages from source node incrementally digit by digit until the messages reach the destination node. Each node receiving a message and that is not the destination node consults in its neighbors table the level corresponding to the number of hops already done and thus determines the next node that should receive the message. This protocol ensures that a message reaches the destination node by performing a maximum number of hops equals to m. The Figure 1.1 shows an example of routing in a Plaxton mesh where identifiers are sequences of 5 digits of base 3, the 21,022 node sending a message to node 10,211.

A Plaxton mesh is static by definition, there is no initialization and maintenance services. However, the overlay networks based on a Plaxton mesh implement these services.

Chord (Morris et al. [2001]) illustrates the overlay networks structured into rings. Each node in Chord has an unique identifier. This identifier is a sequence of m bits, which accordingly sets the maximum number of nodes in Chord to 2m. The identifiers are obtained, as in the case of plaxton mesh network, by using a hash function on specific data of the nodes. Initially, each node only knows its successor on the ring. However in order to improve routing performance, each node knows its predecessor and also maintains a neighborhood table called finger-table . The i^{th} entry of the finger-table of a node contains the node identifier of the first successor to a minimum distance of $2^i - 1$ on the ring.

The endogenous routing of Chord consists in routing the messages from the source nodes to the destination nodes through other nodes in the network. Each node receiving the message consults its finger-table and sends the message to the node nearest the destination node. In this way, a message is routed to its destination in $O(log(N))$, where N is the total

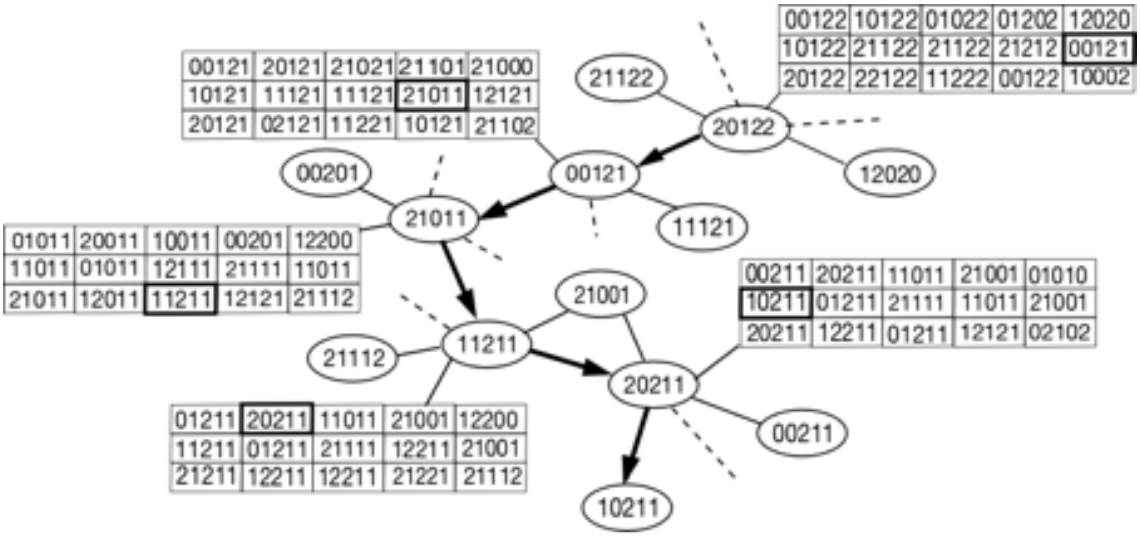


Figure 1.1: Routing a message from the node 21,022 to the node 10,211 in a Plaxton mesh

number of node on the network. More details concerning the routing of Chord is given in the Section 3.1.3.1.

The initialization service of Chord requires that any node wishing to join Chord already knows a node already present in the network. This node is called bootstrap node. The bootstrap node has to perform two operations. It first helps the new node to build its finger-table by determining the predecessor and successor of the new node according to its identifier. Then the bootstrap node fills the i^{th} entry of the finger-tables of the nodes whose the i^{th} entry is the successor of the new node.

The maintenance service is responsible for ensuring the invariant of Chord, namely that each finger-table must be properly filled. Three scenarios may affect this invariant, the departure of a node, the failure of a node or the case where multiple nodes joins Chord in the same time. The case departure of a node is not hard to deal with, because the node has to notice another node from its departure. This noticed node has to perform the inverse of the second operation performed by a bootstrap node during the join of a node. The function stabilize handles the two other cases. If a node does not give sign of life, the node that detect the non-responsive node contacts another node in its finger-table and asked the identifier of the successor of the node that does not anymore gives sign of life. It replaces in its finger-table the identifier of the non-responsive node by the new identifier received.

Kademlia (Maymounkov and Mazières [2002]) is another structured overlay network. Each Kademlia node has a 160-bit identifier, that identifier is obtained in the same way as Chord and the Plaxton mesh. Each node n stores for all k, $k \in [0, 160]$, a list of k triples <IP, UDP, Identifiant> nodes whose distance to n belong to the interval $[2^i, 2^{i+1}]$. These lists are called k-buckets. The distance used in Kademlia is given by the XOR function: Let n and n' be two nodes of Kademlia, the distance $d(n, n')$ is equal to $n \oplus n'$. From the definition of this distance we can deduce that all nodes present in a single k-bucket share the same prefix. Each time a node receives a message, the triplet <IP, UDP,

Identifiant> corresponding to the node that sent the message is placed at the beginning of the k-bucket and the corresponding k^{th} element of the k-bucket is lost in this way k-buckets are compounds that the node seen recently.

The endogenous routing of Kademlia consists, for a node that wants to a message to another node in the network, in obtaining the triplet <IP, UDP, Identifiant> corresponding to the destination node. To get this triplet, the node chooses, in the k-bucket corresponding to the identifier of the destination node, the node closest to the destination node. Then the node sends a message asking the selected node to do the same, and this until the destination node is found. If the destination node exits it returns its triplet to the requesting node. Otherwise a message indicating the failure location is returned in place of the triplet. Once the triplet target node in its possession, the node can send messages directly to that node. This type of routing is called iterative routing. This will be addressed later in the document. The Kademlia routing is detailed in the Section 3.1.3.1.

The initialization service of Kademlia is quite ingenious and simple, it takes advantage of the endogenous routing service. A node that wants to join Kademlia needs to know a node already connected. It sends through an already connected node a request in the hope of locating itself. Since it is not known on the network, this request will be unsuccessful and the node will receive a series of messages noticing this fail. Thanks to the properties of routing protocol, these messages come from nodes close to him. The node will use the nodes that have replied to initialize its k-bucket. Moreover, the nodes receiving the request of the new node, have added to their k-buckets, and it is now known on the network.

There is no maintenance service in Kademlia at the opposite of most of the structured overlay networks. If a node fails or leaves the network, it will be automatically phased out of the k-buckets of the other nodes.

CAN (Ratnasamy et al. [2001]) is a structured overlay network of using the Euclidean distance. The IDs of nodes in a CAN network are Cartesian coordinates of dimension d. A CAN network can be viewed as a d-dimensional torus. The whole torus is partitioned between different nodes, each node is (from all at least) responsible for an area whose center is the identifier of the node. Each node then maintains a list of neighborhood containing the list of nodes which are geographically close.

The endogenous routing service of CAN is based on a greedy algorithm. When a node wants to send a message to another node in the network, it sends to its neighbor geographically closest to the destination node. This neighbor does the same and so on until the message reaches the destination node. This routing algorithm has a complexity of $O(d.N^{1/2})$.

The initialization service of CAN is divided into four stages. The new node randomly chooses a point P in the spatial coordinates of CAN and, through a node already present in the network sends a JOIN request to this destination. The node responsible for the area which belongs to P, divide the area into two sub-areas and assigned one of these sub-areas the new node. Neighbors of the shared area are warned of the change and modify their neighbors table in function of the topology change. Finally, the new node obtains its neighborhood table asking the node that gives its sub-area to it.

The use of a maintenance service, during the addition, the departure or failure of a node is essential in CAN. Indeed the two invariants of a CAN network is that all space must be assigned and that each node should be responsible for only one area of space. But

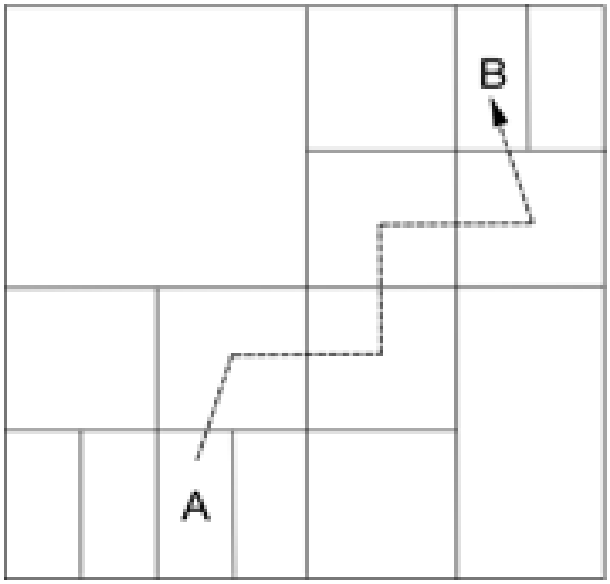


Figure 1.2: Example of Routing in a CAN network in two dimensions

the departure of a node can force such a neighbor of this node need to manage more than one area. A decentralized algorithm is used periodically to ensure these invariants.

To give an order of magnitude of the scalability of such structured overlay networks, it is estimated between 3.5 and 5 million users on Kademlia and approximately between 500 million and 1 billion files indexed.

1.2.2.5 Conclusion

We just defined a simple classification for different types of services initialization, endogenous routing and maintenance used by these different overlay networks. This is summarized in the table 1.2.

This table is correct in most cases. However, it is possible to find exceptions among the wide variety of the existing overlay networks. One might cite the case of ALMI (Pendarakis et al. [2001]) described later in this document, which is an unstructured overlay network, which therefore belong to the decentralized architecture overlay networks classes but, nevertheless, has a centralized initialization service.

1.2.3 Classification of applications

Now that a classification of the overlay networks has been established, it remains to classify the applications developed on these networks. We can decompose all of these applications into four groups. The first group deals with applications for indexing, locating resources. The second group consists of applications that allow users to share their resources. A third group will be composed by applications of overlay networks for routing. And the last one includes applications designed to increase the security of the underlying network. There also are applications that do not fit into any of these four categories, they will be listed in a fifth group.

Services / Network	Manually organized	self-organizing			
		Centralized	Pseudo-centralized	Decentralized	
				Unstructured	Structured
Initialization	none	Simple centralized service O(1)	Simple centralized service O(1)	Simple centralized service O(1)	Complex decentralized services (complexity often equals to those of the routing protocol)
Endogenous routing	Undefined	Simple centralized service O(1)	Simple decentralized service O(1)	Decentralized service (by flooding or random walk)	Complex decentralized service ($O(log(N)$, $O(N^{1/2})$, etc.)
Maintenance	none	none	none	none	Complex distributed service

Table 1.2: Classification of different overlay networks

1.2.3.1 Indexing and locating resources

The applications of this first group aim to allow the users to index and locate resources on an overlay network. These resources may be, for example files in the case of applications for file sharing, or identifiers of a contact person in case of application of voice over IP.

We must be careful in a lot of document these applications are not considered as application for indexing resources but they are considered as distributed file storage storage or voice over IP applications. This is not totally false, since the user uses the index of these applications to exchange files or looking for an interlocutor. But here's a short example in which it will be shown that it is possible to make voice over IP over Kademlia the same way as Skype without changing the initialization, maintenance and endogenous routing services of Kademlia. This proves that there is no real difference between the functionality of these two networks and the only difference lies in their respective uses.

Let us imagine that an user, called *Lambda*, has in its file shared with Kademlia a zero length file named *voiceLambda.voip*. A search for this file through a normal Kademlia client results in obtaining the IP address of anyone who owns this file and download this file. Consider now a modified Kademlia client that, when it obtains the IP address of an user that stores a *.voip* file, no longer seeks to establish a connection leading to the download of the file but instead it contacts the person with the file in order to establish a voice conversation over IP. If the person with the *.voip* file also have a modified Kademlia client that accepts connection for voice over IP conversation. It is then possible to use Kademlia to initiate voice conversations over IP just as it does with Skype and without modifying Kademlia. This explains why we have defined the applications running on these networks as indexing application.

The first type of overlay networks which have been developed such applications are centralized ones. The central server acting as an centralized index. This is the case of Napster and BitTorrent. In both overlay networks a central server lists files and IP addresses that the users have. A user wishing to download a given file, consult the index (server) to obtain the IP addresses of users with the desired file. Once in possession of these addresses, it can initiate a series of transactions at the underlying network level leading eventually to the download of the desired file.

The part on iVisit (iVisit [1997]) that consists in the search for a given interlocutor also works this way. A central server identifies the users connected and allows a search according to their IDs. It is then possible for users to connect directly to establish a video conference connection over IP.

The second generation of indexing applications was developed on overlay networks with pseudo-centralized architectures. This is the case of KaZaA, an application for indexing files and linking users with the intention of sharing those files. This application was developed on the FastTrack network previously described in this document. It is also the architecture which was chosen for Skype concerning the establishment of the telephone communication over IP. The section on management of the telephone itself will be discussed later. These applications developed on centralized architectures have the advantage of allowing search queries as complex as one wants.

However, these applications suffer from the same defects as the networks on which they run, i.e. they are subject to interruptions if the main servers fail or are saturated. Then to remedy these defects, indexing applications were developed on overlay networks with a fully decentralized architecture.

Gnutella was the first overlay network used for files indexing. In Gnutella, search queries are sent by flooding all the nodes of the network. Sending queries by flood allows complexity of these queries as big as you want. But these overlay networks have disadvantages to not be very scalable, which may be detrimental to a network aiming to index the files of millions of users.

The latest generation of applications uses the structured overlay network as basis. This applies to applications using the network Kademlia for sharing files, eMule, AMule, Overnet and many others.

- When a user wants to index a file, it uses a hash function on some specific data of the file (such filename or keywords), thus obtains an identifier for this file. It then searches the node with identifier closest to the obtained file identifier and asks this node to store its IP associated to the file identifier.

- When a user tries to download a file, it searches the node with the identifier that is the closest of the file identifier. This node returns a list of IP addresses of the users that indexes the file. The node wishing to retrieve a file contacts one the node storing the file. Then it follows a series of transactions leading eventually to the download of the file.

The main drawback of the indexing applications developed on such overlay networks is that they no longer allow complex search queries. But they have the advantage to be particularly scalable.

1.2.3.2 Resource Sharing

The applications of this second group aim to enable resources sharing between the users. These resources can be of free disk space or CPU cycles. In fact, one could also mention the applications to share the unused bandwidth users. This is done in some applications providing a multicast service. But these applications are classified in the group devoted to routing.

Regarding the applications for sharing free disk space, it is very important not to confuse them with the applications dedicated to the files sharing. Indeed, in the files sharing applications the files shared by a user are stored on the node of this user. Users of applications dedicated to the sharing of disk space, meanwhile, store information on the disks of all the nodes that participate to the application without knowing exactly where this information is stored.

Sharing free disk space

There are two kinds of sharing disk space applications, the publications/consultations systems and the distributed file systems. Publications/consultations systems allow users to publish or to view content on a network, much like Websites. These systems have the advantage of allowing redundancy of publications and the anonymity of users that publish and consult the contents. Two publications/consultations systems developed on overlay networks will now be presented.

Freenet (Clarke et al. [2001]) is the first publications/consultations system developed over an overlay network. Its purpose is to provide a reliable means for the anonymous exchange of information and provide freedom of expression as text websites or PDF documents whose publication is subject to censorship in the web. It can also provide redundant data availability. Each node of the overlay network has an identifier and a table of other nodes near him on the network. The routing is done as follows: a node wishing to send a message to another network node, the node sends it to the node whose identifier is the closest to the destination node identifier, and so on until reaching the destination node. When a user of Freenet wants to publish content, it computes the identifier of the content and sends a query including the content and the identifier to the node whose identifier is closest to the content identifier. When a user wants to view content, it sends a query to the node whose identifier is closest to the identifier of the content it is looking for. This query contains the identifier of the node that performs the query and the identifier of the content. If this node does not have the desired content, which is possible because nodes could join the network after the publication of content, it forwards the request to the second nearest node, and so on. In the case of a successful query, the node that owns the content returns it through the other nodes of the overlay network until the node that performs the query. Each node on the way back also stores this content. This mechanism allows nodes to specialize the content that they store according to their identifiers. In an attempt to hide the identity of the users posting or consulting the contents, Freenet allows the nodes to replace the source identifier of a request by its own identifier.

Free Haven (Dingledine et al. [2000]) has exactly the same goals as Freenet. Free Haven provides a publications/consultations service as anonymous as possible. When a user wants to publish content on Free Haven, he must first decompose the content into n parts in such a way that only k parts are sufficient to recreate the contents, $1 < k < n$. Each part

associated to the identifier of the content is stored on a node. When an user wants to retrieve a given content, it floods the entire network with a query containing the content identifier and the address of an anonymous remailer such Mixmaster (anonymous remailer). The nodes that owns the parts of the queried contents send them to the querying user via the remailer address. When the user has k parts, he can reconstruct the queried content. To ensure more anonymity, the nodes periodically exchange parts of content. This also allows a node wishing to leave the network to get rid of the parts it has and so does not make them disappear along with it.

It seems clear that the overlay networks are interesting platforms on which we can develop all kinds of distributed applications. It is therefore not surprising to see distributed file systems appearing on these networks. As publications/consultations systems the distributed file systems allow users to store content in a distributed way on a network. The difference between these two systems is the lack of rules of access control in the publications/consultations systems. The overlay networks used to support these distributed file systems are structured ones. The possibility of using these networks as distributed hash tables is an advantage to develop such a system because it allows efficient indexing and access to the files. Many distributed file systems are developed on overlay networks, there exists PAST (Druschel and Rowstron [2001]), Ivy (Muthitacharoen et al. [2002]), Farsite (Adya et al. [2002]), Kelips (Gupta et al. [2003]), Frangipani (Thekkath et al. [1997]), CFS (Dabek et al. [2001]) and OceanStore (Kubiatowicz et al. [2000]).

PAST PAST is a large scale persistent file system. PAST is built on top of the Pastry overlay network. The files have an identifier that is a SHA-1 hash of the file name and the public key of the client that wants to store it. The file is stored in many replicas on the nodes that have the closest identifier to the one of the file. PAST allows users to perform three operations that are: storing a file replicated k times, k being a user specified number, retrieving a copy of the file identified by an identifier if the file exists in PAST and reclaiming the storage occupied by k copies of a given file. PAST is described more in detail in the Section 4.1.3.

IVY is a read/write file systems that is deployed over Dhash (a distributed HashTable deployed over the Chord overlay network). IVY is able to support multiple users concurrently. The system is based on a set of logs stored over the network. These logs reflect the changes made to the files stored on IVY. From the point of view of the user IVY provides an NFS-like file system and is able to detect conflicting modifications. it also benefits of the failure tolerance of Chord.

Farsite is a distributed file system that is able to deal with the untrusted clients. Farsite aims to provide high availability and reliability for file storage, security and resistance to Byzantine threats. Reliability and availability is ensured through replication of the whole file. Farsite has a collection of interacting and Byzantine-fault-tolerant replica groups arranged in a tree overlaying the file system namespace. File stored on Farsite are encrypted and replicated in a non-byzantine way. Digital signatures are used to prevent an unauthorized user to write a file. After encryption, replicas of the file are made and they are distributed to other client machines.

Kelips is an overlay network structured into a Distributed HashTable that achieves a fast look-up. Specifically, it achieves $O(1)$ lookup time, at the cost of memory usage of

square root N where N is the number of nodes of the network. In Kelips, each node has k affinity groups and maintains group views (entries for all nodes within the group) and and a constant number of contacts for all other groups. A node that wants to store a file computes its identifier using a hash function and sends an insert query to the group that corresponds to the computed identifier. A random node of the group is chosen to store the file. A node that wants to retrieve a file sends a lookup query to a member of the group, and then retrieving the file.

Frangipani is a distributed file system developed on PETAL system. PETAL allows a collection of network-connected servers to cooperatively manage a pool of physical disks. The pool of disks appears to the servers as a single large virtual disk. Petal provides a copy-on-write snapshot mechanism. When creating a snapshot, PETAL pauses applications briefly (for less than one second).

CFS is a file system built on top of Chord. It offers a system of reading and writing, write privileges are reserved to the user who inserted the file to CFS. The CFS files are stored in blocks, each block is replicated on multiple nodes to avoid data loss in case of failure. These nodes are nodes whose IDs are closest identifiers blocks. CFS does not offer permanent storage, a user that has inserted a file in CFS must periodically access it if it does not want that the file be deleted. In this way CFS automatically removes obsolete or not maintained data. CFS is described more in detail in the Section 4.1.3.

OceanStore is a distributed file system developed upon Tapestry (Zhao et al. [2004]. In OceanStore, each file has an identifier belonging to the same namespace as Tapestry nodes. The files are then stored on the node with ID closest to file. In fact, to avoid losing data in case of failures, multiple replicas of a file are stored on nodes with similar IDs. OceanStore supports two types of access control, restrictions on writing and restrictions reading. To prevent read access unauthorized files are encrypted and the encryption keys are only issued to users with read permissions. There is no real way to prevent a user has access rights to modify a file. But all these records must be signed and all changes to files are kept with the files. So a user can check a list of access control entries that were authorized and otherwise read a previous version of the file. The list of access control is a file stored on OceanStore which the read access is given to all users and write access are reserved to the system.

1.2.3.3 Sharing CPU cycles

In the case of sharing CPU cycles applications using an overlay network, we must distinguish if this is a participative application or not.

In the first case, it is often a single organization that disposes of large set of proprietary machines. These machines are organized in overlay networks and provide a platform to allow to perform distributed computing. Safety is ensured by the fact that the risk that a node be malicious is relatively low, since all machines belong to the same organization. However, the computational capacity of such a system is limited. Indeed, as large as possible the organization it is, it does not have an unlimited number of machines.

In the case of participative applications, there are a lot of different users who choose to collaboratively perform distributed computing. The computing capacities of these systems greatly exceeds the capabilities of non-participative systems. However, in participative

applications each user owns a node. It can act maliciously, providing wrong results in order to distort the final calculation.

Take the example of an extraterrestrial that landed on earth to prepare an invasion. It could, by participating with his computer to the SETI program, hide that there exists another forms of life and prevent the detection of the invasion. Consequently, this kind of participative overlay networks used for the distributed computation are generally endowed with reputation systems or duplication of calculation systems in order to detect erroneous results.

Once this distinction is done, we must now consider the types of overlay networks that are the most suitable for the applications dedicated to distributed computing. A distributed computing application is in general organized around a central tasks server. This server distributes tasks to other machines that have to perform calculations. These machines, once the calculation is done, returns the results to the tasks server. The server continues this distribution calculations and the retrieving of the partial results until the final result is achieved. If one considers this vision of the distributed computing, the overlay networks which will be used to developed these distributed computing applications, must have a centralized architecture. XtremWeb (Fedak et al. [2001]) and BOINC (Anderson [2004]) are two examples corresponding to this philosophy.

BOINC stands for Berkeley Open Infrastructure for Network Computing. It is on this platform that the calculations of SetiHome and EinsteinHome projects are performed. Each user connecting to BOINC must choose a project among the list of the proposed projects. The user nodes are connected to a central server (different for each project). This server can be divided into three parts, one part manages the database of users, another manages scientific data, and the third part (the largest) manages the storage and distribution of tasks that the nodes will have to perform . A node user can then request a task to the server that returns it. Once the calculation is finished, the user returns the result to the server which stores it in the scientific database. To be sure that the result is correct, a task is distributed more than once, and different results are also stored.

Another architecture used for this application is the pseudo-centralized architecture, the idea is to replace the central server by a collection of connected central servers. This is the choice of CX (Cappello and Mourloukos [2002]) and Self Organizing Flock of Condor (Litzkow et al. [1988]).

CX assigns the distribution of the tasks to a set of tasks servers. The number of servers is proportional to the number of nodes performing calculations. In addition to being resistant to failure, the status of each server must be recoverable at any time. The servers are organized in sibling-connected tree (see Figure 1.3). The status of each server is then replicated on his sibbling on the tree.

When a failure is detected by a node performing calculations or by another server tasks, it is reported to the root server on the tree or to a sibling node if the root server is down. The root server then directs the next server joining the network to the position of the failed server. During the period required for replacement the twin of the failing server is used.

Condor is a mechanism for sharing unused processor cycles. The machines joining Condor are organized around a central server, which is simply one of those machines whose role is to organize distributed computations, and they form a Condor pool. To avoid a single point of failure that represents a single central server and to allow multiple Condor pools

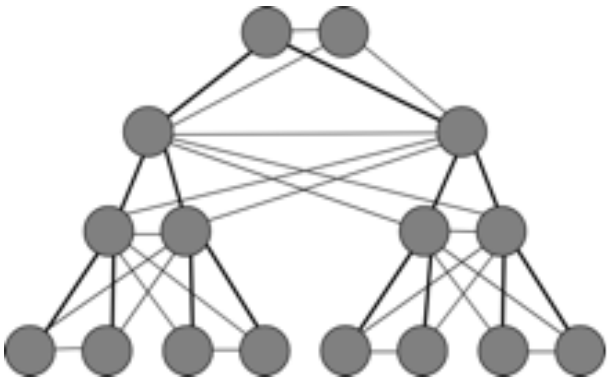


Figure 1.3: A sibling-connected tree

to share their resources and their calculations, a grouping mechanism has been proposed. This mechanism allows to combine the central servers of different Condor pools into an overlay network. However the creation and configuration of it must be done manually. To override this default, an overlay network: self-organizing Condor pools was presented (Butt et al. [2006]).

But it is also possible to share CPU cycles to different nodes of a overlay network in decentralized architecture.

N-Cycle(Bölöni et al. [2005]) for example, creates an overlay network with a decentralized architecture. This network can be viewed as a directed graph with N cycles, each node has n predecessors and n successors. Each node sends tasks to perform at its successors and receives the results of its predecessors. Thus each node receives tasks to perform from its predecessors and returns the results to its successors. Then there are two strategies described in Bölöni et al. [2006] for the distribution of tasks. In the first one, a node receiving a task executes it if it not already running another task and else sends it to one of his randomly chosen successors. In the second strategy, each node owns and maintains a "weight . This weight is calculated from its weight and resources of its successors. A node that receives a task executes it, if it not already running another, and sends it to its successor with a probability proportional to their weight.

1.2.3.4 Routing

In this group are listed applications using an overlay networks and the aims to offers a best routing than IP for some criteria. Since the overlay networks have the advantage to be able to bypass the IP routing it is quite logical to find such applications. There are currently three major challenges regarding routing:

- providing a multicast service at large extent ;
- ensuring an efficient content delivery ;
- ensuring a quality of service.

Multicast
The term multicast was created by an American doctoral student who wrote "the" thesis

on multicast, it is a contraction of "multiplexed broadcast". This is the name given to the process that allows a source to deliver information to a large number of hosts using for this purpose the most effective strategy in order not to be limited by the bandwidth of the source.

In the IP protocol, a multicast group is defined as a dynamic set of machines designated by an IP address (the address range reserved for multicast consists of IP addresses 224.0.0.1 to 239.255.255.254). When a computer wants to multicast a packet to a multicast group, it sends it to the IP address identifying the group. This package is then routed to the router that will be responsible for distributing it to members of the group. To join a multicast group, a machine must use IGMP (Internet Group Management Protocol).

The IP multicast is nowadays mainly limited to ISPs or universities, forming small multicast islands in the unicast ocean that is the Internet and it seems that the IP multicast will never be largely used.

We can distinguish two groups of overlay networks applied to multicast. In the first group, the multicast service, is called hybrid. The hybrid multicast services aim to connect the existing multicast islands together. The overlay networks in the second group provides an applicative layered multicast.

Mbone (Eriksson [1994]) is a best example of the first group. In Mbone, in each IP multicast island, a node called *mrouted* is designated. These *mrouted* nodes are then linked together to form an overlay network. When a multicast IP user wishes to multicast a packet, it sends it to the router responsible for its own island. This router then distributes the packet to all members of the group in this island. The *mrouted* node of this group receives the packet, and forwards it to another one of the *mrouted* nodes that compose the overlay network. The *mrouted* node sends the packet to the router responsible for his island and forwards it to a another *mrouted* node. The package will be IP multicasted to the members of this island. To avoid network congestion due to the presence of old packets, a Time To Live flag is inserted in the packet.

In the second group of multicast services deployed on overlay networks, the IP multicast islands are no longer used. The multicast service is then made possible by the construction and use of the overlay networks to efficiently deliver multicast information. It therefore seems judicious to choose overlay networks with decentralized architectures when one wants to develop such services.

Several protocols exist to build such networks. One approach is to assign a controller to each multicast group. This controller is then responsible for placing new nodes and construct the topology of the overlay network. Such a strategy has be chosen by the creators of ALMI (Pendarakis et al. [2001]).

ALMI is an overlay network that support a multicast service. When a node wants to join a multicast session, it contacts the session controller. The session controller adds that node in its table of participants and then designates another participant in its table to be the parent of the new node. The session controller organizes the overlay network as a tree. To ensure that this tree is efficient, a reorganization of the tree is periodically performed by the session controller. This reorganization is based on the current quality of the connections between the nodes. To obtain the information necessary for this reorganization, the controller assigns to each node the role of measuring the *round-trip delay* time between itself and some other nodes selected by the session controller. To leave a multicast session, a node

notices the session controller of its departure. The controller sets the parent of the node that leaves the session as the new parent of the sons of the leaving node. Using a controller to build and maintain the tree has the advantage of building effective trees. The controller is supposed to have a comprehensive knowledge of all nodes and their capabilities. However the detection of failures remains problematic. Indeed, the only indicators of a failure are the measurements made by the nodes. However, these measures should not be too frequent otherwise the network could be overloaded. The detection of a failure can take considerable time. The repair of failures is also problematic. In case of a detected failure, the same protocol used for the departure of a node could be used. However, in the case of too many simultaneous failures, the tree obtained could be very inefficient.

To overcome the problems raised by the centralized initialization and maintenance service, one can use distributed ones. This solution has the advantage of providing a better detection of the failures. In the other hand, the trees obtained with the decentralized algorithms are less efficient.

Overcast (Jannotti et al. [2000]), which also provides a multicast service on a overlay network, chooses this solution. When a node wishes to join a multicast group, it contacts the owner of this group. The owner is the root of the multicast distribution tree. The new node then compares the free bandwidth of the owner with the one of its son. If the bandwidth offered by a son of the root is greater than the bandwidth of the root, the new node compares the bandwidth offered by the son with those of the sons of that son. This operation is repeated until there is no more improvement. It is illustrated in the Figure 1.4

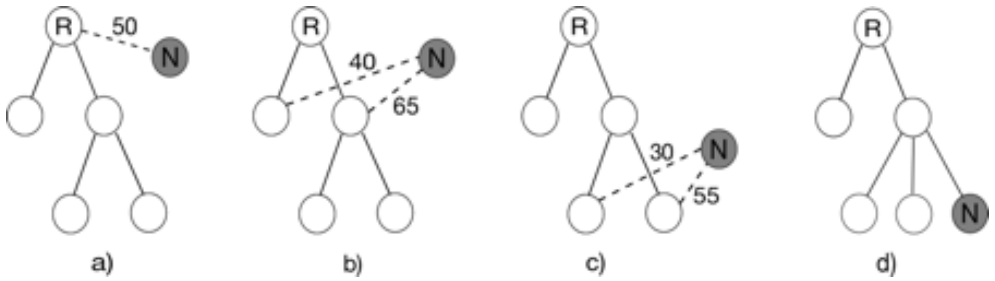


Figure 1.4: An example of constructing broadcast tree in Overcast

The last solution presented in this document, consists in deploying a multicast service over an structured overlay network with decentralized architecture and defining a covering tree among the nodes of the overlay network.

Bayeux (Zhuang et al. [2001b]) is deployed over the overlay network Tapestry. Bayeux provides a multicast service based on the above strategy. When a new node wants to subscribe to a multicast group, it sends to the root node responsible for that group a JOIN query. The root node replies to this node with message TREE containing the ID of the subscriber and the identifier of the multicast session. All the nodes through which passes the request TREE include in their multicast table, the identifier of the new node and the identifier of the multicast session. Subsequently, when the root node decides to send data to a multicast group, it sends data to its neighbors on Tapestry. These neighbors look into their routing table if they have registered users in the multicast session, and if so, they route the multicast data to users. A similar protocol is used to allow nodes belonging to

a multicast group to leave it. Nodes wishing to leave the group send LEAVE requests at the root node of the group. The root node responds with a request PRUNE containing the identifier of the node wishing to leave the group and the identifier of the multicast session. The nodes through which passes the request delete the concerned multicast table entries. This process of construction and modification of trees is summarized in the Figure 1.5

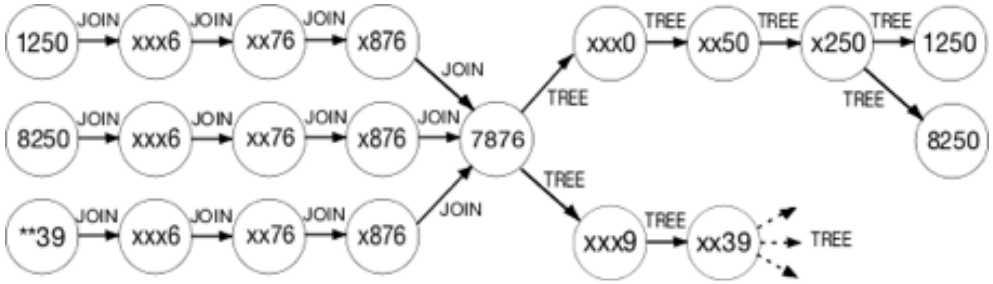


Figure 1.5: Two nodes and a group joining a multicast session

The advantage of this solution is that one has not to worry anymore about routing or detection of failures and repairs, since these services are assured by Tapestry.

Content Delivery

An overlay network that offers a content delivery service is a network where the content of a server is cached on a large set of nodes in order to ensure the most efficient and fastest delivery possible.

A content delivery service must satisfy three conditions:

- the contents must be distributed on a large set of nodes and the service must maintain the consistency of the content on the various nodes even if local updates occur.
- It should redirect a query performed by an user to the node that is in the condition to deliver the content most efficiently. It uses several factors to decide how queries will be distributed to customers (response time, proximity, closeness in the network ...).
- It must always be able to deliver content (fault tolerance).

The content delivered can be, as in the case of Globule (Pierre and van Steen [2006]) or from Akamai (Akamai [1998]), pages of websites.

Globule is a participative overlay network in which each website is hosted on a small proportion of the network nodes. Users can then share their bandwidth in order to effectively and collectively host their websites. In Globule, at each website is assigned a home node, which is usually the node of the user that owns the site. It contains the sovereign documents for the Web site and is responsible for spreading the web site on the other nodes involved. These nodes, called replicates, are strategically placed to meet the quality requirements defined in advance. These replicates are connected to the home node in such a way that they can retrieve, in case of an update, the current version of the site. For each site, the original node must be reachable at any time by the replicates. However, since this node can fail, the home node can be replaced in case of failure. For each website, a number

of nodes of the overlay network are designated as forwarders. These nodes are connected to replicates and their role is to redirect requests from users wishing to consult the website to the replicate node that can deliver the website in the best possible conditions.

Akamai is a private overlay network. It consists of servers distributed around the world. These servers contain a cache a version of the Web sites of the Akamai customers, including Apple, Logitech, Pixmania, or Reuters, and the newspaper Liberation. However, there is very few information about the inner workings of Akamai, for reasons of safety and competition, since Akamai has a commercial purpose.

The delivered content can also be a streaming video or audio content. PROMISE (Hefeeda et al. [2003a]), GnuStream (Jiang et al. [2003]) and DoNet (Zhang et al. [2005]) deliver such content. PROMISE and GnuStream allow a user wishing to attend a media stream to use an overlay network, such CollectCast (Hefeeda et al. [2003b]) and Gnutella to find a large number of users with the requested media. Once these users are known, PROMISE and GnuStream, select a subset of these users, sufficient large to allow a content delivery accurate to a predefined quality of service. Then, the user connects to these nodes and the delivery can begin. For the sake of fault tolerance, when a source does not fulfill its role properly, it can then be exchanged with another (or a group of others) user(s) owning the media.

This strategy where the media to stream is initially owned by several nodes, is opposed to the design strategy of the single source. DoNet has opted for this strategy. In DoNet, a video stream is divided into segments of uniform size. Each stream can then be represented by a "Buffer Map" which is nothing but a sequence of bits, one for each segment of the stream, set to 0 or 1 depending on the availability of stream segments. The "Buffer Map" are then exchanged between the neighbors in the overlay network. Each node has then to choose among all its neighbors which will give him the missing segments, this choice must then take into account two constraints:

- The continuous reading of the stream: in other words, when a node has finished reading a segment of the stream, the next segment must be ready to be read.
- The bandwidth of the neighbors must be load balanced: in order to have information on this subject, along with the "Buffer Map", the network nodes also exchanges information on their reserves of bandwidth.

Quality of service

Quality of service means the control mechanisms that can assign different priorities to different users and different types of data. It also means the mechanisms that guarantee a certain level of network performance. In the current context, where the capacity of our networks are limited and where the routing of packets based on the best effort routing does not give guarantees regarding the performance thereof, the quality of service is highly desired by many applications. These applications includes synchronous video conference over IP or streaming media. Indeed, the data must be sent and received on a continuous flow with an uniform and adequate rate. We can assess the quality of service according to four main criteria (there exist other ones, but they cannot be taken into account in the seventh layer of the OSI model, and consequently by the overlay networks). These criteria are:

- Bandwidth: or rather the capacity and rate of the bandwidth. This criterion defines the maximum amount of bits per time unit.
- Packet loss. This criterion defines the number of packets that are lost on the network between source and destination.
- Latency. This criterion defines the delay between transmission of a packet and its receipt by the addressee.
- The order of packets. This criterion sets, as its name suggests, the differences in the order of packets sent and the order in which they are received. This criterion may have important consequences for streaming applications.

Some solutions have been developed to provide QoS mechanisms at the IP level, including IntServ (Braden et al. [1994]), DiffServ (Nichols et al. [1998]) and MPLS. IntServ defines a number of mechanisms to support a QoS policy without affecting the operation of IP. IntServ is based on IP routers that are able to reserve resources at their disposal for data streams. There are two services offered by Intserv: Guaranteed Service and Controlled Load. The first guarantees the bandwidth and a maximum transit time. The second is equivalent to a service delivery at best effort in a not overloaded environment. DiffServ consists in a set of mechanisms proposing to ensure a QoS policy over IP networks. DiffServ applies a QoS policy to the IP packets rather than data stream. MPLS is a technology network whose primary role is to combine the concepts of IP routing (Layer 3) and the switching mechanisms (level 2). MPLS allows you to specify a QoS policy. The problem with these three alternatives is as follows: the deployment of these QoS mechanisms at large-scale requires to change a large majority of IP routers. To be efficient, IntServ must be deployed on a lot of routers of the network. If IntServ was deployed only on a few routers, these routers will reserve bandwidth as defined by the QoS policy but other routers will not do the same, and thus the quality of service can not be assured.

That is why today these solutions are only used on private networks. One have thought, since the overlay networks can specify routing, to develop mechanisms that ensure QoS at the application level with the help of the overlay networks.

There exists three distinct strategies to achieve QoS with overlay networks. The first strategy consists in finding the best possible path between the source node and destination node, and reserves capacities (bandwidth, etc..) on the nodes that compose this path. That is how QSON (Lao et al. [2006]) works.

QSON has two different types of nodes, users nodes and proxy nodes. These are connected to form a random graph. Each user node is then attached to this graph via a proxy node. A proxy node stores the capacity of the available bandwidth of each virtual link that directly connect itself to the another proxy nodes. They also store a list of paths connecting itself to other proxy nodes. To limit the amount of information that each proxy node must store, a management protocol can be applied to exclude from the lists of paths the paths that are too long. When a user wants to open a connection with a guaranteed bandwidth b between itself and another node. It first contacts the proxy node to which it is connected. Then the node communicates the bandwidth and the proxy node of the user with whom it wishes to establish the connection. Then the proxy node looks in its list of paths for the best path that is able to ensure such communication. Once this path known, it reserves on each

nodes that compose the path the necessary bandwidth. The mechanism for searching and booking a path that ensures a certain bandwidth between two nodes proxy is as follows:

- In order to find a path, the proxy node probes all possible paths. It probes the first node in the path by sending a search query containing the value of the required bandwidth and the list of other nodes forming the path to explore.
- In the case where a node belongs to several paths to test, it is possible to probe it only once by providing the sets of node of the various paths in a single search query.
- When a proxy node receives a search query, it probes the next node on the path. And so on until the query reaches the destination proxy node or that the available bandwidth of a virtual link connecting the next node in the path probed is less than what was requested.
- If the request reaches the destination proxy node, it responds with a reservation request along the same path as the search query. Each node receiving this request reserves the required bandwidth on the virtual links forming the path.
- Upon receipt of this reservation request, the initiator node proxy notifies the user that the connection is established.

One can also rely on a efficient routing mechanism. There are no controls or predefined paths. The quality of service is achieved by ensuring that the routing within the network is optimal. This is often achieved by a rearrangement of the virtual links according to the capabilities of each node. Detour (Savage et al. [1999]), QRON (Li and Mohapatra [2004]) and RON (Andersen et al. [2001]) are good examples of this strategy.

RON, which stands for Resilient Overlay Network, can find alternative more attractive routes or bypasses a failure occurring on the underlying network. As will be shown a little lower, RON can also specify a QoS. RON is composed of interconnected nodes, each node is directly connected to all other nodes by a virtual link, thus forming a fully connected graph. Each node RON actively checks the state of virtual links connecting itself to others by asking them periodically about their statements (latency, rate of packet loss and value of available bandwidth). These probes limit the scalability of RON. Today RON can handle a maximum of one hundred of nodes. At each incoming packet of RON is assigned an identifier indicating which flow this packet belongs. The nodes in RON route all the packets belonging to the same flow by the same route. When a node receives a packet with an identifier of an unknown flow, it then considers the QoS associated with the flow (it is in the header of each packet flow) and then defines the next node which has to send the packet. It notes in its routing table that packets of this flow must be sent to this node.

The last possibility is to assign to the nodes the charge of distributing the capacity of the virtual links between each data stream types in accordance with the QoS policy.

OverQoS(Subramanian et al. [2004]) opted for this strategy. Each virtual link in OverQos is associated with an CLVL (Controlled-Loss Virtual Link) abstraction. This abstraction provides a value q on the amount of lost packets per time unit. This value is computed using a combination of FEC (Forward Error Correction) and ARQ (Automatic Repeat reQuest). The CLVL abstraction also provides a value for the bandwidth c,

this value is the maximum rate above which the virtual link can not guarantee that the amount of lost packets per time unit is less than q. Nodes can then distribute the bandwidth between the different flows of applications running on OverQoS. A node of OverQoS is composed of two modules, one handling the traffic management and a CLVL module. The CLVL module determines the level of redundancy necessary to achieve an amount of lost packets per time unit less than q, and then deducts the bandwidth c. The module handling the traffic management distributes the computed bandwidth between the flows going through the node, in accordance with the priorities defined by the QoS policy. A diagram of a node OverQoS is described in the Figure 1.6.

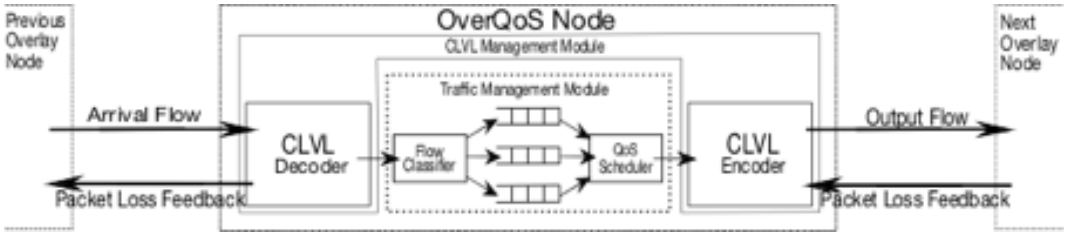


Figure 1.6: A node in OverQoS

1.2.3.5 Security

This group deals with the overlay networks and with the applications built on those networks designed to increase the underlying network security. There is three main security objectives: confidentiality, availability, integrity. The essential security functions of the overlay networks are the functions relating to identity, hidden identity (anonymity), proof of identity (authentication), identification and traceability of this identity by functions of observability (observability).

Confidentiality

The International Organization for Standardization (ISO) has defined confidentiality as the assurance that information is available only to subjects with permission. Confidentiality is usually implemented by cryptographic mechanisms, using symmetric algorithms (DES, AES) for files or long messages, or asymmetric algorithms (Diffie-Hellman and RSA) for short messages.

To create a service upon an overlay network that increase the confidentiality of the underlying network one must take into account the characteristics of the underlying network. Indeed, establishing the confidentiality of the information circulating on the virtual links, or in other words the encryption of this information often relies on a public/private key mechanism which occurs at a lower level than the overlay network. Consequently it seems difficult to increase confidentiality by an overlay network. Nevertheless there exists some methods to do so.

The first method is the stochastic routing (Bohacek et al. [2002]), which consists in randomly routing the data packets in the overlay network. Then, the attacker is not able to identify the used virtual links. The second method uses a routing protocol based on a protocol resistant to to Byzantine attacks (Malkhi et al. [2001]) This solution assumes that the attacker can listen to only a bounded number of virtual links, let m be this bound.

Then when one want to send a message, it transforms it into $m + 1$ different encrypted parts in such a way that all the parts are necessary to decode the message. The $m + 1$ encrypted parts are routed over the overlay network using $m + 1$ distinct routes. The attacker is no longer able to access information. From a practical point of view these two methods can use some virtual links that can actually share the same physical links on the underlying network. If the attacker knows where are these physical links, it may increase its chances of obtaining the desired information.

Availability

In a computer security context, the system availability is the ability of this system to provide the service for which it was designed. The main attacks against the availability are the attacks called denial of service (DoS). These attacks aim to degrade the performance of parts or of the whole the network by monopolizing its resources (CPU cycles, bandwidth, disk space, etc..). The DoS attacks can be initiated by a single entity. However, the power of these attacks mainly depends on the resources of the attacker. Nobody can today, with a single computer, launch a DoS attack against an Apache server by sending a huge amount of queries to the overload. The power and the bandwidth of the server surpassing by far those of the attacker. It is much more common nowadays to see these attacks using a large set of coordinated computers. These attacks are called distributed denial of service (DDoS). There exist some overlay networks that protect a target against this type of attack. They use two different strategies.

The first strategy is to hide the potential target of the DDoS attacks behind an overlay network. The access to the service offered by the target is then carried through the overlay network. This strategy is one that has been chosen by SOS (Keromytis et al. [2002]).

SOS is an application, deployed over Chord, that acts as a filter around a potential target. The packets from the DoS attack are ignored and only legitimate packets can pass through the filter. This filter is composed of servers with large capacity. These servers are configured to only accept packets from a small subset of nodes in the overlay network. The IP addresses of the servers are kept secret. Users are not able to contact them. To reach the target, users must first have access to the overlay network. To do this, users should contact a SOAP (Secure Overlay Access Point) node. The SOAP nodes are a small subset of nodes in the overlay network. They are responsible for receiving and verifying packets that are not yet registered as legitimate. Only the traffic coming from SOAP nodes is allowed on the overlay network. The SOAP nodes acts as a distributed firewall. When a packet is declared legitimate by a SOAP node, SOS adds the hash of the IP address of the target as the key to the packet. The packet is then routed using the routing protocol of Chord to the node whose identifier is closest to the key. This node is called beacon node. To continue to operate despite the failure of the beacon node, several beacon nodes are obtained using several different hash functions. The beacon nodes are the only nodes that knows the IP addresses of the secret server. The use of a service protected by SOS is described in the Figure 1.7. An user connects to a SOAP node, authenticates and then sends packets to the SOAP node. These packets are then routed using Chord until the packets reach a beacon node. The beacon node sends the packets to a node secret servlet, that sends the packets to the target.

Another solution is proposed by RON and Rewire (Bu et al. [2006]). These two overlay networks are designed to withstand fault caused by DoS attacks.

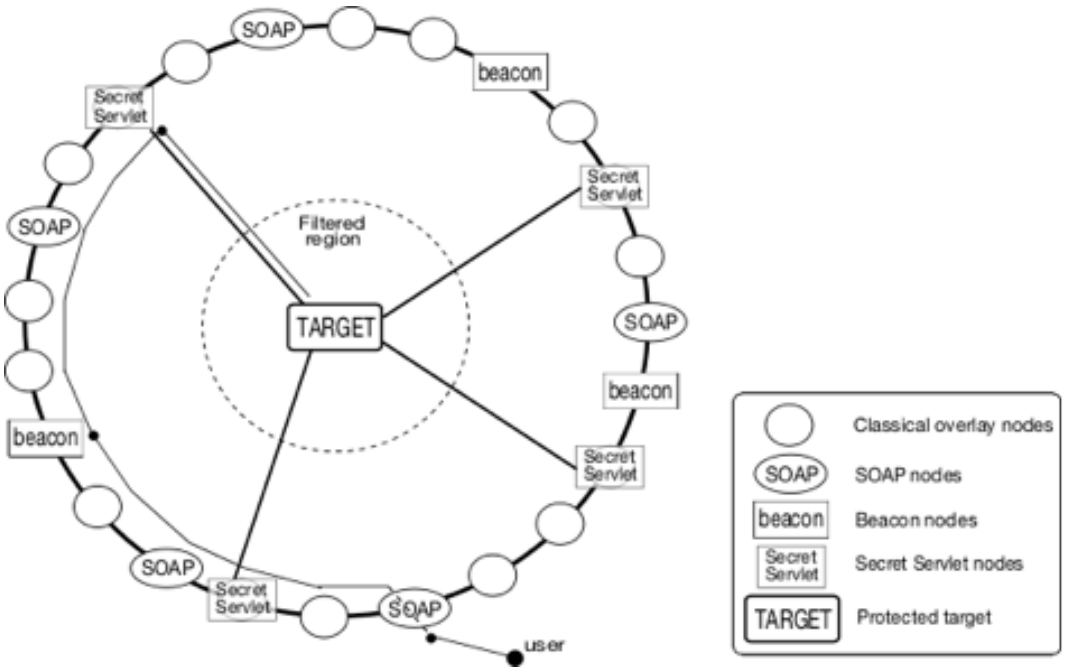


Figure 1.7: Using SOS

Rewire as SOS is designed to protect a server from DoS attacks. Three types of nodes composing Rewire:

- The gateway nodes that have the same role as the SOAP nodes of SOS.
- The target nodes are the only ones who know the IP address of the protected server.
- The router nodes that route packets from gateway nodes to target nodes

To be as efficient as possible, Rewire must satisfy three conditions:

- The router nodes must form a strongly connected graph. When a virtual link failure is detected, the router node that detects the failure must create a virtual link with another router node.
- The router nodes route the packets until the target nodes. Each router node must select the best path possible.
- Rewire must be scalable and can be deployed on a large area.

The first two conditions can be satisfied by collecting information about the overlay network topology and about the status of nodes and virtual links. This information are obtained, as in RON, by probing the nodes and the virtual links. However, the authors of Rewire then propose a more elaborate way of probing than RON. Instead of probing each nodes at regular time interval, the nodes in Rewire probes their nodes and the virtual links according to their frequencies of use. The idea is that the fluctuations on the most used virtual links have more impact on performance that fluctuations on the other virtual links.

To determine the frequency of probes, the nodes are organized in a hierarchical binary structure (see Figure 1.8), the rank of a node in this hierarchy determines its frequency of probe. So, if for each node the total number of other neighbors is equal to N the number of nodes probed at each time interval is $\lfloor log(N-1) \rfloor + 1$ for Rewire and $N-1$ RON .

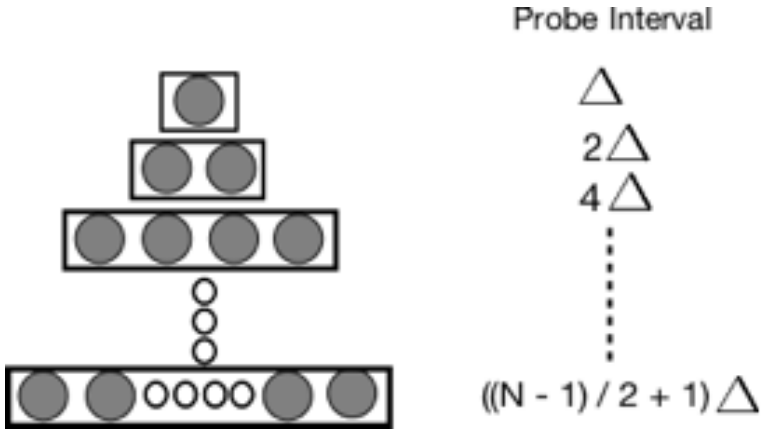


Figure 1.8: The probes interval in Rewire

Rewire combines the advantages of SOS and RON. Indeed, just as SOS, the protected server can no longer be the target of DoS attack because its direct IP address is unknown. The only potential target is the overlay network acting as filter around the target. But as RON, Rewire is resistant to failures, limiting the impact of a DoS attack against the overlay network.

Integrity

In the computer security context, the term integrity refers to both the integrity of the system and integrity of the data circulating over the system. A system where data integrity is a system where:

- The information circulating on the system are complete.
- The information are not modified, regardless of any operations performed by the system.
- The information is protected from malicious intentional modification.

The solution the most often used in networks to ensure the data integrity consists in digitally signing the data circulating on the networks. It does not prevent any attacker to alter the data, but any modifications invalidate the signatures and therefore reveal the fact that data integrity is no longer maintained. Some Message Authentication Code are also used, but these codes also only allow a verification of the integrity. Moreover, these methods are based on the use of a third party, or this third party becomes a single point of failure of the network (this can be an issue in the case of DoS attack). DSO (Gu et al. [2006]) has been created in order to remove this single point of failure.

DSO stands for "Dependable Signing Overlay", it is an overlay network that acts as a third party that provides a digital signing service. DSO is resistant to DoS attacks because

it has an architecture similar to the architecture of SOS. There are four kinds of nodes in DSO, the routing nodes, the access point nodes, the beacon nodes and the SH nodes. DSO users only know the IP addresses of the access point nodes. All the requests for signature must be sent via the access point node. DSO can manage multiple signature services, each of these services are designated by a service key.

Any services of DSO must first be initialized. To do this, m routing nodes are selected to become the SH nodes of this service. The SH nodes share, using the method of the secret sharing of Shamir, the private key of the service. Each SH node finally selects a small number of other routing nodes, elects these as SH nodes and gives them their portion of the private key of the service. This replication process enables the service to be always available even if a SH node fails. The SH nodes warn the beacon nodes that they are the SH nodes responsible for this service. Once the service is initialized, an user can send a request via an access point node. The request contains the data to sign. This request is then routed as in SOS to a beacon node. The beacon node will forward the request to SH nodes in charge of this service. When a SH node receives a request from a node beacon, it uses its key portion of key to generate a part of the signature. The SH node sends this part of the signature to the beacon node. The beacon node, once in possession of all the parts of the signature, build the final signature and send it to the user that initiates the request.

A second approach consists in basing the routing of the overlay network on a tolerant Byzantine attacks protocol (Malkhi et al. [2001]). This solution assumes that the attacker can only a modify the data circulating over a predetermined maximum number of virtual links of the network. Let m be this number. If one wants to ensure the integrity of the data, it duplicates it into $2m + 1$ identical copies. These copies are sent over the overlay network. The attacker can then change only m copies. To be sure of the integrity of the data, it only requires to have $m + 1$ copies.

A system with integrity can be defined as a system where none of the elements of this system is corrupt. An elements is corrupt if it is controlled by an attacker or by an unauthorized entity. Preventing such an intrusion is what the firewalls aim to do. There exist alternative ways to prevent an attacker from entering a computer networks. But the efficiency of the solution that aims to prevent the intrusions is only relative and is often obtained at the expense of freedom of the authorized users. But if we can not efficiently prevent an intrusions, we can at least try to detect. It is shown in Dnad [2004] that the intrusion detection can be done efficiently only in a distributed way.

DOMINO (Yegneswaran et al. [2004]) is an overlay network that was designed for this purpose. Three types of nodes composing DOMINO, the Axis nodes, nodes the Satellite nodes and the Terrestrial nodes. The first nodes are responsible for sharing information regarding the detected intrusions. Each Axis node maintains updated:

- A database of malicious activities: the database keeps the logs concerning the packets, the global and local summaries, as well as the vulnerabilities and the alerts ;
- An Active Sink: A sinkhole is a large group of unallocated IP addresses. The Active Sink oversees the traffic sent to the IP addresses of the sinkhole and simulates virtual machines by providing a certain level of interaction. The Active Sink reviews the packets with the intention of revealing attacks associated with specific vulnerabilities.

- A Firewall: This firewall provides security rules for the traffic to accept or not, and a large number of attack signatures.

The Axis nodes periodically exchange information on intrusion attempts. These information packets are called summary. Each node Axis retrieves information from the Satellite nodes to which it is connected. It combines these information into summary and multicasts these summaries to the other Axis nodes.

The Satellite nodes are nodes that implement a local version of DOMINO. They are hierarchically organized. The information sent by these nodes are considered less reliable than those from Axis nodes.

Terrestrial Nodes are the largest source of information. However, they are very unreliable because these nodes do not implement a version of DOMINO and use their own resources to collect information. They are attached to the DOMINO network through an access point. The DOMINO architecture is summarized in the Figure 1.9.

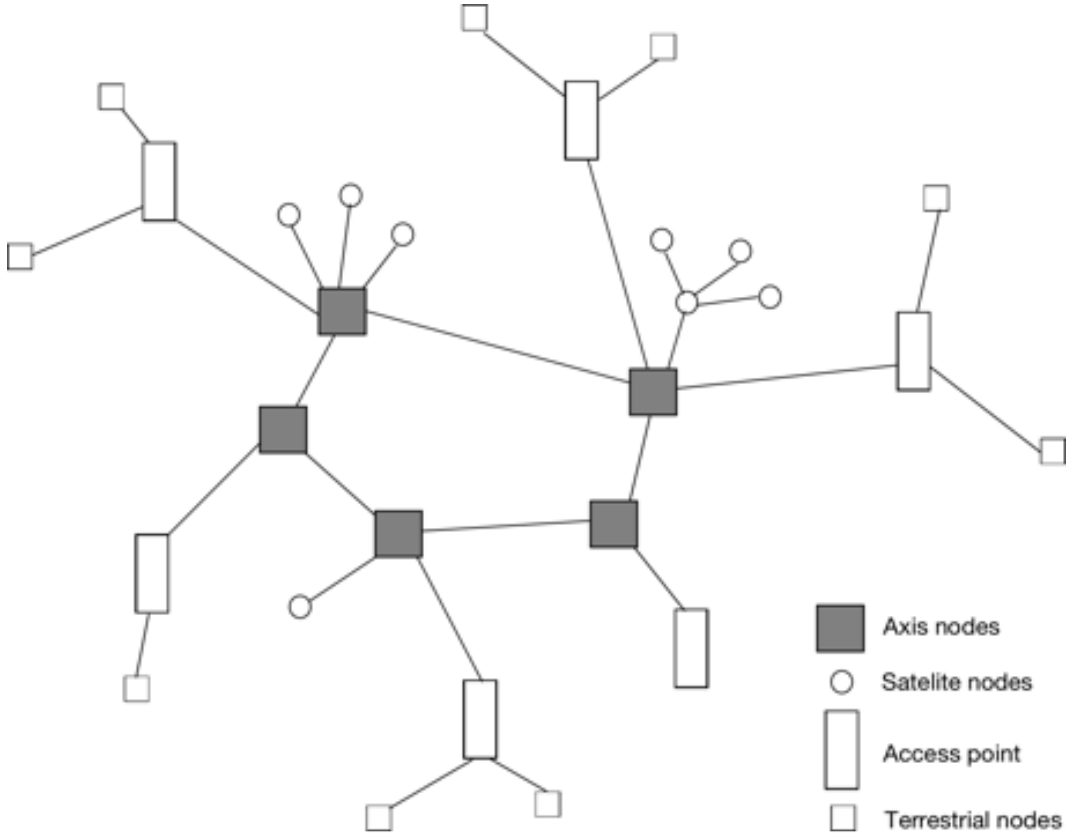


Figure 1.9: DOMINO

DOMINO is not the only overlay network used for intrusion detection, Indra (Janakiraman et al. [2001]) was also designed to detect intrusion.

Indra is an overlay network that provides network intrusion detection and prevention. It is built on top of the Pastry overlay network. Each node runs an Indra daemon. This one listens and looks for suspicious activity such as multiple failed login attempts, port-scan or suspicious system-call sequences. When a suspicious activity is detected by one node

of the overlay network, it multicasts an alert to all the other nodes using Scribe. Scribe is a publish/subscribe system developed on top of the Pastry network. To avoid that an attacker can launch a false alert, in the prototype version Indra relies on trusted certificate authorities. But authors think that it would be more realistic to use a Web of Trust model to determine if an alert must be considered serious or as a fake one. Once a valid alert is propagated, it belongs to Indra nodes to decide which solutions will be employed. Solutions can range from paranoia to indifference.

Authentication

Authentication is a security feature that allow an entity to convince others that it has secret information sufficient to prove his identity. It is usually done through cryptographic protocols and algorithms. The public key certificates are the basis of most authentication mechanisms on computer networks. The public key certificates or identity certificates are certificates that use a digital signature to associate a public key to the identity of an entity. The certificate can then be used to verify that the public key belongs to an entity. Authentication can prevent attacks like man-in-the-middle attacks. Usually these certificates are provided by a third party, therefore, becomes a single point of failure in the network. One can imagine, eliminating this weak point by using DSO or another overlay network that offers a digital signature service.

Another possibility is to interpret the authentication on a network such as the continuity of a relationship between entities or as it is formulated in Schneier [2004] to "know who to trust or not, this new definition makes the concept of system reputation and trust management, these concepts will be discussed in more detail later in this document.

Anonymity

The anonymity in computer security means many things. If we use the term anonymity about a communication network, this refers to the anonymity of the source and destination. In other words, nobody can guess, by listening to the communication, from where it originates and to whom it is intended. If we use the term anonymity about a publications/consultations service, it refers to the identity of the publisher of a document and the identity of users consulting this document. In addition to these different means of the term anonymity in computer science, there are different degrees of anonymity, partial or absolute anonymity. The partial anonymity means that there is no possibility of breaking this anonymity directly, but that it is possible to collect a little more information each time that anonymity is used. Let us consider the example of an entity that wishes to hide the identity of the recipient of a message. It can send the message to set of addresses that includes the recipient address. The attacker may not know who the actual recipient is, but still knows that the address of the recipient is among the set of addresses. With an absolute anonymity, in contrast to the partial anonymity, an attacker cannot obtain any information about the identity of the source or the recipient.

It has already been presented in this paper some overlay networks that offers anonymous publications/consultations services (Freenet and Free Haven). It will now be presented an overlay network that offers an anonymity service Tor (Dingledine et al. [2004]) and Cashmere (Zhuang et al. [2005]).

Tor consists of a set of nodes, called onion router. Each onion router is connected via a virtual link with each other "onion routers". Tor can be seen as a fully connected graph. Each user node, called onion proxy is connected to one or several onion routers. The onion

proxy role is to form circuits in the graph formed by the onion routers through which the flows anonymously go. The creation of a circuit is costly in time and cycles calculation, it requires successive ciphers based on public key algorithms. Each onion proxy creates in advance several circuits. And to limit the chances for an attacker to break the circuit when anonymity is used many times, it is abandoned and exchanged for another. Meanwhile, a new circuit is created to replace the abandoned one. To open a TCP connection with an anonymous host at a given address and a given port, a user asks his onion proxy to connect instead of itself. The onion proxy selects the last open circuit (or create one if necessary) and designates the best suited onion router for the circuit output. Usually it is the last onion router in the circuit. Then the onion proxy opens the anonymous stream by sending a cell "relay begin" to the onion router designated for the output (one cell is a message containing the identifier of the circuit, identifying the flow and figures). The encrypted data of this cell contains the IP address and port of the host to contact. The onion router designated to be output decrypts the encrypted data contained in the cell, and contacting the host with corresponding to the IP address and the port contained in the data. Once the connection is done the onion router sends a cell "relay connected" using the circuit to the onion proxy of the user. The onion proxy sends a request “SOCKS” to the user to notice him of the the successful establishment of the connection. From there, all requests to be sent anonymously to this host will be sent in the same way. The Figure 1.10 illustrates the anonymous communications between a user and a website through a circuit composed of two onion routers.

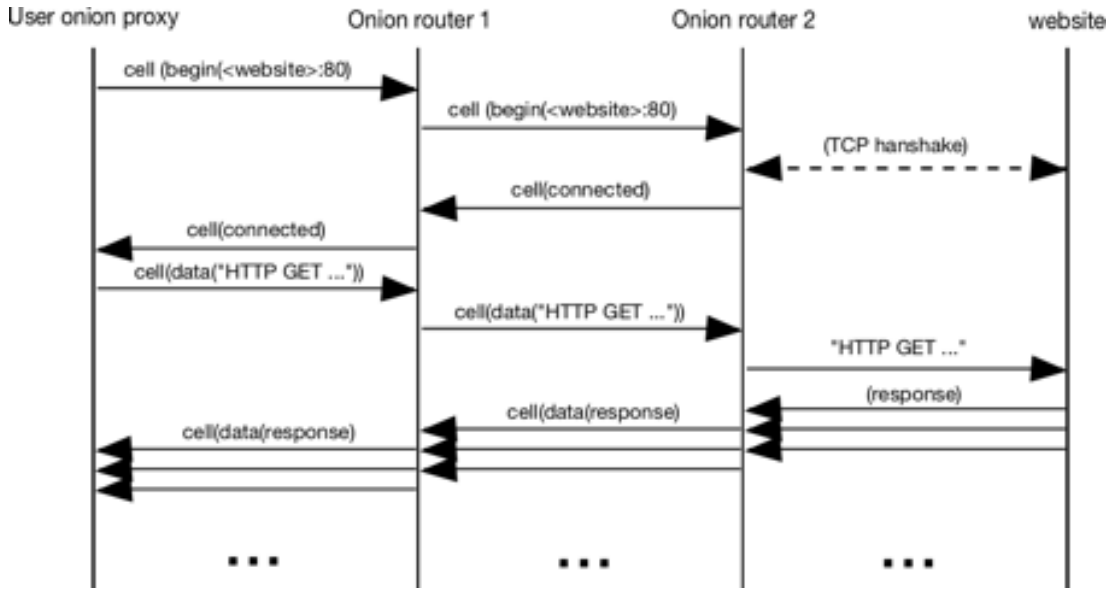


Figure 1.10: A user visiting a Web site anonymously through Tor

The anonymity of the user is guaranteed, as neither a attacker nor the contacted host can access to the IP address of the user because the host is contacted by any onion router. All the communications are encrypted: the only means by which an attacker to obtain information about the identity of the user is to study the traffic to find a relationship between circuits and the onion proxy user. However his chances of achieving that are

reduced because the circuit changes are frequent.

Cashmere is a resilient anonymous routing overlay network. It is designed to provide both source anonymity and unlinkability of source and destination (and also payload confidentiality). Unlinkability means that even if one can determine that a node participate to a communication process, it is not possible to determine if this node is the source, the destination or only a simple routing node. Cashmere affirms to be able to fulfil all these requirements even if an attacker controls some nodes in the network and even if these nodes are able to collude and share all information such as private keys. In order to do so, Cashmere uses a set of nodes that act as a virtual relay, instead as using a single node at each hops as most of the other overlay networks, these sets of nodes are called relay groups. In this perspective a forwarding path consists not anymore of a sequence of single nodes but of a sequence of relay groups, since all the members of a relay group share a common public/private key pair, it is possible to decrypt the forwarding path information for a message and so to forward it to the next relay group. These public/private key pairs are assumed to be distributed offline by a certified authority. Relay groups are anycast groups and the forwarding of a message is analogous to anycasting it to the next relay group. Cashmere is built on top of a Pastry overlay network, and the dynamic creation and maintenance as well as the routing between relay groups are delegated to it. One have to remember that nodes in Pastry overlay network possess an unique ID that is a sequence of k bits, then each relay group also has an unique identifier that is a sequence of m bits, with m smaller than k, a node is member of a relay group if the relay group identifier is a prefix of its identifier. Since nodes identifiers are randomly assigned (by the hash of their IP address), relay groups are random subsets of the overlay nodes. When a node wants to route anonymously a message to another node that is not in its own relay group, it selects a random sequence of m relay group identifier and randomly insert in it the relay group of the destination node. These identifiers are then used to construct the forwarding path. Since the source node randomly selects the forwarding path this one cannot be predicted by others. Since the relay group of the destination node is randomly inserted, an attacker may at most determine which are the nodes that will receive the message. The source node n encrypts the forwarding path in multiple layers using the public key associated to each relay group, and then sends the message to the first relay group in the forwarding path. When any matching with the current prefix receives the message, it uses the private key of the relay group to decrypt the next relay group of the path, and then routes the message to it, and so on until the message reaches the end of the path. To avoid that any nodes in the routing path may read the content of the message, the source node also have to encrypt it in order to obtain:

$$Payload_i = \begin{cases} < Payload_{i+1} >_{Ri} & 1 \leq i \leq L \\ < M >_{PubKeyB} & i = L+1 \end{cases}$$

Where L is the number of relay groups, Ri is a symmetric key shared by the source node and the ith relay group, these keys will be added to the encrypted forwarding path. This solution provides payload confidentiality; however there is no guarantee that when the message reaches the destination node relay group the messages will also reach the destination node, so to ensure that, in a relay group the first node which receives the message in addition to path decryption and message forwarding this node also has to broadcast the

payload to its entire relay group.

Residual group
There is four main groups of applications of overlay networks. However, any distributed application can be implemented on an overlay network. It is therefore not surprising to find other applications do not fit into any of the groups previously defined.

Venus (Klara et al. [2004]) is an overlay network that offers an audio conferencing over IP service. Venus tends to distribute the voice mixing service to a set of nodes. This voice mixing service is usually centralized of the conferencing systems. The nodes participating in a conference are divided into groups according to the "node gateway" by which they joined Venus. The "nodes gateway" elects in each group, a node as the "mixer node". The "mixers nodes" are interconnected to form a path. Each node participating in the conversation is linked to his "mixer node" respectively. The role of "mixer nodes" is to mix the signals received and then send the results to other "mixer nodes" Their role is also to spread the communications to the nodes in their group. The functioning of Venus is shown in the Figure 1.11.

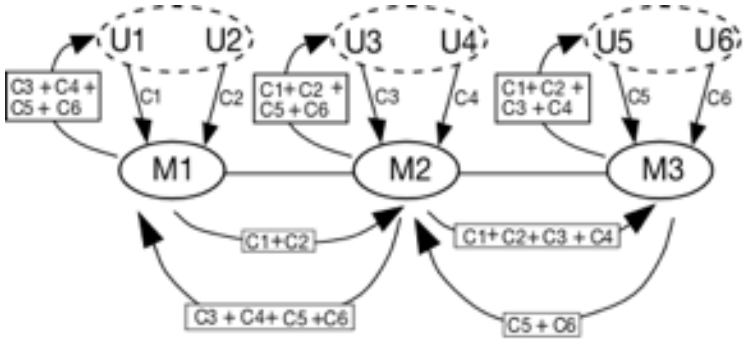


Figure 1.11: An audio conference on Venus with six participants (the "+" means the operation of mixing)

Twinverse (Twinverse [2008]) is a virtual world deployed over an overlay network with a decentralized architecture. Twinverse is at the beginning empty. Only users, creating entities and the hosts on their own machines, fill Twinverse. Its distributed architecture allows Twinverse, according to its creators, to handle 10 million users. Twinverse provides a display of 2D avatars, and allows users to communicate and share media. In the Figure 1.12 is a screenshot of what users can see in Twinverse.

1.2.3.6 Conclusion

Now that the various groups of applications of the overlay networks are defined. It is interesting to see what kinds of overlay networks is more suitable for the development of each class of applications. The relationships between the types of applications and their overlay network are summarized in the Table 1.3.

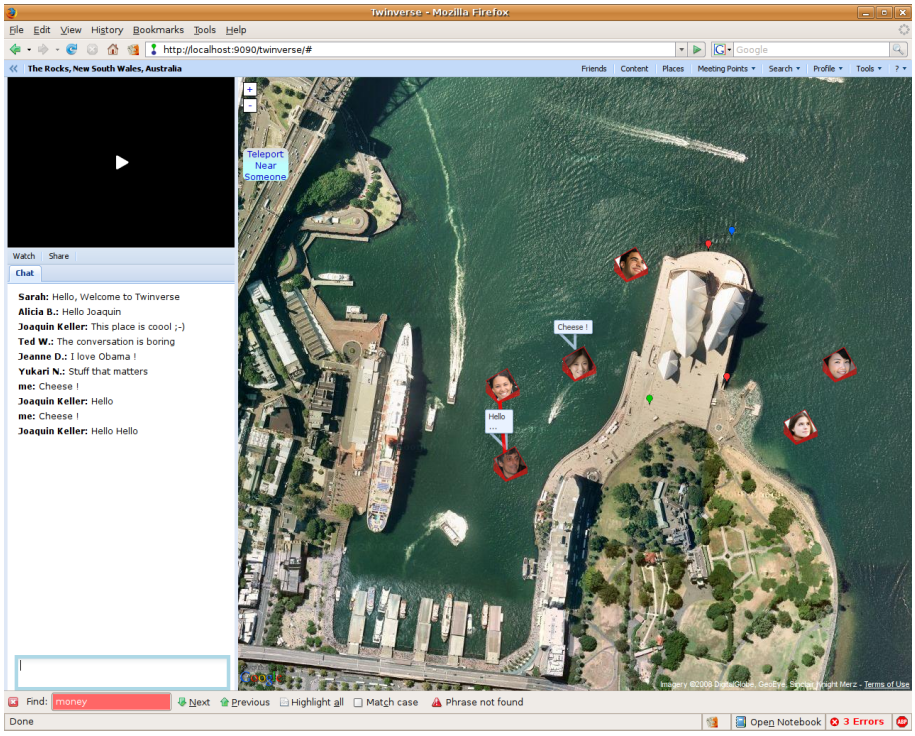


Figure 1.12: Twinverse screenshot

1.2.4 Security of the overlay networks

It is first important to realize that most attacks against overlay networks are against the participative overlay networks. Indeed, in these networks, each user owns a node. Adding a node is generally an automated process and therefore nothing prevent an attacker to insert a malicious node it the overlay network. This section will therefore focus mainly on security issues raised by the participative overlay networks. It will first be presented attacks using flaws in the initialization and maintenance services of the overlay networks to take control of part or of the whole network. Then it will be presented several attack patterns of using the flaws in the endogenous routing services of the overlay networks.

1.2.4.1 Attacks manipulating the topology

Partitioning the network

In the overlay networks with decentralized architectures, the mechanism the most often used to enable a node joining the network is to contact a node that is already on the network and retrieving the necessary information to join the overlay network. A malicious node can give information to the new node, in order to make the new node join only a part or a parallel version of the overlay network. Attackers can then use the control they have on the parallel network in order to gather information about users for example. Furthermore, if another user tries to join the network through the node that was previously abused by attackers, this new user also join the network under the control of attackers. This type of attack creates a partition of the network, this partition is completely under the influence

<table>
<tr><th colspan="2">Applications</th><th>Overlay networks</th></tr>
<tr><td colspan="2">Indexation</td><td>Indexing Applications are used to index the information held by the users on a overlay network are deployed by definition over participative overlay networks. The first generation of these applications were based on network with centralized architectures. Today it is the networks with structured and decentralized architectures that are chosen for this type of applications.</td></tr>
<tr><td rowspan="2">Resources sharing</td><td>free disk space</td><td>The best materials for such applications are structured and participative overlay networks with decentralized architectures.</td></tr>
<tr><td>CPU cycles</td><td>These applications are often developed over participative overlay networks with centralized architectures. But there are participative overlay networks with decentralized architecture that supports such applications</td></tr>
<tr><td rowspan="3">Routing</td><td>Multicast</td><td>It is very difficult to define what type of overlay networks most likely to support a multicast service, as most of overlay networks supporting such services are hybrids. We can still say they are largely participative ones.</td></tr>
<tr><td>Content Delivery</td><td>Delivery Networks refers to decentralized architecture. There are as many content delivery applications deployed on participative overlay networks as on non-participative overlay networks.</td></tr>
<tr><td>QoS</td><td>Non-participative overlay networks are often used to support such applications</td></tr>
<tr><td rowspan="5">Security</td><td>Confidentiality</td><td rowspan="5">The diversity of the overlay networks used to support applications designed to increase network security is too large to define a best type. Nevertheless, the overlay networks must be non-participative to prevent an easy takeover of a node by an attacker. These overlay networks must have a decentralized architecture to avoid have a single point of failure.</td></tr>
<tr><td>Integrity</td></tr>
<tr><td>Availability</td></tr>
<tr><td>Authentication</td></tr>
<tr><td>Anonymity</td></tr>
</table>

Table 1.3: Relations between type of applications and types of overlay network

of the attackers. There are no miracle solutions to counter such attacks. One solution is to join the network only via trusted nodes. The problem is to determine which nodes can be trusted. These nodes are known to via a public list as in the case of Kademlia, or referenced by the server as in the case of Gnutella. It is also possible for the nodes that have already been connected to a network and being "satisfied" of the network during the last connection to keep a list of trusted nodes. These nodes will serve to connect to the network a possible next time. This solution, however the disadvantage of being of no help in a passive attack. Indeed, nothing prevents a sub-network under the control of attackers from operating normally if the attackers just, for example, passively collect information

about the users. Another solution consists in creating multiple nodes via several different nodes on the network, then to launch on each of these nodes the same query and then compare the results. If the result coming from a node differs greatly from results obtained with the other nodes, one can suspect this node belongs to part a corrupt partition of the network.

Strategic positioning on the network
This attack takes place only in structured overlay networks. In a structured network, the position of a node in the topology of the network depends on its identifier. An attacker can strategically place nodes under his control into strategical point in the topology of the network by choosing adequately the identifiers of the nodes under its influence. This could allow an attacker, within the context of resources sharing application over a structured network, to take control of a resource. On most overlay networks, the identifiers are created using a hash function applied to specific data nodes and verifiable by other network nodes connected to them (IP address, port, etc..) the attacker can not directly choose its identifier. It may, however, in these cases modify its own data to obtain the wished identifier. However, this research may be fastidious and empirical or it would mean that the hash algorithm has been broken. In most cases using a hash function allows to verify that a node is the "owner" of the identifier it has on the overlay network. Another possible solution is to entrust the distribution of identifiers to a third independent party. However, that third party then becomes a single point of failure in the network and contrasts with the philosophy of the fully decentralized architecture.

Sybil attacks (Douceur [2002])
Sybil attacks are attacks where a malicious node fakes having multiple identities and pretends to be several distinct nodes. This allows the attacker to strategically position its node in many different point of the network topology, or to gain importance in a reputation or trust management systems (such systems will be discussed later in this document). A simple and obvious solution is to use a central authority that controls the allocation of identities by checking that they are associated with only one node. For example, it might be required for the registration number identifying the person as the card number or identity of the social security. This solution is not scalable, it only applies in the case of small overlay networks where the central authority can maintain the authentication data. For large overlay network, a solution is proposed in Douceur [2002]. This solution is based on the idea that two distinct nodes are at different distances from other nodes in the network, these distances can be obtained for example by calculating the round trip delay time. However, the node with multiple identities may delay responses for some of their identities in order to change the distance and therefore mislead the detection system.

Eclipse attacks (Singh et al. [2006])
Eclipse attacks are attacks where the attackers takes control or creates a large number of nodes in the network. Attackers can then use these nodes, if they are strategically positioned, to obscure a part of the overlay network by not forwarding anymore the query from the other users for example. The part of the network becomes invisible to the non malicious users.

A solution to protect overlay networks against these attacks was presented in Castro et al. [2002]. This article recommends the use of constrained routing tables (CRT). These tables impose strong structural constraints on all neighboring nodes. Identifiers of the

nodes are chosen randomly and the set of neighbors of a node is composed of nodes whose identifiers are closest to the identifier of the node. This method has the advantage to prevent an attacker to strategically place its nodes. It has the disadvantage of reducing the flexibility of the network. Another solution was presented in Singh et al. [2006]. This solution is based on a study that shows that the eclipse attacks are only effective if they are launched on a network with nodes having a high degree of connectivity. The solution proposed consists in limiting the degree of connectivity of the nodes.

1.2.4.2 Attacks manipulating routing

Shortcut attack

This type of attack is only possible on overlay networks that use a flood based routing algorithm. This attacks are mainly launched on unstructured networks with decentralized architecture. In such networks, a message is sent by flooding to all nodes in the network. However, the destination node only replies to the first copy of each message it will receive. If an attacker know which path the reply will go through will may sabots the path by controlling or causing an failure on a node that composes the path. The reply will not reach the node whose it is addressed and no further response will be sent because the various copies of the message will be ignored thereafter. A shortcut attack is represented in the Figure 1.13.

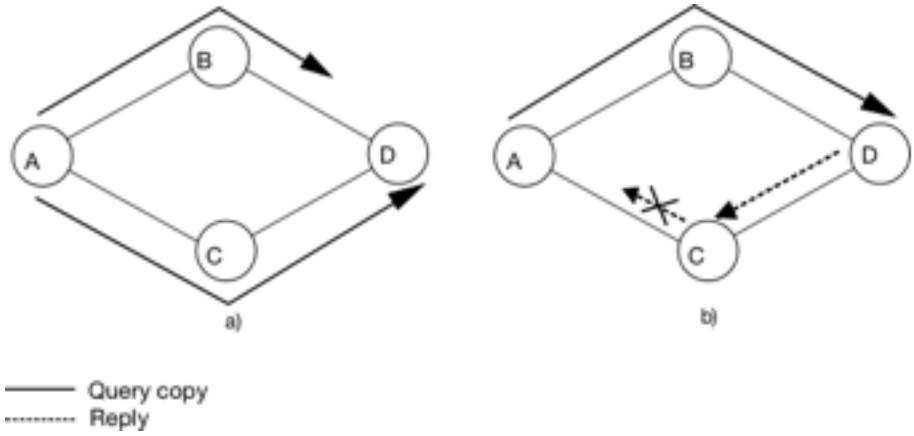


Figure 1.13: A shortcut attack

One solution to counter this type of attack is proposed in Misha [2003]. This paper proposes Cascade, an overlay network that introduces the concept of recursive and iterative routing. The recursive routing is the type of routing the most commonly used in the overlay networks. A node wishing to send a message, using the recursive routing, sends the message to one its neighbors, that neighbor will send it to one of its neighbors, and so on until the message reaches the destination node. A node wishing to send a message, using the iterative routing, try to obtain the underlying network address of the destination node using the overlay network. Once the node has the physical address of the destination node it sends the message using the underlying network. The combination of routing and iterative routing by flooding can therefore ignore the failures caused by malicious node, as there exists with high probability, still uncorrupted path between two nodes in the network.

Man-In-The-Middle attacks
Man-In-The-Middle attacks is the name given to the attacks where the attacker node is placed between two nodes of the network in order to intercept their communications and/or act on the communication. The attacker may, if the communication is not encrypted, eavesdrops the communication. The attacker also may, if no authentication processes are used, usurps the identity of one of the interlocutors. The overlay networks with decentralized architecture are vulnerable to this kind of attacks since with distributed routing protocols a large number of nodes act as intermediary. One solution to prevent a malicious node to eavesdrops a communication between two nodes is to encrypt the communication. However, if the encryption keys are exchanged over the overlay networks the attacker may intercept the keys or provide wrong encryption keys. The best solution would be to encrypt communications with keys distributed by a third party.

Attacks on the routing tables
In participative overlay networks with decentralized architectures, each node has to build its routing table with other network nodes. A malicious node can attempt to corrupt the routing table of the other nodes by providing bad information. A node with a corrupted routing table may unintentionally spread bad information extracted to its table to the other nodes In overlay networks where the routing tables are built according to the proximity of the nodes in the network topology and where the attackers are able to place their nodes where they wish, this attack can target only a part of the network and makes its detection more difficult. The attacker only needs to position its nodes in the part of the topology that it wants to compromise and since the tables are built based on the proximity of nodes, it may target the nodes that will have their routing table corrupted. A solution for a node to counter this attacks is to check each entry in its routing table. In CAN for example, a node has to verify that there is not two areas that overlap.

Selective forwarding attacks
Selective forwarding attacks are launched by one or more malicious nodes of the overlay network. In these attacks, the attackers do not properly forwards the messages that pass through their nodes. They can create "black holes" that no messages can not escape. The more node the attacker controls the more important the "black hole" is. With a sufficient number of nodes, the attacker can paralyze the whole network. A more advanced version of this attack consists in blocking only the messages coming from one or more target nodes. This more advanced version is much more discreet. If the “black hole” is close to the targeted node, or, if the attacker ensures that the traffic destined to the targeted node passes through the "black hole", the attacker can completely isolate the targeted node. A simple solution to avoid this kind of attack is to repeatedly sent the same message through different paths. The probability that the "black hole" intercepts all the copies of the message will be decreased. A second solution can be applied to the more advanced version of the attack, it is to prevent the attacker to distinguish the messages. If the attacker cannot determine which node is the source of the message, he can not decide if he must let the message pass or not.

1.2.4.3 Attacks against applications

Poisoning attacks
Poisoning attacks consists in deliberately, distort the index in the cases of the indexing applications or falsifying the data or the result of a calculation on the case of sharing resources applications. An example of a poisoning index attack, is the case of some companies that wants to fight against illegal sharing of songs on their catalog. These companies may join the overlay network and fills- the index with wrong information. The users that want to download the song will look in the poisoned index, and obtain some information that do not allow them to download the requested song. An example of a poisoning resources attack, is the case of the distributed computing. The attacks consists in returning a wrong answer to a calculation For this type of attack by poisoning, using a reputation system is an interesting solution.

Viruses and worms
Generally, viruses are mainly present on applications that index files. The virus is caught like a classic virus, by downloading a corrupted file. A new generation of virus has emerged on the overlay networks. They take advantage of the endogenous routing service. Indeed, a classic virus, in order to quickly spread must retrieve a list of new hosts to infect at each infection. The most effective method was to retrieve email addresses stored by the user and thus spread through the messaging application. The routing tables of the nodes in the overlay networks possess a large set of IP addresses. Some viruses have been designed to use an expansion based on these IP addresses. As an example, the virus VBS.Gnutella once it has infected a Gnutella user, infects the files hosted and shared across the Gnutella network. The best solution is to use a reliable anti-virus.

Free riding
The "free riding" attack is a passive attack is that takes advantage of the applications running over the overlay networks without contributing. This kind of attack applies to the applications that index files and applications that share resources. One solution is to keep a trace of the activity of the users. But most applications do not implement this function at a network level. In the case in BitTorrent or eDonkey network the client software has to regulate itself. If this solution is valid for users with no programming knowledge, it is however less for other users. It has emerged modified versions of eMule called eChamblar for example that allows to download files from other nodes without contributing.

1.2.5 Reputation system

1.2.5.1 Introduction

In a distributed system where it is impossible to prevent an entity of the system from maliciously acting, the best security solution is to assign important roles only to entities that we can trust. The problem is then to make the process of reputation attribution a distributed process.

In first, it is necessary that each user can give its opinion on an other user or on a transaction. It is the second time, a distributed mechanism has to aggregate these local inputs to achieve an overall score for each user. To finish, the overall scores must be available for all users.

The two important points of a reputation system are:

- The aggregate function.
- The storage of the overall score.

To be accurate, the aggregate function must take into account the fact that even malicious users can express their opinions on the other users. This function must also take into account the date of issuance of each opinion. An old opinion should have less impact on the overall score a recent opinion. Finally, the context in which has been issued the opinion must also be taken into account. Everyone must be able to access to the overall scores without being able to modify them. It is also inconceivable that an user stores his own global mark. Storage should be either anonymous, encrypted, or even shared.

The centralized reputation systems operate on a simple principle: a server stores the opinions given by the users. This server is also responsible for aggregating the opinions that it receives and for storing the overall scores. The server must sent the score of a given user to any user who requests it. These systems suffer from defects inherent to all the centralized systems (congestion and single point of failure).

The decentralized reputation systems, however, are more effective and more in accord with the philosophy of the overlay networks. There are two subclasses of distributed reputation systems. Local systems and global systems. In the local ones, each user stores the overall scores he attributes to the other users. Consequently, there are several scores for each user. In the global ones, the overall score assigned to each user is stored over the network and there is a single note for each user. In addition to these subclasses, there exist two philosophies for the reputation systems, the first philosophy consist in considering that any node has a low reputation until the node acquires reputation points, the second one considers that any node is acting properly considered until a charge is brought against him by the other nodes.

1.2.5.2 Examples

It will now be presented two reputation systems deployed over an overlay network.

PeerTrust (Xiong et al. [2004]) provides a distributed reputation system over an overlay network. Each node in the network can deliver an opinion on the transactions performed with the other nodes. PeerTrust was originally developed on P-Grid (Aberer [2001]) , but it can be developed on any structured overlay network. The overall score is stored on some network nodes. To identify these nodes, a hash function is used on the identifier of the node whose the reputation is evaluated in order to obtain a key. The nodes that will be responsible for storing the overall score are those whose identifier is closest to the key. When a node wants to give an opinion on another node of the overlay network, it sends it to these nodes. The overall reputation score is calculated using the following function T:

$$T(n) = \alpha \sum_{i=1}^{I(n)} S(n,i) \times Cr(p(n,i)) \times TF(n,i) + \beta CF(n)$$

Where $I(n)$ is the total number of transactions performed by the node n, $p(n,i)$ is the set of the other nodes that have participated in the ith transaction of the node n, $S(n,i)$

the amount of satisfaction that the node n has received for $p(n, i)$, $Cr(m)$ is the confidence in the opinion submitted by the node m, $TF(n, i)$ the context of the ith transaction of the node n and $FC(n)$ the adaptive context assigned to the community for the node n. To better understand the above formula, let us only note of a transaction. For a transaction, the rating is calculated by the following formula:

$$S(n) \times Cr(p(n)) \times TF(n)$$

$S(n)$ is the satisfaction that the node n received for the transaction. The satisfaction is multiplied by $Cr(p(n))$. This value corresponds to the reputation of the nodes that have participated and judged the transaction. This reduces the impact of the opinion of the malicious nodes. Since a malicious node has a low reputation its opinion will be less considered. The last value is the context in which the node has completed the transaction. Thus if a node performs a transaction correctly in a critical time during an attack such as the note assigned to the transaction will be more important. The reputation score of a node is the sum of the scores obtained for all the transactions that the node have performed to which we add $FC(n)$. This is the context given by the community node. The values α and β, which have not been mentioned are parameters that allow to give the priority to the context or notes of transactions.

Fireflies (Johansen et al. [2006]) is an overlay network endowed with reputation system. In fireflies, a node have a state that can be: correct nodes that are nodes following the protocol of Fireflies, stopped nodes that are nodes that encounter momentary failures and byzantine nodes that are nodes that no longer follow the protocol. Initially, all the nodes are in the correct state and then it is in a distributed way that the nodes will decide of their states. Each node in a correct state has a *view*, a table of neighbors, and a communication channel that can be used to send a message to the all the nodes (the nodes in the byzantine state can also use this channel). The *view* of a node is a subset of the nodes of Fireflies. The *view* of a node is filled by the nodes that it considers in a correct state. If a node is not in the *view* of another node, it means that the second node considers that the first node is in a stopped state. The routing table of a node is a subset of the view of this node.

In Fireflies, the nodes are organized in $2t + 1$ rings. The value of t depends on the degree of intrusion that Fireflies must be able to tolerate: the more this value is large, the more fireflies will be tolerant. Each node of Fireflies belong to each ring and its position is calculated using a hash function on the identifier of the node concatenated with the identifier of the ring. Consequently, the order of a node on each ring is different with a high probability. When a node suspects its successor on a given ring to be in the stopped state, it sends a signed report through the communication channel to the other nodes. This report contains the identifiers of the ring and of the suspected node. The suspected node has a finite amount of time to emit a counter-report over the communication channel. In the lack of counter-report, the suspected node is removed from the *view* of all the nodes of Fireflies. If a node sends two many unjustified report, the nodes declare this node as in byzantine state and remove the byzantine node from their *views*. In the Figure 1.14 is represented a Fireflies network with 7 nodes and 3 rings. In this Figure, the valid report are represented by plain arrows and invalid report by dotted arrows. The report from D against B on the middle ring is considered valid because there is no node between B and D on the ring. The report from C against A on the outer ring is valid because there is a

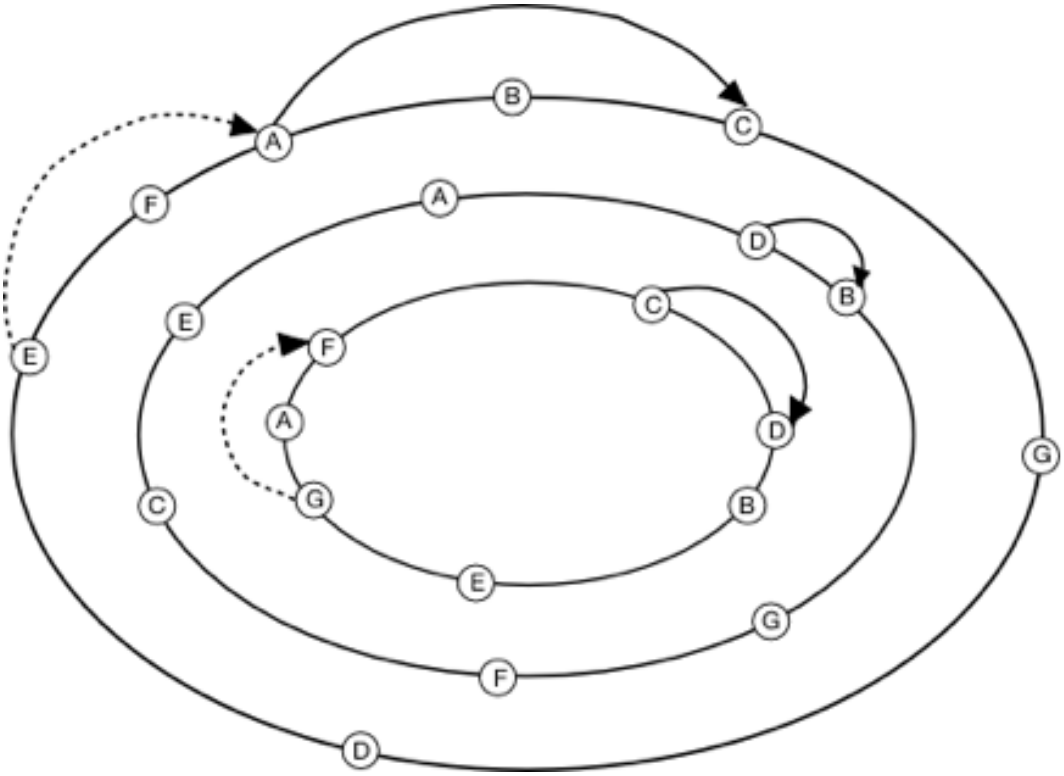


Figure 1.14: A fireflies overlay network with 3 rings

valid report against B. The report from E against A is invalid because there is no valid accusation against F.

Chapter 2

The protocol ROSA

We have seen in the last chapter that the type of an overlay network must be chosen in function of the purpose of this overlay network. In our case, we want to develop ROSA as an adaptable overlay network, i.e. an overlay network with a parametrizable purpose. So ROSA must belong to the bouncier type of overlay network, the unstructured overlay network. Making ROSA an unstructured overlay network allows to build a topology management protocol that depends of the purpose and the underlying network type. This is presented in the Chapter 2

Once this topology management protocol is built, it remains to define a routing algorithm more efficient that the flooding routing algorithm generally used over the unstructured network. In order to do so, we have built over ROSA a Distributed HashTable. Once endowed with this DHT ROSA possesses the benefits of the unstructured and structured overlay networks. This is presented in the Chapter 3

If we refer to the classification given of the overlay network given in the last Chapter (Chapter 1), ROSA cannot be classified. It is quite logical since we have seen that the type of the overlay network depends on the goal of this overlay network and that ROSA is generic template of an overlay network. Indeed the purpose of ROSA is set when one sets the definition of the density (see Chapter 6) that will be used on ROSA.

2.1 Principles of ROSA

ROSA nodes are organized in cluster called lumps. A lump is a set of fully connected nodes. In graph theory such objects are called cliques. A clique in an undirected graph is a subset of the vertex set, such that for every two vertices in the subset, there exists an edge connecting the two. ROSA can be represented by a entanglement of lumps. Each node of ROSA belongs to at least one of the lumps. Each of the lumps of ROSA is associated with a metric called density. We shall see in Chapter 6 there are several kinds of densities and that the choice of this density that defines the characteristics of ROSA.

We call the topology management the way of how the graph composed of the nodes and the links and the set of lumps of ROSA evolves during the time and according to the events.

The principle of management of the topology of ROSA is similar to a recipe for pancake batter. In a recipe of pancake batter one should dilute the lumps with high densities to

increase the density of the areas with less flour. In ROSA, the nodes calculate the density of lumps which they belong, share with their neighbors about the knowledge of the low densities lumps, and leave lumps having high densities to increase the lumps with low densities.

2.2 The constants of ROSA

A number of parameter sets the properties of ROSA. There is the definition of the density. A chapter of this thesis is dedicated to the density, Chapter 6. There is also few constants that all the nodes of ROSA must share that any new node can acquire when it connect to ROSA. In this section we list these constants, explain their roles and influences on the behavior of ROSA.

Interval : The constant Interval sets time that a node has to wait between two executions of the main loop. This constant determines the relation between the efficiency and the speed of reaction of ROSA and the network bandwidth and computational use necessary to the functioning of ROSA. The smaller the Interval will be, the faster the reaction to failure will be. In return the excess use of bandwidth and computing will be great. Unlike, the greater the Interval will be, the less greedy ROSA will be. But Rosa will also be less responsive and effective.

MessageIdNumMax : Each node of ROSA keeps a list of the identifiers of the last messages he received. The constant MessageIdNumMax sets the maximum size of this list. The risks taken when choosing a value MessageIdNumMax too small is to contrast with the consumption of resources brought caused by a too high value of this constant. If MessageIdNumMax is too small then a node may receive a message and may not realize that it has already received this message. If MessageIdNumMax is too large then looking up the list of identifiers of recent messages take more time than necessary. And the list would be over-dimensioned in order to achieve the goal that the list is supposed to do.

LumpSizeLimit : This constant is one of the two constants which limit the number of neighbors per node. This limitation is explained in detail in Section 2.4.9. The constant LumpSizeLimit sets the maximum number of nodes that can compose a lump. When a lump is composed of more nodes than LumpSizeLimit, some of the nodes of the lump must leave it.

LumpNumberLimit : The constant LumpNumberLimit sets the maximum number of lumps to which a node can belong. This constant has a dual role. The first role is, with the constant LumpSizeLimit, to limit the number of neighbors that a node can have. The second reason for the constant LumpNumberLimit is to set the maximum number of shortcuts in the 'chain of lump'. All the details concerning the second role of this constant are given in Chapter 3.

2.3 Representation of a node, a neighbor and a lump in the memory of a node

Each node in ROSA has an identifier, a list of neighbors and a list of lumps. The identifier of a node is an integer belonging to $[1, 2^{128} - 1]$. We consider for now, that the identifier

of a node is randomly chosen. The list of lumps of a node is a list of lumps which the node belongs. The list of neighbors of a node is a list of the identifiers and the physical addresses of nodes to which it is virtually linked. This list of neighbors must reflect every changes on the list of lumps. Each node also possesses one flag 'connected'. This flag is set to true if the node is connected to ROSA and else is set to false. To complete the representation of a node we add a list of lumps with low density and a list of the last received messages identifiers. These two list are initialized to empty. The first one is used to store the knowledge of the nodes about the lumps with low densities in its neighborhood. The second one is used to allow every nodes of a lump to receive a message despite failures (see Section 2.4.3).

A neighbor of a node is represented by the pair <id, phy> where id is the identifier of this neighbor in ROSA and phy is its physical address on the network on which ROSA is deployed. A flag 'alive' complete this representation. This flag is set to true by default and will be used for the detection of failures.

A lump has an identifier, a list of the identifiers of the nodes which composes it and all the information allowing each node to calculate its density. The information necessary to compute the density depends on the definition of the density chosen, further details will be given in the Chapter 6 dealing with the density. The identifier of a lump is an integer belonging to $[1, 2^{128} - 1]$. As for the node this identifier is randomly chosen. A lump also has a list of physical addresses of the nodes that compose it.

Node

id : *integer* ;	// rand$(0, 2^{128} - 1)$
neighbor_list : *list<neighbor>* ;	// nodes linked to *node*
lump_list : *list<lump>* ;	// lumps to which *node* belongs
connected : *boolean* ;	// true if node is connected
know_lump_list : *list<lump>* ;	// low densities lumps
message_id_list : *list<integer>* ;	// Ids of received messages

Neighbor

id : *integer* ;	// identifier of neighbor
phy : *physical address* ;	// physical address of neighbor
alive : *boolean* ;	// true if neighbor alive

Lump

id : *integer* ;	// rand$(0, 2^{128} - 1)$
nodeId_list : *list<(integer>* ;	// Ids
nodePhy_list : *list<(physical address>* ;	// physical addresses
data_density : ... ;	// cf (Chapter 6)

Figure 2.1: Representations of nodes and lumps

2.4 Protocol

In this section we deal with the basis of the ROSA protocol. We focus on a few primitive functions used by the nodes of ROSA, such for example the function that allows a node to connect and to leave ROSA, or the function that allows a node to send a message to all other nodes in a lump without the failures on the virtual links prevent the message from reaching its destination. Next, we see the main loop of ROSA and the different functions that compose it. Thus we see first how the failures on the links are detected and managed in ROSA. Then we see how the protocol limits the number of neighbors per node. We see how a node optimizes the ROSA density in its neighborhood. To finish we see how the absorptions of the lumps are managed.

2.4.1 Connecting to or initiating ROSA

To join ROSA a node must know a node already connected to ROSA, this node is called bootstrap_node. If the node cannot find such a bootstrap_node, it must initiate the network. This is described in the pseudo-code 2.2.

In the current implementation the discovery of a bootstrap_node is done using a central server called bootserver. This server maintains a list of active nodes of ROSA. To connect to ROSA, a node must contact the bootserver. It responds with the physical address of a node connected. This address is randomly selected in its list. If the list is empty, the bootserver replies with the void address. This void address tells a node wanting to join that ROSA that is the first node and that it must initiate ROSA.

Node.connectROSA

```
1: bootstrap_node ← findBootsrapNode() ;
2: if bootstrap_node = void then
3:     initROSA() ;
4: else
5:     send(Hello()) to bootstrap_node ;
```

Figure 2.2: The connectROSA function

To keep up to date the list of the nodes connected to ROSA, every connected nodes must periodically give signs of life to the bootserver. The bootserver deletes from its list all the nodes that do not do this.

This solution based on a central server is not a very good solution. The main drawback of this solution is that if the bootserver fails or suffers a DDOS attack, no more nodes can join ROSA. Another possible solution, but that has not been implemented yet, is that each node connected to ROSA acts as bootserver. In this solution, a node willing to join should broadcast a request over its physical network. The nodes connected to ROSA will wait for these requests and will reply with the physical address of one of their neighbors.

Once in possession of the physical address of a node already present on ROSA, a node can start the procedure of connection. This procedure consists in sending a message *Hello*

to the bootstrap_node.

Node.initROSA

```
1: lump ← new lump ;
2: lump.nodeId_list ← id ;
3: lump.nodePhy_list ← phy ;
4: lump_list ← lump ;
5: connected ← true ;
```

Figure 2.3: The initROSA function

At the reception of this message, the bootstrap_node looks down its list of lumps, selects the lump with the lowest density, encapsulates the selected lump in a message *BootstrapLump* and sends it to the node that wants to connect to ROSA. As soon as the node willing to connect to ROSA receives this message, it verifies that it is not already connected. If it is already connected and it has received a message *BootstrapLump* by mistake, it simply ignores the message. Otherwise it extracts the lump contained in the message and tries to join this lump. We will see in Section 2.4.4 how a node joins a lump and which are the conditions that the node has to satisfy in order to join a lump. If the node is not able to join this lump the procedure of connection must be started again from the beginning. The pseudo code of this process is described in the Figure 2.4.

Messages handling

```
...
1:  upon receive Hello() do
2:      if connected then
3:          return ;
4:      sender ← getSender(Hello()) ;
5:      lump_to_send ← getWeakestLump() ;
6:      send(BootstrapLump(lump_to_send)) to sender ;
7:  upon receive BootstrapLump(lump) do
8:      if joinLump(lump) then
9:          connected ← true
10:     else
11:         connectROSA() ;
...
```

Figure 2.4: Handing *Hello* and *BootstapLump* messages

If this node is the first one, it must initiate ROSA. In order to initiate ROSA, the node creates a lump composed by only itself. Afterwards the node adds the newly created lump

to its list of lump. We will see later that this lump is the only lump that is created from scratch. All the other lumps of ROSA result from a split of an already existing lump. To finish the node positions its connected flag to true as described in the pseudo code in the Figure 2.3).

2.4.2 Leaving ROSA

Two methods are available to a node that wants to leave ROSA. The first one is the good method and must be used whenever it is possible. The other method is much less elegant. These two methods are described in this section.

The first method consists for a node in calling the function leaveROSA. A node using the first method has only to send a message *LeaveROSA* to all its neighbors. A node receiving a message *LeaveROSA* gets the identifier of the sending node. Afterwards the node that received the message, removes form its list of lumps the identifier, the physical address and all the data concerning the sending node. This method is described in the pseudo code in the Figure 2.5.

```
Node.leaveROSA
1: for all neighbor ∈ neighbor_list do
2:     send(LeaveROSA()) to neighbor ;

Messages handling
   ...
3: upon receive LeaveROSA() do
4:     sender ← getSender(LeaveROSA()) ;
5:     for all lump ∈ lump_list do
6:         if sender ∈ lump.nodesIdList then
7:             lump.nodesIdList ← lump.nodesIdList - sender.id ;
8:             lump.nodesIdList ← lump.nodesIpList - sender.phy ;
               /*
               removing the data concerning sender from lump.data_density
               */
   ...
```

Figure 2.5: The good method for a node to leave ROSA

The other method is applied when the node does not give any sign of life to its neighbors. As discussed in Section 2.4.8 when a node does not give sign of life anymore to its neighbors, these neighbors gradually remove the failing node from their list of lumps and neighbors. Normally, the first method should be preferred because it generates less bandwidth and CPU cycle use. The second method should be used only when a node fails or when it is in the inability to apply the first method.

2.4.3 Sending a message to all the nodes of a lump

Some messages, that are needed for the good working of ROSA, have to be sent to all the nodes that compose a lump. This is particularly the case for all the messages that are intended to tell the nodes about the changes (splits and absorptions) of the set of the lumps of ROSA.

To send a message to all the nodes that compose a lump, a node could simply and directly send the message to all the other nodes of the lump. But some failures could disrupt the communication on the virtual links between the nodes composing the lump. In this case the message could not reach some nodes of the lump. To prevent that the failures interfere with the communication between the nodes of a single lump, each message has a unique identifier. This identifier is an integer randomly chosen in $[1, 2^{128} - 1]$ and each node of ROSA keeps a list of the identifiers of the last messages (see MessageIdNumMax in the Section 2.2) that it received.

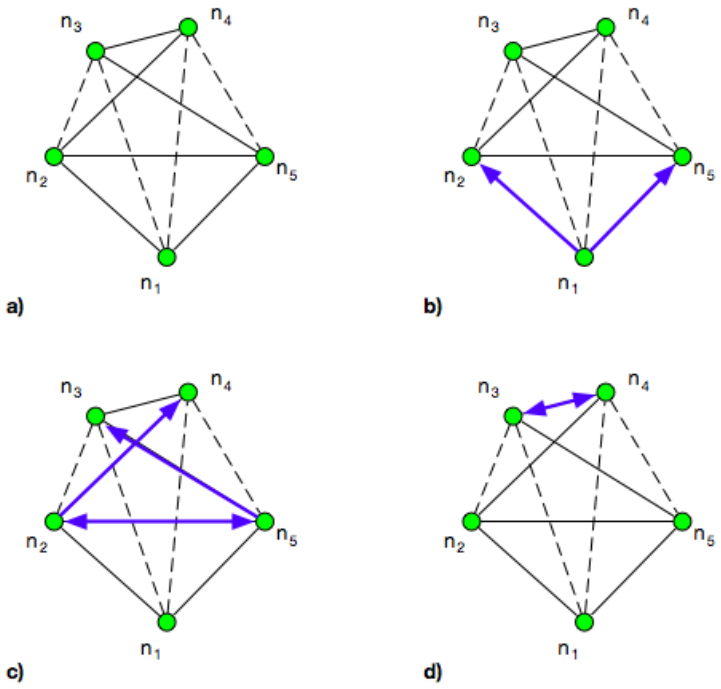


Figure 2.6: Propagating a message to all the other nodes of a lump

So any node is able to distinguish the new messages from the ones that it has already received and handled. When a node receives a message destined to all the nodes of a lump, it checks in its list of the already received messages identifiers if the received message is a new one. If the message was already received before, the node ignores it. If it is the first time that the node receives this message, the node handles it and forwards it to the other nodes that compose the lump. This way, despite the presence of some failures, if there exists a path composed by the virtual links between the nodes that goes through all the nodes of the lump, it is sure that the messages will reach all the nodes to which it is intended.

In Figure 2.6 is represented a node sending a message to all nodes in a lump despite many failures. The nodes are represented by green circles. Virtual links by lines connecting the nodes, solid lines for valid and dashed lines for links experiencing failures. The sending of the message are represented by blue arrows. In the figure the nodes n_1 wants to send a message to all nodes in the lump. Some failures only allow the node to send the message to nodes n_2 and n_5. In the remainder of the figure one can see that the message spreads from node to node to reach all nodes in the lump. Everyone can also see that a node does not propagate the message that it has already received it.

2.4.4 Joining a lump

In this Section, first we see what it means to join a lump for a node. We then see how a node joins a lump and notice the other nodes of this lump about the join.

A node joins a lump means that the node creates virtual links with the nodes that compose the lump. But it is not always possible to create a virtual link between two nodes. In this case a node can perform a partial join of a lump. We will see the differences between all possible cases: the case where a node can join a lump and the case where it is unable to join this lump.

Node.join(lump) (**Input:***lump*; **Output:***boolean*)

```
1: joinRatio ← 1 - checkLinks(lump) / lump.size ;
2: if joinRatio > joinRatioLimit then
3:     return false ;
4: lump.nodeId_list ← lump.nodeId_list ∪ node.Id ;
5: lump.nodePhy_list ← lump.nodePhy_list ∪ node.Phy ;
   /* adding some data to lump.data_density */
6: send(UpdateLump(lump)) to lump ;
7: return true ;
```

Messages handling

```
   ...
8: upon receive UpdateLump(lump1) do
9:     for all lump2 ∈ lump_list do
10:        if lump1.id = lump2.id then
11:            lump2 leftarrow lump1 ;
12:            return ;
   ...
```

Figure 2.7: Joining a lump

To join a lump, a node must be aware of this lump, i.e. be in possession of the representation of the lump. Once in possession of the representation of the lump, the node must first check if it can create virtual links with all the nodes that compose the lump. This is done with the help of the the ping utility (Muuss [1983]) on IP networks or with

any other utility that performs the same action on non-IP networks. Once verification is complete the node determines the joinRatio. The joinRatio is equal to 1 minus the ratio between the number of virtual links that the node can create and the number of nodes that compose the lump, i.e. $1 - checkLinks(lump)/lump.size$. The pseudo code of the CheckLinks function can be found in the Appendix.

If the joinRatio is equal to zero, we are in the case of a full join. If it exceeds the limit sets by joinRatioLimit then it is impossible for the node join this lump. And in other cases the node performs a partial join of the lump.

After the computation of the joinRatio and if it does not prevent the node to join the lump, the node modifies the representation of the lump in order to reflect the fact that the node has joined the lump. It adds its identifier in the list of the node identifiers and its physical address in the list of the physical addresses. The node also modifies all the information necessary to compute the density of the lump (more details will be given in the Chapter 6).

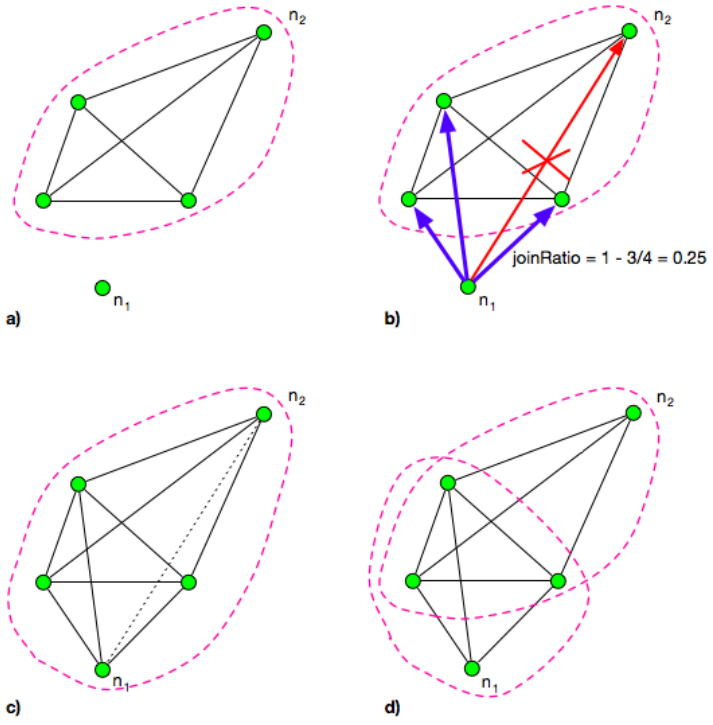


Figure 2.8: The partial join of a lump

Finally the node sends through the intermediary of a message *UpdateLump*, the new representation of the lump to all the nodes that compose the lump. The nodes receiving the message, extract the new representation and update their list of lumps, replacing the old with the new representation. The join procedure is summarized in the pseudo code of the Figure 2.7.

The difference between a complete join and a partial join is the handling of the broken links that will occurs after the partial join. Indeed after the partial join procedure, since it has been impossible to establish virtual links with some nodes of the lump, the partial

join leads to many lump splits.

The joinRatio is used to limit the number of split that will occur after a join. It is not productive, for example, to join a lump composed of 5 nodes if it is not possible to create virtual links with 3 of these nodes. This would create a significant number of lumps and would generate a substantial network usage.

In the Figure 2.8 is schematized the partial join of a lump. The nodes are represented by green circles. The lumps are represented by purple dashed shapes. The pings are represented by arrows, blue for the successful pings and red by the failing ones. The broken virtual links are represented by dashed black lines. In **a)**, the node n_1 wants to join a lump composed of 4 nodes. The node can only create virtual links with 3 of these nodes as it is described in **b)**. The computed joinRatio is equal to 0.25. In this example, we assume that the joinRatio is less than the joinRatioLimit. The node n_1 is allowed to join the lump. It results a new lump from from this join. The missing virtual link between n_1 and n_2 leads to the split of the new lump.

2.4.5 Leaving a lump

Leaving a lump consists in a node modifying the representation of the lump that the node wants to leave. The representation must reflect the fact that the node does not compose the lump anymore. The modifications that the node must perform are the inverses of those that node must perform when it wants to join a lump. Once the representation of the lump up to date, the node send it with a message *UpdateLump* to the nodes that compose the lump. Each node that receives the new representation updates its list of lumps. The pseudo code corresponding to the handling of the messages *UpdateLump* is shown in the Figure 2.7.

Node.leave(lump) (**Input:***lump*)

1: lump.nodeId_list ← lump.nodeId_list - node.Id ;
2: lump.nodePhy_list ← lump.nodePhy_list - node.Phy ;

/* removing some data to lump.data_density */

3: send(*UpdateLump*(lump)) to lump ;

Figure 2.9: Leaving a lump

2.4.6 Splitting a lump

The split of a lump happens for two reasons, when a virtual link is broken or during the limitation of the number of nodes per lumps. The split of a lump is organized around two nodes of the lump, it consists in splitting the lump in order to obtain two lumps. The nodes that compose the first lump that results from a split are the nodes of the split lump minus one of the two nodes around which the split is organized. The nodes that compose the second lump that results from a split are the nodes of the split lump minus the other node around which the split is organized.

Node.split(lump, neighborId) (**Input:***lump*, *integer*)

1: split(lump, pair(node.id, neighborId)) ;

Node.split(lump, ids) (**Input:***lump*, *pair<integer>*)

2: $lump_1 \leftarrow$ lump.without(ids[1]) ;
3: $lump_2 \leftarrow$ lump.without(ids[2]) ;
4: send(*SplitLump*(lump.id, $lump_1$, $lump_2$)) to lump ;

Messages handling

...
5: **upon** receive *SplitLump*(lumpId, $lump_1$, $lump_2$) **do**
6: **for all** $lump_{tmp} \in$ lump_list **do**
7: **if** $lump_{tmp}$.id = lumpId **then**
8: lump_list $\leftarrow$ lump_list - lump_list.get(lumpId) ;
9: lump_list $\leftarrow$ lump_list $\cup$ $lump_1$;
10: lump_list $\leftarrow$ lump_list $\cup$ $lump_2$;
11: return ;
...

Figure 2.10: Splitting a lump

Let n_1 and n_2 be some nodes that belong to the set of the nodes that compose a lump. Let *lump* be this lump. More formally split of *lump* around the pair of node $<n_1, n_2>$ means:

$$\text{split}(lump, < n_1, n_2 >) \rightarrow \{lump_1, lump_2\}$$

such that:

$$\begin{aligned} lump_1\text{.nodeId\_list} &= lump\text{.nodeId\_list - } n_2\text{.id} \\ lump_1\text{.nodePhy\_list} &= lump\text{.nodePhy\_list - } n_2\text{.phy} \\ lump_1\text{.data\_density} &= lump\text{.data\_density - all the data concerning } n_2 \end{aligned}$$

and:

$$\begin{aligned} lump_2\text{.nodeId\_list} &= lump\text{.nodeId\_list - } n_1\text{.id} \\ lump_2\text{.nodePhy\_list} &= lump\text{.nodePhy\_list - } n_1\text{.phy} \\ lump_2\text{.data\_density} &= lump\text{.data\_density - all the data concerning } n_1 \end{aligned}$$

When a node wants to split a lump around itself and one of the other nodes that compose the lump, it builds the representation of the two lumps that result from the split. Then the node sends the representations of the lumps resulting from the split to the affected nodes via some messages *SplitLump*. These messages contain, in addition to representation of the resulting lumps, the identifier of the split lump. A node that receives a message *SplitLump*, looks in its list of lumps the one that has the identifier contained in the message. If the node does not find such a lump, it ignores the message. Otherwise it replaces in its list of lumps, the split lump by the two lumps contained in the messages. The Figure 2.10 contains the pseudo code relative to the split of lump.

2.4.7 Main loop of ROSA

The main loop of a ROSA node is in pseudo-code in the Figure 2.11. This loop is repeated every 'Interval'. This parameter must be defined experimentally according to the capacities of the network on which ROSA will be used. More details concerning the role of 'Interval' can be found in Section 2.2.

The main loop of ROSA consists of 5 functions: checkFailure, checkAbsorption, checkMemberLimit, checkLumpLimit et enhanceROSA. The first function aims to detect the failure occurring on the virtual links and to react to these failures. The role of the function checkAbsorption is to detect and manage the absorption of a lump of the list *lump_ list* by another lump of this list. The functions checkMemberLimit and checkLumpLimit aim to limit the number of neighbors per node. The function enhanceROSA is responsible for the optimization of ROSA. This function decides in which cases the node must leave a lump of high density to increase the density of a lump with low density.

```
Node.main
    while connected do
1:      checkFailure() ;
2:      checkAbsorption() ;
3:      checkNpL() ;
4:      checkLpN() ;
5:      enhanceRosa() ;
6:      sleep(Interval) ;
```

Figure 2.11: The main loop of ROSA

We will see in the Chapter 3 concerning the chain of lumps that the main loop will be completed and modified.

2.4.8 Handling the broken links

A major advantage of ROSA is that the protocol can quickly detect the failures occurring on the virtual links between the nodes and modify the topology of the network to reflect such failures. The handling of the broken virtual links is done by the function checkFailure of the main loop of ROSA. The handling of the broken links consists of two steps, the failures detection and the topology modification.

The first step consists for each node in giving signs of life to its neighbors by sending a message *Alive*. If a link between two nodes was broken then the sending of these messages *Alive* are interrupted. The messages *Alive* were also used to spread the knowledge about the lumps with low density. In each message *Alive* sent by a node is encapsulated a lump. This lump is the lump with the lowest density in list of lumps. When a node receives a message *Alive*, it extracts the lump contained in the message and adds it to its list of known lumps. This way each node knows the lump with the lowest density to which belong each of its neighbors.

Node.checkFailure

```
1:  for all neighbor ∈ neighbor_list do
2:      if neighbor.alive = true then
3:          send(Alive(getWeakestLump())) to neighbor ;
4:          neighbor.alive ← false ;
5:      else
6:          for all lump ∈ lump_list do
7:              if neighbor ∈ lump.nodesId_list then
8:                  split(lump, neighbor.id) ;
```

Messages handling

```
    ...
9:  upon receive Alive(lump) do
10:     sender ← getSender(Alive(lump)) ;
11:     know_lump_list ← know_lump_list ∪ lump ;
12:     neighbor_list.get(sender).alive ← true ;
    ...
```

Figure 2.12: Handling the failures of the virtual links

Once a node detects a broken link by the lack of message *Alive* from one of its neighbors, the second step is for a node to modify its list of lumps to take into account the failure of this link. When a lump detects that the virtual link between itself and a neighbor is broken, it looks down through his list of lumps for the lumps that contains the failing neighbors. When a node encounters such a lump, it splits the lump around itself and the failing neighbor. The split of a lump is described in Section 2.4.6. The pseudo code concerning the detection and the handling of the broken links is shown in Figure 2.12.

2.4.9 Limiting the number of neighbors

To be scalable, the number of neighbors of a ROSA node is bounded. Rather than setting a maximum size to the list of neighbors and to force a node to separate from one of its neighbors when the capacity of the list of neighbors is reached, it is by imposing a maximum number of nodes per lump and a maximum number of lumps per node that the protocol of ROSA limits the number of neighbors per node. Limiting the number of nodes per lump is done with the function checkNpL and limiting the number lumps per node is done with the function checkLpN.

To limit the number of nodes that compose a lump, a node looks down its list of lumps for the lumps whose the number of nodes exceeds the limit. When such a lump is found, the node may leave or split this lump.

A node can leave a lump that has reached the maximum number of node if the node belongs to at least another lump and if the leave of the lump does not imply a loss of density. If either of these conditions is not satisfied, the node must select a pair of nodes

```
Node.checkNpL
 1: for all lump ∈ lump_list do
 2:     if lump.size > LumpSizeLimit then
 3:         if canLeave(lump) then
 4:             leave(lump) ;
 5:         else
 6:             split(lump, findNodeForSplit(lump)) ;

Node.canLeave(lump) (Input:lump; Output:boolean)
 7: if lump_list.size > 1 then
 8:     if lump.density() = lump.without(node).density() then
 9:         return true ;
10: return false ;

Node.findNodeForSplit(lump) (Input:lump; Output:pair<integer>)
11: result_density ← infinity ;
12: result_pair ← null ;
13: for all id1 ∈ lump.nodeId_list do
14:     for all id2 ∈ lump.nodeId_list and id1 ≠ id2 do
15:         tmp_pair ← pair(id1, id2);
16:         if getSMD(lump,tmp_pair) < resulting_density then
17:             result_density ← getSMD(lump,tmp_pair) ;
18:             result_pair lestarrow tmp_pair ;
19: return result_pair ;
```

Figure 2.13: Limiting the number of nodes per lump

in order to split the lump around these two nodes. This selection is done as follows, the node looks down the list of the nodes that compose the lump. For every pair of elements of this list, the node computes what would be the density of the two lumps resulting from the split if the lump was split around these elements. Finally the node selects the pair that maximizes the densities of lumps that will result from the split.

The pseudo code of the function checkNpL is in Figure 2.13. The function getSMD (see The Appendix) is a function that returns the minimum density obtained if a node splits a lump around the nodes given in argument

We have seen the limitation of the number of nodes per lump. This limitation is the first of the two steps of the limitation of the number of neighbors per node. The second step consists in limiting the number of lumps to which a node can belong. To limit the number of lumps to which it belongs, a node looks at the size of its list of lumps. While the size of the list exceeds the limit, the node has to select a lump to leave. This choice is based on the hypothetical density that will have the lump if the node left it. The node

selects the lump that will have the greatest theoretical density and leaves it. This is done by the function checkLpN, its pseudo code is in Figure 2.14.

```
Node.checkLpN
1: while lump_list.size > LumpNumberLimit do
2:     leave(findLTL()) ;

Node.findLTL() (Output:lump)
3: result ← null ;
4: for all lump ∈ lump_list do
       lump1 ← lump.without(node.id) ;
       lump2 ← result.without(node.id) ;
5:     if lump = null or lump1.density() > lump2.density() then
6:         result ← lump ;
7: return result ;
```

Figure 2.14: Limiting the number of lumps per node

We shall see in the Chapter 3 concerning the 'chain of lumps' that these two functions will be completed and modified.

2.4.10 Handling the lumps absorptions

When the set of nodes that compose a lump contains all the nodes that compose another lump, it happens an absorption of the second lump by the first one. It results from this, the disappearance of the absorbed lump from the list of lumps of all the concerned nodes. The handling of such absorption cannot be done without exchanging messages between the nodes that compose the absorbed lump. Since all the nodes that compose a lump can be included in the sets of nodes of several different lumps, many lumps are susceptible to absorb the lump. The nodes must choose one lump among the set of possible absorber lumps. This choice should be the same for all nodes that compose the absorbed lump.

The detection and the handling of the absorptions of lumps is done by the nodes with the help of the function checkAbsoprtion. A node detects the absorption of a lump in looking down its list of lumps. For each lump of this list it checks if there is another lump on the list that can absorb it. If such a couple of lumps is found, the node modifies its list of lumps in order to reflect the absorption and notices the other nodes of the lumps about this absorption. This is done using messages *AbsorbLump*. These messages contain the identifier of the absorbed lump and the identifier of the absorbing lump. A node that receives a message *AbsorbLump* removes from its list of lumps the lumps corresponding to the identifier of the absorbed lump. We will see in the Chapter 3 dealing with the 'chain of lumps' how the identifier of the absorbing lump is used. The pseudo code corresponding to the detection and the handling of the absorptions of lumps is in the Figure 2.15.

```
Node.checkAbsorption
1: for all lump1 ∈ lump_list do
2:     for all lump2 ∈ lump_list do
3:         if lump1.canAbsorb(lump2) then
4:             send(AbsorbLump(lump1.id lump2.id)) to lump ;

Lump.canAbsorb(otherLump)  (Input:lump; Output:boolean)
5: for all id1 ∈ otherLump.nodesId_list do
6:     found ← false ;
7:     for all id2 ∈ lump.nodesId_list do
8:         if id1 = id2 then
9:             found ← true ;
10:            break ;
11:    if not found then
12:        return false ;
13: return true ;

Messages handling
    ...
14: upon receive AsorbLump(absorbedLumpId, absorbingLumpId) do
15:     lump_list ← lump_list - lump_list.get(absorbedLumpId) ;
    ...
```

Figure 2.15: Handling the absorptions of the lumps

2.4.11 Increasing the overall density

We saw in Section 2.4.8 dealing with the handling of the broken links that each node periodically receives messages *Alive* from its neighbors. These messages contain some lumps with low densities. We show in this section how, from this list of the received lumps with low densities, each node locally increases the density of ROSA. A node increase the density with the help of the function enhanceROSA of the main loop of ROSA.

The first step consists in a node browsing the list of the lumps received from its neighbors. For each of these lumps, the node compares the current density of the lump to the hypothetical density that the lump will have if the node joins it.

The calculation of this density is done with the function getRMDJ. The pseudo code for this function is in the the Appendix. This function performs a copy of the representation of the lump and simulates the changes caused by the join of the node. In the case where the lump have reached the maximum number of nodes per lump, the function getRMDJ also simulates the split resulting from the join. Then the function returns the hypothetical density of the lump(s) obtained during the simulation.

After comparing the densities of lumps received from its neighbors to their hypothetical densities if the node joins them, the node removes from this list each of the lump with an

hypothetical density less of equal to their current density. The node now possesses a list of lumps that it can increase the density. If this list is empty, the increasing process stops, else it continues.

Let us consider that the list of the lumps which the density may be increased by the node is not empty. The node tries to join the lump of this list which has the lowest density. If the maximum number of lumps which a node can belong is not reached, the node joins the lump. In case where the limit of lumps is reached, the node must leave a lump if it wants to join a new one. It means that in order to increase the density of the lump, the node may have to decrease the density of another lump. Consequently, before joining the lump the node ensures that the join will effectively increase the local density of the node and not decrease it. The node looks down its list of lumps, and for each lump it compares the current density to the hypothetical density that the lump will have if the node leaves it.

```
Node.enhanceROSA()
 1: toEnhance ← null ;
 2: for all lump ∈ know_lump_list do
 3:     if lump.density() ≤ getRMDJ(lump) then
 4:         if toEnhance = null or lump.density() < toEnhance.density()
            then
 5:             toEnhance ← lump ;
 6: if toEnhance = null then
 7:     return ;
 8: if lump_list.size > LumpNumberLimit then
 9:     foundLumpToLeave ← false ;
10:     for all lump ∈ lump_list do
11:         if getRMDL(lump) > toEnhance.density() then
12:             foundLumpToLeave ← true ;
13:             break ;
14:     if not foudLumpToLeave then
            return ;
    joinLump(toEnhance) ;
```

Figure 2.16: Increasing the local density of a node

This comparison is done with the help of the function getRMDL. The function getRMDL, in a similar way that the function getRMDJ, simulates the lumps obtained by the fact that the node leaves the lump and returns the density of these hypothetical lumps. The pseudo code of the function getRMDJ can be found in the Appendix. The node can join the lump to increase its density only if the current density of the lump is less than the hypothetical density that will have the lump that the node has to leave.

The whole procedure described above is in the pseudo code of the function enhanceROSA

that can be found in the Figure 2.16.

Chapter 3

Distributed HashTable over ROSA

3.1 Introduction to the DHTs

DHT is an acronym that stands for **D**istributed **H**ash**T**able. In this section we introduce all the necessary concepts for the understanding of the DHT. First, we present the definition of the DHT the most commonly accepted. Then we explain the general properties and common to all the DHTs. We also see how we can class the different DHTs into several different classes. Finally we see some examples of well known DHTs.

3.1.1 Definition

An HashTable is a data structure that efficiently associates keys to data values using a hash function. One can access to a value of the table using its keys. The access to an element is performed by transforming, using the hash function, its associated key into an HashTable index value. The HashTable index value allows to retrieve the desired element.

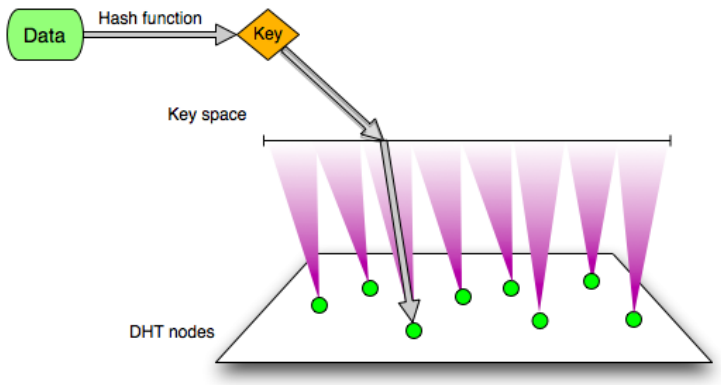


Figure 3.1: Distributed HashTable

A DHT is a decentralized system that acts as an HashTable. The key space is uniformly allocated over the set of the participating nodes in such a way that the pairs <key, value> are uniformly distributed over the nodes. In addition to this uniform distribution, the DHT provides a lookup service that allows any participating node of the DHT to efficiently

retrieve any element stored on the table.

The maintenance of the mapping and of the consistency must be a distributed process and in order to a DHT be scalable the network and CPU cycles used for nodes failures, arrivals and departures must not depends on the size of the table.

The DHTs were popularized in 2001 with the introduction of the first DHTs implementation, Chord (Morris et al. [2001]), Pastry (Rowstron and Druschel [2001]), CAN (Ratnasamy et al. [2001]) and Tapestry (Zhao et al. [2004], Zhuang et al. [2001a]). The first researches about DHT were motivated by the needs of the peer-to-peer systems. The needs of these peer-to-peer systems, mainly used for sharing files, are to achieve a complete decentralization in order to avoid disruptions due to failures and to achieve an efficient <key, value> lookup.

3.1.2 Properties

There is three main properties that the DHTs must have: scalability, correctness and fault tolerance. Other non obligatory properties exist such the anonymity of the nodes or the protection against malicious nodes. But since these properties are implemented only by few DHTs we will not deal with these in this Section.

The correctness of a DHT is achieved when, for any pair <key, value> stored over the DHT, any participating node is able to retrieve the value if the node possesses the key. It means that any data stored on the DHT can be accessed from any point of the DHT. This is not the case for the other type of distributed system. Gia (Chawathe et al. [2003]) does not give such an assurance. A resilient lookup protocol is the base of the correctness of a DHT, but a smart failures, node arrivals and departures management protocol is needed too.

The fault tolerance property is linked to the correctness, since the fault tolerance assures that the presence of failures will not prevent the lookup protocol to achieve correctly its goal. It consists in finding a substitute for the failing nodes. When a node fails, the pairs <key, value> that were stored by the failing node must be confided to another node of the DHTs. The failure tolerance also relies on the lookup protocol. This protocol must be able to operate in the presence of failures.

The scalability of a DHT is its capacity to bound its network and CPU cycle usage by a limit that does not depend on the number of nodes that participate. A scalable DHT should function efficiently with thousands of nodes. The scalability is reached by using a distributed lookup protocol and a good distribution of the keys over the nodes.

The scalability and the efficiency of the lookup protocol are intrinsically tied with the number of neighbors that a node has. The maximum number of neighbors that a node can have is called maximum node degree. The efficiency of the lookup protocol is the maximum number of hops in any route that any lookup request needs. It obvious that the length of the route between two nodes, i.e. the number of hops needed for one of these two nodes to access to a value stored by the other node, is low when the maximum node degree is high. And the lengths of the routes between any nodes of the DHT is high when the maximum degree is low. It is possible to classify the different DHTs according the maximum degree and the route length. Using the Bachmann-Landau notation we have the four classes described in the Table 3.1. Most of the implemented DHTs belong to the third class. Since the node degree of ROSA is a constant, ROSA belongs to the first class.

More details about the theoretical and measured route length in ROSA will be given in the Chapter 7 that deals with the performances.

Node degree	Route length
$O(1)$	$O(n)$
$O(log(n))$	$O(log(n)/log(log(n)))$
$O(log(n))$	$O(log(n))$
$O(\sqrt{n})$	$O(1)$

Table 3.1: The DHTs classification according to node degree and route length

3.1.3 Example

In order to gain a better understanding of what the DHTs are, we will now see two examples of DHTs. The first one, Chord (Morris et al. [2001]), is one of the four pioneer of the DHTs (Chord, CAN, Pastry, Tapestry). It was developed at MIT in 2001 by Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. The second one that will be studied is Kademlia (Morris et al. [2001]). Kademlia is famous because it is one of the most used DHT protocol. This protocol was adopted by a lot of file-sharing clients and is the first that have reached such acceptance in a worldwide file-sharing community. It was developed by by Petar Maymounkov and David Mazières.

3.1.3.1 Chord

In the Chord protocol the identifiers of the nodes are organized into a circle modulo 2^m where m is the length in bits of the identifiers. Therefore the identifier space is equal to $[0, 2^m]$. Chord can be composed of at most 2^m nodes. The identifiers of a node is computed using 'consistent hashing' (Karger et al. [1997]). The use of 'consistent hashing' to determine a node identifier ensures that each node receives the same number of keys.

Each key k, $k \in [0, 2^m]$, is confided to the first node whose the identifier is equal to or follow k. This node is the 'successor node' of the key k.

A node in Chord stores in its routing table, called finger table, the information necessary to connect to other nodes of Chord. A finger table of a node contains m entries, so a node can connect to m other nodes. So the node degree of Chord is in $O(log(N))$ where N is the number of nodes of Chord.

Let id be the identifier of a node, the i^{th} entry of finger table of this node is the first node found whose the identifier is included in:

$$\left[(id + 2^{i-1}) \mod 2^m, (id + 2^i) \mod 2^m\right)$$

. The lookup protocol of Chord consists, given a key k, in a node sending the request to the node contained in its finger table that has the identifier that is the closest to k. This process is repeated until the request reaches the targeted node. It was analytically demonstrated that the lookup can be performed in $O(log(N))$, i.e. the route length in Chord is in $O(log(N))$, where N is the number of nodes of Chord. In fact the real value,

obtained experimentally, is roughly $\frac{1}{2}log_2(N)$. The Figure 3.2 represents an instance of Chord with $m = 3$ (i.e. with 8 possible identifiers). In this figure there is 4 nodes with the identifiers 0, 1, 4 and 6. one can see the 'finger table' and the set of keys managed by the node that possesses the identifier 4.

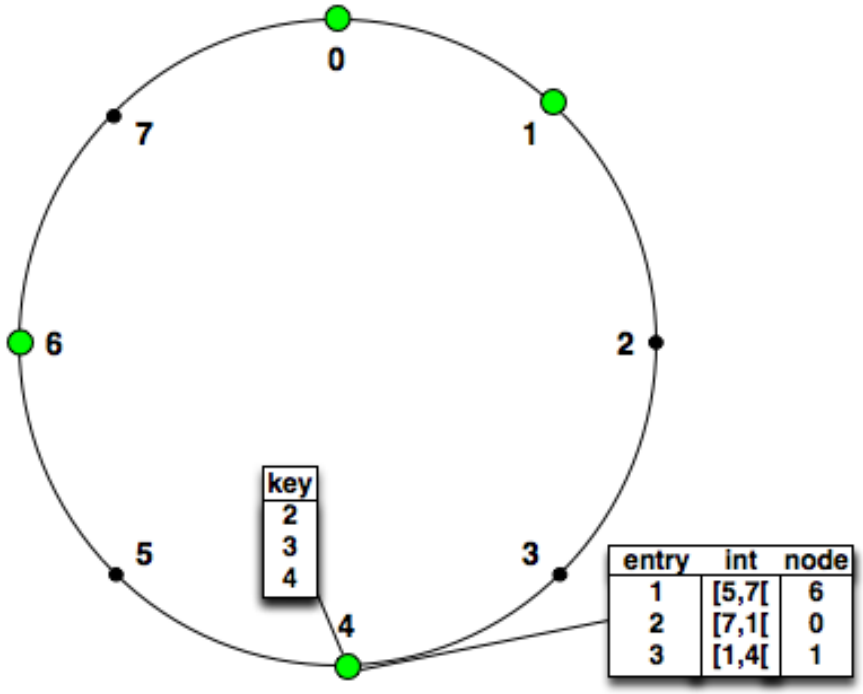


Figure 3.2: An example of Chord DHT

3.1.3.2 Kademlia

The protocol of Kademlia is based on the XOR metrics. The distance between two identifiers i_1 and i_2 is equal to $i_1 \oplus i_2$. The Kademlia identifiers system also relies on a 'consistent hashing'. The first time that a node joins Kademlia, it computes its IDs using a consistent hash function. Afterwards, this identifier will be used each time the node connects to Kademlia. In Kademlia the nodes are organized as the leafs of a binary tree where the position of a node is determined by the bits of its identifier.

Each node in Kademlia stores as many lists as the length in bits of its identifier. The size of these lists are bounded by an integer k and are so called 'k-buckets'. The k-buckets contains the necessary information for a node to join another node. Let id be the identifier of a node, the i^{th} k-buckets of this node is filled with the information concerning nodes that are at a distance (using the XOR metrics) i of the node, i.e. filled with information concerning nodes with identifiers that differ from i bits to id. The Kademlia protocol ensures that every k-buckets contains at least one entry. A node can obtain these entries by exchanging messages FIND_NODE with other nodes of Kademlia. When a k-buckets is full a node only keeps the entries of last seen nodes. The node degree of Kademlia is in

$O(log(N))$ where N is the size of the network. In the Figure 3.3 is represented the binary tree of a Kademlia DHT where the identifiers of the nodes have a length of 3 bits. In the figure is also represented the 3-buckets of the node 101.

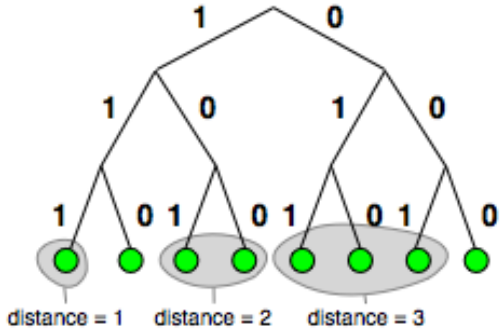


Figure 3.3: An example of a Kademlia binary tree, with the 3-buckets of the node 101.

In Kademlia the key space must be equal to the node identifier space and each pair <key, value> is assigned to the node whose the identifier is the closer of the key. In Kademlia, given a key, the lookup protocol consists in a node sending the request to the node of its k-buckets that is the closest to key. This process is run iteratively until the node that manages the key receives the request. The distance between the request and the node that manages the key is reducing by at least $1/2$ at each step. Since the height of the binary tree is in $O(log(N))$ where N is the number of nodes of Kademlia. The route length is also in $O(log(N))$.

3.2 The chain of lumps

In ROSA it is possible to organize the entanglements of lumps into a DHT. This DHT will be called the 'chain of lumps'. The 'chain of lumps' has to evolve according to the modifications that occur in ROSA. In this section we first describe the 'chain of lumps'. In second we see how the modifications of the entanglement of lumps affect the 'chain of lumps' and how the nodes react to the changes in order to maintain the 'chain of lumps'. To finish we show how the 'chain of lumps' can be used to route data packets from node to lump. The advantages of the 'chain of lumps' is that this DHT is compatible with the ROSA protocol. To use the 'chain of lumps' over ROSA it is only needed to add some information to the messages exchanged between the nodes and to perform some light modification on how the messages are handled by the nodes.

3.2.1 Description

The 'chain of lumps' is a DHT built over ROSA. The key space of this DHT is an integer interval called I_{init}. I_{init} is defined as : $I_{init} = [0, 2^{128} - 1]$. Each lump of ROSA will have to handle a part of this key space and the entanglement of lump is projected into I_{init} in order to form a chain. The projection is achieved by partitioning I_{init} into sub-intervals. These sub-intervals are attributed to the lumps in order to satisfy the following conditions:

- All the sub-intervals are allocated (1) ;
- Two lumps that possess contiguous sub-intervals share at least a common node (2) ;
- Each lump has at least one sub-interval (3).

In order to be operative the 'chain of lumps' must imperatively satisfy the conditions (1) and (2) at anytime. We will show further that any modification of the 'chain of lumps' keeps these two conditions satisfied. The condition (3) only ensures a good efficiency and load-balancing over the lumps.

Let consider a lump l that handles one sub-interval I of I_{init}. The lump l is called the lump owner of I. The lump that possesses the sub-interval just before I is called the predecessor of l. A lump and its predecessor are consecutive lumps. The lump that possess the sub-interval just after I is called lump the successor of l. A lump has as many predecessors and successors as the number of sub-intervals that it handles. In order to be more efficient the last ans first lumps of the chain are respectively predecessor and successor of each other. This way the lumps form a circular chain. Since a node can belong to LumpNumberLimit lumps, there is shortcuts in the 'chain of lumps', i.e. a way to jump from a lump to another lump without going through the lumps that are between them in the 'chain of lumps'. The Figure 3.4 shows the projection of the entanglement of lumps over I_{init}. On this figure the nodes are represented by the green circles, the lumps by the purple dashed shapes and the shortcuts by the red curves.

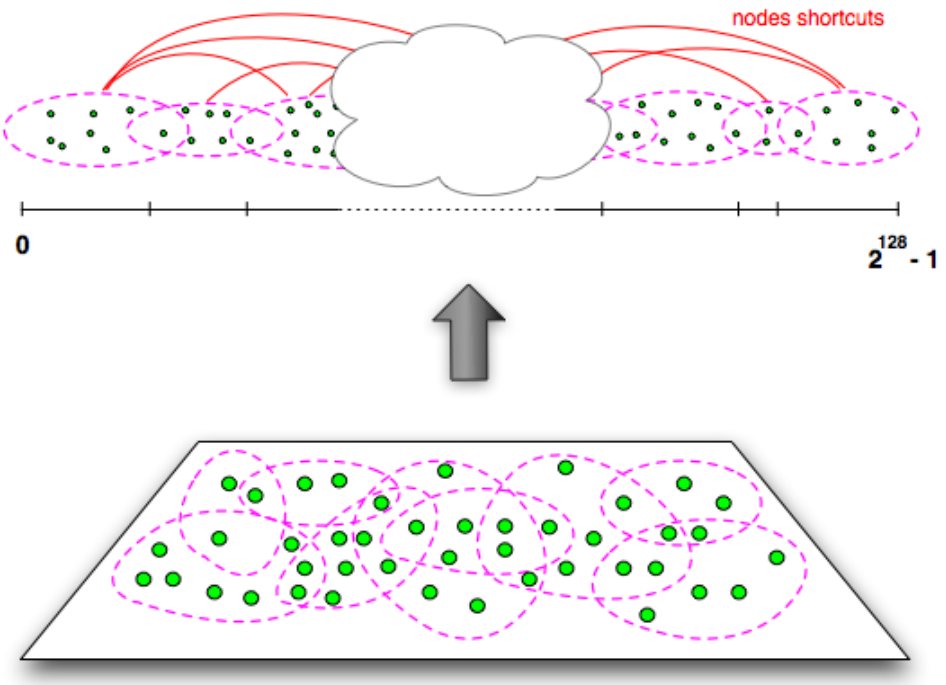


Figure 3.4: The transformation of the entanglement of lumps into a 'chain of lumps'.

The projection of the whole entanglement of lumps onto an integer interval combined with the fact that any sub-interval is allocated to a lump and the fact that any lump has a member in common with its predecessors and its successors ensures that there exists a path composed by the nodes that links the first lump to the last lump of the chain. This property is fundamental because it will guarantee that the routing algorithms proposed further in this section will terminate. The projection onto a greater dimensional space would allow for better routing quality, but the construction and maintenance of the resulting structure would have been much more complex. Consequently the gain made at the cost of routing would have been lost by the increased cost of maintenance.

The 'chain of lump' constrains to modify the representation of a lump that is shown in the Section 2.3. It is necessary to add the list of the sub-interval handled by the lump. We also have to present the representation of a sub-interval. The representation of a sub-interval is composed by the lower bound and the upper bound of the sub-interval. This representation is completed by the identifier of the lump owner, the lists of the nodes that compose the predecessor and the successor of the lump owner and a list of DHT entries. A DHT entry is value that a node wanted to store on the 'chain of lumps' and its associated key. We will see in the Section 3.2.2 that when a lump is split into two other lumps, the sub-intervals handled by the lump have to be split and shared over the two lumps resulting from the split. The two lists containing the identifiers of the nodes composing the predecessor and the successor of the owner will be helpful during this process. The representation of a sub-interval is represented in Figure 3.11.

Lump
...
subInt_list : *list<subinterval>* ; `// sub-interval handled by lump`
...

SubInterval
lowBound : *integer* ;
upBound : *integer* ;
ownerId : *integer* ;
predId_list : *list<integer>* ;
succId_list : *list<integer>* ;
DHTentry_list : *list<dhtentry>* ;

DHTEntry
key : *integer* ;
value : *data* ;

Figure 3.5: Completion of the representation of a lump and representation of a sub-interval and a DHTentry.

In the remainder of this chapter, the sub-interval that just precedes a given sub-interval I is called the predecessor of I and the sub-interval that just follows I is called the successor of I. The predecessor and the successor of I are respectively noted $I-$ and $I+$. The lump that handles a sub-interval I is denoted to l_I. We refer as the lump predecessor and the lump successor of a sub-interval I to respectively the lump handling $I-$ and the lump

handling $I+$. Consequently in regard to a sub-interval I, we refer to the lump predecessor of l_I as l_{I-} and to the lump successor of l_I as l_{I+}. The list of the identifiers of the nodes that compose l_{I-} is named the predecessor list and the list of the identifiers of the nodes that compose l_{I+} is named the successor list.

3.2.2 Building and maintaining the 'chain of lumps'

The ROSA protocol (described in Chapter 2) consists in nodes leaving some lumps in order to join other lumps and increase the overall density. When a lump is composed by too much nodes, this lump has to be split into two smaller lumps. The lumps are also split when a virtual link is broken.

When the set of the nodes that compose a lump includes the set of the nodes composing another lump, the first lump absorbs the second one. The absorption of a lump by another one may happen after a join. Let us consider two lumps l_1, l_2 with their corresponding set of nodes s_1 and s_2 ($s_1 \neq s_2$ and $s_1 \not\subset s_2$). Considers that there exists a node $n_1 \in s_2$ such that $s_1 \cup n_1 \subset s_2$. If n_1 joins l_2 the lump l_2 absorbs l_1. Here is an example of a join leading to a lump absorption, let the set of the nodes that compose l_1 be $\{n_1, n_2\}$ and the set the nodes that compose l_2 be $\{n_2, n_3\}$. If the node n_1 joins the lump l_2, the set of nodes of l_2 becomes $\{n_1, n_2, n_3\}$ and includes the set of the nodes of l_1. Therefore the lump l_2 absorbs the lump l_1.

The absorption of a lump may also happen after a split. Let us consider two other lumps l_3, l_4 and their corresponding set of nodes s_3 and s_4. Let $s_{common} \subset s_3$ and n_4 a node such that $n_4 \notin s_3$ and $s_4 = s_{common} \cup n_4$. If the the lump l_4 is split around n_4 and another node n of s_4 two lumps are created. The set of nodes of the first lump is $s_4 - n_4 = s_{common} \subset s_3$ and the set of nodes of the second lump is $s_4 - n$. Therefore l_3 will absorb the first lump resulting from the split. Below is an example of a split leading to a lump absorption, considers that the sets of the nodes that compose the lumps l_3 and l_4 are respectively equal to $\{n_1, n_2, n_3\}$ and $\{n_1, n_2, n_4\}$. Let us consider that the virtual link between n_1 and n_3 is broken, the lump l_3 has to be split around these two nodes. The lumps that results from the split are l_{31} and l_{32} respectively composed bu the nodes of the sets $\{n_2, n_3\}$ $\{n_1, n_2\}$. The lump l_{32} is absorbed by l_4.

In ROSA new lumps are only created during the split of an already existing one and the lumps only disappear during this absorption process. The 'chain of lumps' must react and reconfigure itself according to the splits and the absorptions. The initialization of the 'chain of lumps' is presented in the Section 3.2.2.1. In the Sections 3.2.2.2 and 3.2.2.3 we see how the 'chain of lumps' handles the absorptions and the splits of the lumps. The 'chain of lumps' constrains to complete and modify some functions of the initial ROSA protocol. These changes are described in this Chapter.

3.2.2.1 Initializing the 'chain of lumps'

During the initialization of ROSA the first node creates the first lump. The first node has to handle the initial interval I_{init}. As it is shown further, the sub-intervals handled by the lumps could be split during the split of the lumps handling them but neither the absorptions nor the splits must cause the loss of some sub-intervals, consequently, this implies that the condition (1) satisfied. The initialization of the 'chain of lumps' constrains to modify the

initROSA() function as described in the Figure 3.6.

Node.initROSA

...
6: I_{init}.lowBound $\leftarrow 0$;
7: I_{init}.upBound $\leftarrow 2^{128} - 1$;
8: I_{init}.owner $\leftarrow$ lump.id ;
9: I_{init}.predId_list $\leftarrow$ lump.nodeId_list ;
10: I_{init}.succId_list $\leftarrow$ lump.nodeId_list ;
11: lump.subInt_list $\leftarrow$ subInt_list $\cup$ I_{init};
...

Figure 3.6: The completion initROSA function for the 'chain of lumps'

These modifications consist in setting the first lump as the owner of I_{init}. Since there is, for the present a single lump, in ROSA and according to the definition of the 'chain of lump' this first lump is its own predecessor and its own successor. That is why the lists of the identifiers of the nodes that compose the predecessor and the successor is equal to the list of the identifiers of the nodes of the first lump. To finish the node that initiates ROSA adds the representation of I_{init} to the list of the sub-intervals handled by the first lump.

3.2.2.2 Reacting to the absorptions of a lump

When a lump is absorbed by another one, the sub-intervals handled by the absorbed lump must be handled by the absorbing one. This is schematized in the Figure 3.7. In the upper part of the figure one can see two lumps l_{I-} and l_I and a part of 'the chain of lumps'. The lump l_{I-} handles the sub-interval $I- = [a, b)$ and the lump l_I handles the sub-interval $I = [b, c)$. The node n joins the lump l_I. Consequently the set of the nodes that compose l_I includes the set of the nodes that compose l_{I-} and the lump l_I absorbs the lump l_{I-}. On the lower part on the figure one can see that the sub-interval handled by l_{I-} is given to the lump l_I. After the join, the lump l_I have to handle the sub-interval $[a, c)$.

The detection and the management of the absorptions of the lumps are dealt in the Section 2.4.10. We have seen that it consists in a node checking if one of the lumps of its list of lumps can absorb another one. If such a pair of lumps is found the node sends messages *AbsorbLump* to the nodes that compose the absorbing lump. The 'chain of lumps' forces to modify the content of the messages *AbsorbLump* and the way how the nodes handle these.

A message *AbsorbLump*, that is sent to notice that a lump absorbs another one, contains the identifiers of the absorbed and absorbing lumps. In order to be compatible with the 'chain of lumps' it is necessary to add the list of the sub-intervals that was handled by the absorbed lump to the message *AbsorbLump*. When a node receives a message *AbsorbLump*, it removes the absorbed lump from its list of lumps and adds the sub-intervals contained in the message to the list of the sub-intervals that the absorbing lump handles. The node also has to check if the absorbing lump does not handle now two contiguous sub-intervals. If contiguous sub-intervals are found, then these sub-intervals have to be merged. Let us

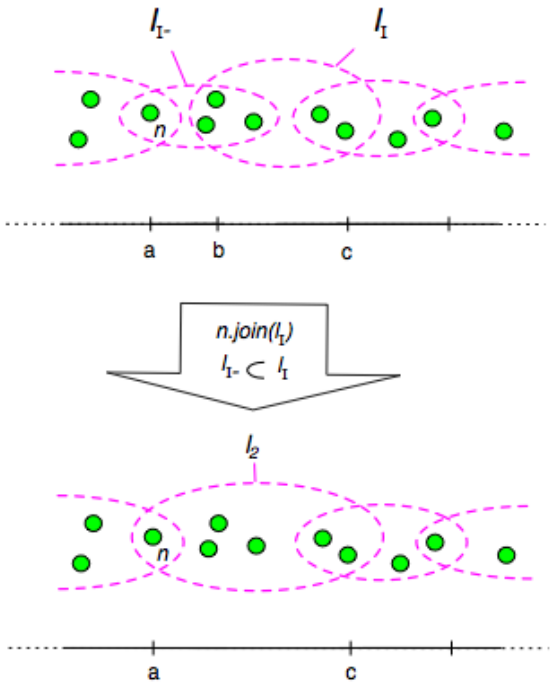


Figure 3.7: An absorption and its impact on the 'chain of lumps'

consider two contiguous sub-intervals I and $I+$, such that $I+$ is the sub-interval that just follows I in the 'chain of lumps'. To merge I and $I+$ a node creates a new sub-interval I_{merged} whose the lower bound is set to the lower bound of I and the upper bound is set to the upper bound of $I+$. The predecessor and successor lists of I_{merged} are respectively the predecessor list of I and the successor list of $I+$.

The modified and completed pseudo code concerning the absorption of a lump is shown in the Figure 3.8. The functions contiguous and merge are described in the Appendix. The function contiguous takes two sub-intervals and returns true if the sub-intervals are contiguous. The function merge takes two contiguous sub-intervals in argument and returns the sub-interval that results from the merge of the arguments.

3.2.2.3 Reacting to the split of a lump

When a lump is split into two new lumps, the sub-intervals handled by the split lump must be handled by the two new lumps. Some of these sub-interval have to be split and distributed to the lumps that result from the split. Some of these, have to be confided (without splitting these) to the resulting lumps. We see in first in this Section, how a sub-interval is split and distributed and we also see in which case a sub-interval cannot be split. Afterwards we see in which cases it suits to split and to distribute a sub-interval and in which cases the best solution is to keep a sub-interval unchanged and to confide it to one of the lumps that result from the split.

A lump l possesses a predecessor lump and a successor lump, for each sub-interval $I = [a, b)$ that it handles. Let us call l_{I-} the lump predecessor of l and l_{I+} the successor of

```
Messages handling
   ...
1: upon receive AsorbLump(absorbedLumpId ,
                          absorbingLumpId,
                          subInterval_list  )
   do
2:   lump_list ← lump_list - lump_list.get(absorbedLumpId) ;
3:   lump ← lump_list.get(absorbingLumpId) ;
4:   for all subint1 ∈ lump.subInt_list do
5:     for all subint2 ∈ subInterval_list do
6:       if contiguous(subint1, subint2) then
7:         subint2 ← merge(subint1, subint2) ;
8:         lump.subInt_list ← lump.subInt_list - subint1 ;
9:     subint2.owner ← lump.id ;
10:    lump.subInt_list ← lump.subInt_list ∪ subint2 ;
   ...
```

Figure 3.8: Handling the absorption of a lump

l corresponding to I. Considers now the set of the nodes that compose l_I, l_{I-} and l_{I+}, let us call these subsets s_I, s_{I-} and s_{I+}. If we refer to the definition of the 'chain of lumps' we have $s_{I-} \cap s_I \neq \emptyset$ and $s_I \cap s_{I+} \neq \emptyset$. One can remark that this property is true when the 'chain of lumps' is created. Indeed since at the beginning I_{init} is handled by a single lump, this single lump is its own predecessor and successor. From the split of the lump l_I results two lumps l_1 and l_2. The set of the nodes of l_1 is s_1 and the set of the nodes of l_2 is s_2. If the size of l_I is greater than 2 then $s_1 \cap s_2 \neq \emptyset$. The sub-interval $I = [a, b)$ can be split into two other sub-intervals $I' = [a, \lfloor \frac{a+b}{2} \rfloor)$ and $I'+ = [\lfloor \frac{a+b}{2} \rfloor, b)$ if one of the two conditions:

$$s_{I-} \cap s_1 \neq \emptyset \text{ and } s_{I+} \cap s_2 \neq \emptyset \quad (4)$$

$$s_{I-} \cap s_2 \neq \emptyset \text{ and } s_{I+} \cap s_1 \neq \emptyset \quad (5)$$

is respected. It may happen that none of the conditions (4, 5) is satisfied. In this case I cannot be split. However, since $s_{I-} \cap s_I \neq \emptyset$ and $s_I \cap s_{I+} \neq \emptyset$ it means that either:

$$s_{I-} \cap s_1 \neq \emptyset \text{ and } s_{I+} \cap s_1 \neq \emptyset \quad (6)$$

or

$$s_{I-} \cap s_2 \neq \emptyset \text{ and } s_{I+} \cap s_2 \neq \emptyset \quad (7)$$

In the first case I must be given to l_1 and else must be given to l_2. Else if one of the conditions (4, 5) is satisfied, I' and $I'+$ could be given to l_1 and l_2. This is schematized in the Figure 3.9.

In the upper part of this Figure, the node n_1 detects that the virtual link with the node n_2 is broken. The lump l_I that is composed by n_1, n_2 and some other nodes has to

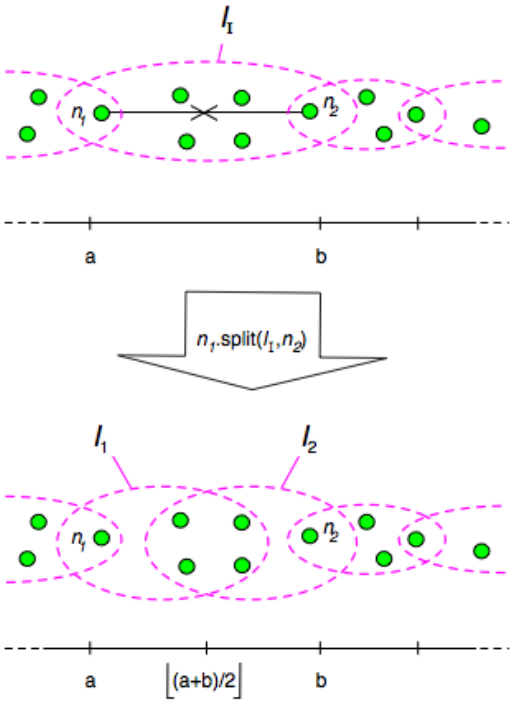


Figure 3.9: A split and its impact on the 'chain of lumps'

be split around n_1 and n_2. It results from this split two lumps l_1 and l_2. In the lower part of the Figure The sub-interval $[a, b)$ is split in two sub-intervals and the left one is given to l_1 while the right one is given to l_2.

Since the conditions (4, 5) could be true together it is necessary to define a strategy for the distribution of I' and $I'+$. Two distributions are possible. In the distribution 1, the sub-intervals I' and $I'+$ are confided to respectively l_1 and l_2. In the distribution 2, the sub-intervals I' and $I'+$ are confided to respectively l_2 and l_1. This strategy is summarized in the Table 3.2. The cells with 'confided' in, correspond to the cases where the sub-interval cannot be split. In this cases the sub-interval is given to the resulting lump that can handle the sub-interval. A lump can handle a sub-interval I if it shares some nodes with the two lumps that handle the sub-intervals that are contiguous to I. The cells in grey corresponds to the cases where the sub-interval cannot neither be split nor confided to one of the resulting lumps. In this cases the chain of lumps is broken because the condition (2) is not satisfied anymore. However, it is demonstrated in the Section dealing with the operating proof (Section 3.2.4) that, if before the split the condition (2) is satisfied and if the size of the split lump is greater than 2, these cases cannot be encountered. The bottom left cell corresponds to the case where the conditions (4, 5) are both satisfied. In this case the sub-intervals resulting from the split can be distributed following the both way, therefore we follow the way that maximize the number of nodes in common between the resulting lumps and the predecessor and successor.

The number of sub-intervals that compose the 'chain of lumps' has an effect on the efficiency of the routing algorithm. The greater the number of sub-intervals is, the greater

	$s_1 \cap s_{I-} = \emptyset$ and $s_1 \cap s_{I+} = \emptyset$	$s_1 \cap s_{I-} \neq \emptyset$ and $s_1 \cap s_{I+} = \emptyset$	$s_1 \cap s_{I-} = \emptyset$ and $s_1 \cap s_{I+} \neq \emptyset$	$s_1 \cap s_{I-} \neq \emptyset$ and $s_1 \cap s_{I+} \neq \emptyset$
$s_2 \cap s_{I-} = \emptyset$ and $s_2 \cap s_{I+} = \emptyset$				confided to l_1
$s_2 \cap s_{I-} \neq \emptyset$ and $s_2 \cap s_{I+} = \emptyset$			distribution 2	distribution 2
$s_2 \cap s_{I-} = \emptyset$ and $s_2 \cap s_{I+} \neq \emptyset$		distribution 1		distribution 1
$s_2 \cap s_{I-} \neq \emptyset$ and $s_2 \cap s_{I+} \neq \emptyset$	confided to l_2	distribution 1	distribution 2	if $min\#(s_1 \cap s_{I-}, s_2 \cap s_{I+}) > min\#(s_2 \cap s_{I-}, s_1 \cap s_{I+})$ distribution 1 else distribution 2

Table 3.2: The sub-intervals distribution strategy

the number of hops needed to perform a lookup is. That is why it is interesting to reduce the number of the sub-intervals that compose the 'chain of lumps'. Ideally the number of sub-intervals of the 'chain of lumps' must be equal to the number of lumps in ROSA. Accordingly, the splits of the sub-intervals have to be avoided when it is possible. Let us consider that a lump l_I is split into l_1 and l_2. The split of one of its sub-intervals I is avoidable if I can be handled by l_1 or l_2 and if l_1 and l_2 handle each one at least one sub-interval after that all the sub-intervals handled by l_I have been split and distributed or confided to l_1 or l_2.

Once ROSA endowed with the 'chain of lumps', a node that wants to split a lump must deal with the sub-intervals that this lump handles. If the lump handles only one sub-interval, this sub-interval is split and distributed to the lumps resulting from the split (or simply confided to one of these lumps if the sub-interval cannot be split). If the lump that is split handles more than one sub-interval, for each of the sub-intervals the node determines if the sub-interval can be handled by one or both of the resulting lumps or if the sub-interval must be split. Once it is done, the nodes shares out the sub-intervals over the two resulting lumps in such a way that both of the resulting lumps have at least one sub-interval and that any sub-intervals is confided to a lump that can handle it. This distribution is done in order to minimize the difference between the number of sub-intervals handled by each of the two resulting lumps. During this process a node must split a sub-

```
Node.split(lump, ids) (Input:lump, pair<integer>)
 1: lump_1 ← lump.without(ids[1]) ;
 2: lump_2 ← lump.without(ids[2]) ;
 3: for all subinterval ∈ lump.subInt_list do
 4:     which ← none ;
 5:     if lump_1.canHandle(subinterval) then  which ← l_1;
 6:     if lump_2.canHandle(subinterval) then
 7:         if which = l_1 then  which ← both;
 8:         else  which ← l_2 ;
 9:     switch which do
10:         case none
11:             subinterval.splitOver(lump_1, lump_2) ;
12:         case l_1
13:             lump_1.subInt_list ← lump_1.subInt_list ∪ subinterval ;
14:         case l_2
15:             lump_2.subInt_list ← lump_2.subInt_list ∪ subinterval ;
16:         case both
17:             if lump_1.subInt_list.size() < lump_2.subInt_list.size() then
18:                 lump_1.subInt_list ← lump_1.subInt_list ∪ subinterval;
19:             else
20:                 lump_2.subInt_list ← lump_2.subInt_list ∪ subinterval ;
21: for i ∈ {1,2} do
22:     if lump_i.subInt_list.size() = 0 then
23:         for all subinterval ∈ lump_{i+1 mod 2}.subInt_list do
24:             if subinterval.canSplitOver(lump_1, lump_2) then
25:                 lump_{i+1 mod 2}.subInt_list ←
                            lump_{i+1 mod 2}.subInt_list - subinterval ;
26:                 subinterval.splitOver(lump_1, lump_2) ;
27:                 return send(SplitLump(lump.id, lump_1, lump_2)) to lump ;
28: return send(SplitLump(lump.id, lump_1, lump_2)) to lump ;
```

Figure 3.10: The modified split function

interval only if the sub-interval cannot be handled by one of the resulting lumps or if one of the resulting lumps does not handle any sub-interval. The pseudo code of the modified split function is shown in the Figure 3.10. The pseudo code of the functions splitOver, canSplitOver and canHandle can be found in the Appendix. The splitOver function splits a sub-interval into two contiguous sub-intervals and distributes these over the two lumps given in argument and according to the strategy resumed in the Table 3.2. This function

also shares out the DHT entries of the sub-interval split to the resulting sub-intervals. The function canSplitOver returns true if the sub-interval can be distributed following the distribution 1 or the distribution 2 and else it returns false. The function canHandle returns true if the lump can handle the sub-interval given in argument and else it returns false.

3.2.2.4 Maintaining the predecessors and successors nodes identifiers lists

In order to determine if and how a sub-interval I can be split, the predecessor and successor lists are essential. However it may happen that these lists are not up to date. Let us consider a lump l_I that handles the sub-interval I and a node n_i that wants to join l_I. After that the node n_i joins l_I the successor list corresponding to $I-$ and the predecessor list corresponding to $I+$ are not up to date anymore. Indeed the identifier of n_i is not an element of these lists. Consider now that a node n_j leaves l_I, in this case the identifier of the node n_l has to be removed from the predecessor and successor lists mentioned above. The absorption of a lump may also lead to an inadequacy of the predecessor and successor lists of some sub-intervals.

keepSubIntUTD

```
1:  for all lump_I ∈ lump_list do
2:      for all lump_I+ ∈ lump_list do
3:          if succeed(lump_I, lump_I+) then
4:              for all pair ∈ getSuccSubIntPair(lump_1, lump_2) do
5:                  if not checkPredList(lump_I, lump_I+, pair) then
6:                      send UpdtPred(lump_I.nodeId_list, pair[0].lowBound)
                        to lump_I+;
7:                  if not checkSuccList(lump_I, lump_I+, pair) then
8:                      send UpdtSucc(lump_I+.nodeId_list, pair[1].lowBound)
                        to lump_I;
```

Messages handling

```
    ...
9:  upon receive UpdtPred(Id_list, bound) do
10:     sub_int ← findSubInt(bound) ;
11:     sub_int.predId_list ← Id_list ;
12: upon receive UpdtSucc(Id_list, bound) do
13:     sub_int ← findSubInt(bound) ;
14:     sub_int.succId_list ← Id_list ;
    ...
```

Figure 3.11: Maintaining the predecessor and successor lists of the sub-intervals

To keep these lists up to date, a function has to be added to the main loop of ROSA. This is the function keepSubIntUTD. This function consists in a node checking if it belongs

to both a lump and its successor. Let us consider that the node finds such a couple of lumps and let us call l_I the first lump and l_{I+} its successor. The node looks down the list of the sub-interval handled by these two lumps in order to find the concerned pairs of sub-intervals. It may happen that there is more than one pair if the lumps handle more than one sub-interval. For each of these pairs $<I, I+>$, the node checks that the list of the identifiers of the nodes composing l_I is equal to the predecessor list of $I+$. If these lists are not equal, the node sends a message *UpdtPred* to the lump l_{I+}. This message contains the list of the identifiers of the nodes that compose l_I and the lower bound of I. Afterwards, the node checks that the list of the identifiers of the nodes composing l_{I+} is equal to the successor list of I. If these lists are not equal, the node sends a message *UpdtSucc* to the lump l_{I+}. This message contains the list of the identifiers of the nodes that compose l_{I+} and the lower bound of $I+$. The node can performs these checks and builds such messages because it belongs to both l_I and l_{I+}. A node receiving a message *UpdtPred* or *UpdtSucc* updates the predecessor or successor list of the corresponding sub-interval with the list of the identifiers contained in the message. The pseudo code of the function keepSubIntUTD is shown in Figure 3.11. The function succeed determines if the second lump given in argument is a successor of the first lump given in argument. This function returns true if this is the case and false else. The function getSuccSubIntPair takes two lumps in argument and returns a list of pairs of sub-intervals. These pairs are composed of a sub-interval and its successor, the first one must belong to the first lump given in argument and the second one must belong to the second lump given in argument. The functions checkPredList and checkSuccList return true if the corresponding predecessor and successor list are up to date and return false else. The pseudo-code of these functions can be found in the Appendix.

3.2.3 Using the 'chain of lumps'

In a DHT any participating node can store a <key, value> pair over the DHT. Generally, this pair will be stored by a node of the DHT. In the 'chain of lumps' a given <key, value> pair is stored by all the nodes of a lump. This confers to the 'chain of lumps' a better failure tolerance. Indeed, with the DHT in which a pair is only stored by one node, the failure of this nodes leads to the loss of all the pairs that the node manages. With the 'chain of lumps' the failure of a node does not cause such a loss. In the remainder of this section, we refer to the lump that handles the sub-interval that contains a given key k as the lump that handles k. And we said that a node handles a key k if this node belongs to the lump that handles k.

In this Section we see how a node can store a <key, value> pair over the 'chain of lumps' (see 3.2.3.1). Afterwards we see how a node proceeds to retrieve a pair stored over the 'chain of lumps' (see 3.2.3.2). Then we see how we can generalized these processes to send data packets to the nodes that handle a given key (see 3.2.3.3). To finish, we explain how we deal with the nodes that do not handle any key (see 3.2.3.4).

3.2.3.1 Storing a <key, value> pair

In this section we see how a node can store a <key. value> pair over the 'chain of lumps'. To store a value over the 'chain of lumps', a node must first compute the key associated to

the value. This key is an integer that must belong to $[0, 2^{128} - 1)$. In order to compute this key we can use an existing hash function on the bytes of the value. The key is finalized by performing a modulo 2^{128} over the value given by the hash function. Once this key is computed, the node can start the store process.

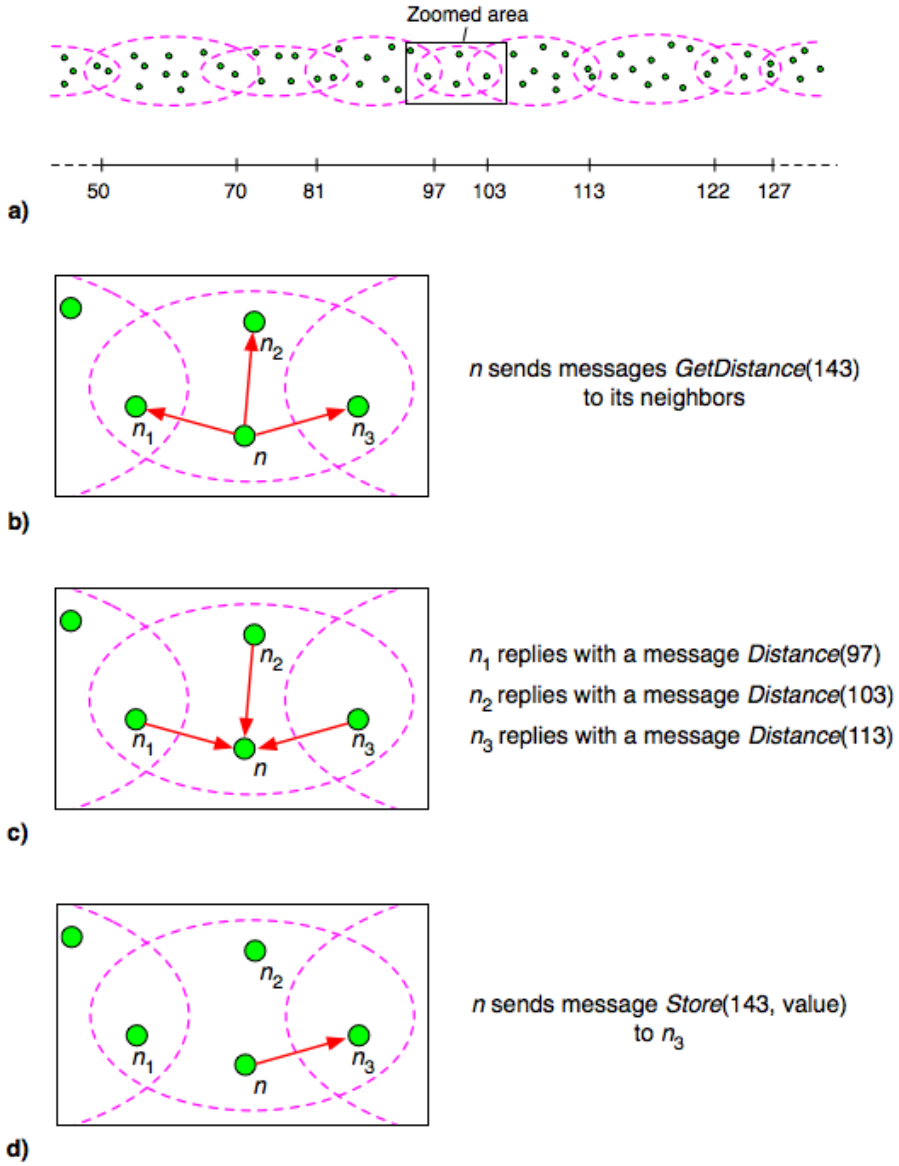


Figure 3.12: A step of the store process

The first step of the store process consists in a node n checking if the node handles the key. If the node handles this key, the node builds a message *StoreHere*. Let the sub-interval containing the key be I_{key} and the lump handling the key be l_{Ikey}. The message *StoreHere* contains the <key, value> pair and the lower bound of I_{key}. The node n sends this message to l_{Ikey}. A node that receives a message *StoreHere*, looks for the sub-interval $Ikey$ whose the lower bound is equal to the one contained in the message. Once this sub-interval is

found, the node completes the list of the <key, value> pairs of $Ikey$ with the <key, value> pair contained in the message.

If the node n that wants to store the <key, value> does not handle the key, it sends a message *GetDistance* to its neighbors. This messages only contains the key. A node that receives such a message retrieves the key contained in the message and computes the distance between itself and the key. The distance between a node n and a key k is defined as the distance minimum between the lumps of its list of lumps, the distance between a lump l and a key k is the minimum distance between k and the sub-intervals handled by l and the distance between a sub-interval I and the key k is defined as follows:

$$dist(I,k) = \begin{cases} min(|k - I.lowBound|,\ |k - I.upBound|) & \text{if} \quad k \notin I \\ 0 & \text{if} \quad k \in I \end{cases}$$

Once the distance is computed a node that received a message *GetDistance* replies with a message *Distance*. This message contains the computed distance. The node n that sends a message *GetDistance* eventually receives from its neighbors a set of messages *Distance*. Then n selects the neighbor that sent the message that contains the smallest distance and builds a message *Store*. This message contains the <key, value> pair that the node wants to store. Finally n sends this message to the selected neighbor. A node that receives a message *Store* repeats the whole process. This process will be repeated until the <key, value> pair reaches the appropriate lump. The figure 3.12 shows a step of this process. In the top of this figure a part of the 'chain of lumps' is represented. In order to simplify the figure we deliberately ignore the shortcuts, i.e. we consider that each node of ROSA only belongs to only one lump. In the rest of the figure, we focus on the node n. This node n wants to store a value on the 'chain of lumps'. The corresponding key of this value is 143. In **b)** n sends the messages *GetDistance* to its neighbors. In **c)** each of the neighbors of n replies with a message *Distance*. In **d)**, n sends the message *Store* to the neighbors that is the closest to 143. This neighbor is n_3.

3.2.3.2 Retrieving a value

To retrieve a value stored over the 'chain of lumps' a node n must know the key k that corresponds to the value. If n knows k, it can start the lookup process. First n checks if it handles k. In this case the node can retrieve from the lump that handles k the wanted value. Else the node builds and sends a message *Lookup*. This message contains the key k, the identifier and the location of n. The location of a node is constituted by its identifier and the bounds of the sub-intervals handled by the lumps whose the node belongs.

A node that receives a message *Lookup* checks if it handles k. If the node handles this key, it retrieves the value that corresponds to k, and builds a message *DataPacket* containing the wished value. Afterwards, the node sends the message to the node n. In order to be able to send the message, the node finds a key that is handled by n with the help of the location of n that is contained in the message *Lookup*. Once such a key is found, the node sends the retrieved value to n as in the process described in the Section 3.2.3.3.

If a node that receives a message *Lookup* does not handle the key k, it sends a message *GetDistance* to its neighbors. Therefore, the node selects the neighbor that replies with the message *Distance* that contains the smallest distance, and forwards the message *Lookup* to

this neighbor. This process will be repeated until the message *Lookup* reaches a node that handles the key k.

3.2.3.3 Sending data packets to a given lump

It is possible to extend the store process in order to allow a node to send data packets to the nodes handling a given key. This process can be used to build a resilient file storage. It will also used to build algorithm that enables the routing from node to node.

```
sendDataPacket(key, packet) (Input:integer, data)
 1: for all lump ∈ lump_list do
 2:     if lump.handle(key) then
 3:         return send DataPacket(key, packet) to lump ;
 4: return forwardDataPacket(key, packet) ;
forwardDataPacket(key, packet) (Input:integer, data)
 5: for all neighbor ∈ neighbor_list do
 6:     send GetDistance(key) to neighbor.phy ;
 7: wait until rcv_list(Distance(distance)).size() = neighbor_list.size() do
 8:     nextHop ← null ;
 9:     distance_nextHop ← infinity ;
10:     for all message ∈ rcv_list(Distance(distance)) do
11:         if message.distance < distance_nextHop then
12:             distance_nextHop ← message.distance ;
13:             nextHop ← getSender(message) ;
14:     return send DataPacket(key, packet) to nextHop ;
Messages handling
    ...
15: upon receive DataPacket(key, packet) do
16:     if handle(key) then
17:         sender ← getSender(DataPacket(key, packet)) ;
18:         if not sender.handle(key) then
19:             sendDataPacket(key, packet) ;
            /* extract data from packet */
20:         return ;
21:     return forwardDataPacket(key, packet) ;
    ...
```

Figure 3.13: Sending a data packet to the nodes handling a given key

A node n that wants to send data packets to the nodes that handles a given key k builds a message *DataPacket*. This message contains the key k and the data to convey.

Afterwards, the node checks that it does not belong to the lump that handles k. If n belongs to such a lump, it sends the message *DataPacket* to all the nodes that compose the lump. If the node does not belong to such a lump, it sends messages *GetDistance* to its neighbors, receive a set of messages *Distance*, selects the neighbor that is the closest to k and forwards the message *DataPacket* previously built to this neighbor.

When a node receives a message *DataPacket* it checks if it handles the key k contained in the message. If the node handles this key, it first extracts the data contained in the message. Afterwards, the node checks if the node that sends the message also handles k. This verification consists in looking if the sender node is one of the nodes that compose the lump that handles k. If the sender node does not handle k, the node that received the message *DataPacket* forwards it to all the nodes that compose the lump that handles k. If the node that received the message *DataPacket* does not handle k, the node repeats the process mentioned above. This process is repeated until the message *DataPacket* reaches one node that handles k.

The pseudo code corresponding to this process is shown in the Figure 3.13. The function handle takes a key as argument and returns true if the caller, node or lump, handles the key. It returns false else. The rcv_list(*Distance*(distance)) is filled with the messages *Distance*(distance) received by the node.

3.2.3.4 Dealing with the nodes do not handle any keys

It may happen that after the split of a lump, in the worst case, one of the resulting lumps does not handle any sub-interval and it may also happen that a node only belongs to lumps without sub-interval.

In this case a node is not able to compute the distance between itself and a given key. Consequently, it is possible that all the neighbors of a node are in such a case. The process described in 3.2.3.3 cannot be used by the node to send data packets. If this node wants to send a data packet to the nodes handling a given key, it has to build a message *RandomDataPacket*(key, data) to one of its neighbor. This message contains the key and the data to convey. When a node receives such a message, the node checks if it handle a key. If this is not the case, the node forwards the message to one of its neighbors randomly chosen. If the node handles a key, it starts the procedure described in 3.2.3.3 to send the data packet to the nodes handling the key. This is detailed in the pseudo-code of the Figure 3.14. The function getRand returns a random element of the caller. The random walk algorithm can be replaced by a flooding algorithm. With the flooding algorithm the message *RandomDataPacket*(key, data) reaches a node handling a key faster than with the random walk algorithm. However, the flooding algorithm generates more network usage.

When a node does not belong to a lump that handles a sub-interval it does not have any location. Let us recall that the location of a node is the set of the bounds of the sub-intervals handled by the lumps whose the node belongs. The node location is for example needed to retrieve a <key, value> pair stored on the 'chain of lumps'. Let us consider a node n, that does not handle any key, wishing to retrieve a value stored over the 'chain of lumps'. The node n sends a message *RandomLookup* to one of its neighbors randomly chosen. A message *RandomLookup* is similar as a message *Lookup* described in 3.2.3.2 excepted that it does not has any location field. The messages *RandomLookup* are routed as the messages *RandomDataPacket*. When a message *RandomLookup* is received by a

```
sendRandomDataPacket(key, packet) (Input:integer, data)
1: neighbor ← neighbor_list.getRand() ;
2: send RandomDataPacket(key, packet) to neighbor ;
Messages handling
   ...
3: upon receive RandomDataPacket(key, packet) do
4:     for all lump ∈ lump_list do
5:         if lump.subInt_list.size() ≠ 0 then
6:             return sendDataPacket(key, packet) ;
7:     neighbor ← neighbor_list.getRand() ;
8:     send RandomDataPacket(key, packet) to neighbor ;
   ...
```

Figure 3.14: Sending a data packet from a lump that does not handle any key

node handling a key, let n_{relay} be this node, n_{relay} transforms it into a message *Lookup* with its own location in. When the message *DataPacket*, containing the value that has to be retrieved, reaches n, the node n_{relay} builds a message *FloodDataPacket* containing the value and the identifier of n. Afterwards, this message is flooded over the nodes that do not handle any key until the message reaches n. In a more general way a node that wants to be personally contacted by another node must communicate its location and this is not possible nodes that do not handle any key.

3.2.4 Operating proof

We have seen in the Section 3.2.1 that the 'chain of lumps' must satisfy the conditions (1) and (2) in order to be operative. In the Section 3.2.4.1 we see that the condition (1) is always respected since the whole interval initial I_{init} is shared over the entanglement of lumps. In the Section 3.2.4.2 we see under which hypothesis we are sure that the condition (2) is satisfied.

3.2.4.1 All the sub-intervals are allocated

At the initialization of the 'chain of lumps' the initial interval I_{init} is confided to the first lump. Afterwards, the sub-intervals are split or merged according to the modifications of the entanglement of lumps. A sub-interval may be split when its lump owner is split and two sub-intervals may be merged when a lump absorbs another one. But at no moment, a sub-interval can disappear or be lost if the nodes strictly follows the protocol. Consequently, it is obvious that the condition (1) is always satisfied.

3.2.4.2 A lump and its successor share at least a common node

Let l_I be a lump and s_I the set of the nodes that compose l_I such that:

$$s_I = \{n_1, n_2\} \cup s'_I$$

where n_1 and n_2 are some ordinary nodes that compose l_I and s'_l is the set of the other nodes that compose l_I.

From the split of a lump l_I around the two arbitrary nodes n_1 and n_2 results two lumps l_1 and l_2 with their corresponding sets of nodes s_1 and s_2 such:

$$s_1 = \{n_1\} \cup s'_I \text{ and } s_2 = \{n_2\} \cup s'_I$$

In regard with a sub-interval I owned by l_I, let l_{I-} and l_{I+} be respectively the lump predecessor and lump successor of l_I and their corresponding sets of nodes s_{I-} and s_{I+}. Let us consider that the condition (2) is satisfied, we have:

$$s_{I-} \cap s_I \neq \emptyset \text{ and } s_I \cap s_{I+} \neq \emptyset$$

Proposition. If the condition (2) is satisfied, the split of lump with a size greater than 2 will let the condition (2) satisfied.

Proof. If the size of the lump l_I is greater than two we can affirm that:

$$s_1 \cap s_2 = s'_I \neq \emptyset$$

therefore the condition (2) still be satisfied if:

$$s_{I-} \cap s_1 \neq \emptyset \text{ or } s_{I-} \cap s_2 \neq \emptyset$$

and:

$$s_1 \cap s_{I+} \neq \emptyset \text{ or } s_2 \cap s_{I+} \neq \emptyset$$

Since $s_{I-} \cap s_I \neq \emptyset$ and $s_I = s_1 \cup s_2$ we have:

$$s_{I-} \cap s_1 = \emptyset \Rightarrow s_{I-} \cap s_2 \neq \emptyset$$

and

$$s_{I-} \cap s_2 = \emptyset \Rightarrow s_{I-} \cap s_1 \neq \emptyset$$

And since $s_I \cap s_{I+} \neq \emptyset$ and $s_I = s_1 \cup s_2$ we have:

$$s_1 \cap s_{I+} = \emptyset \Rightarrow s_2 \cap s_{I+} \neq \emptyset$$

and

$$s_2 \cap s_{I+} = \emptyset \Rightarrow s_1 \cap s_{I+} \neq \emptyset$$

Consequently, the condition (2) is satisfied, the split of lump with a size greater than 2 will let the condition (2) satisfied.

□

At the initialization of the 'chain of lumps' the initial interval I_{init} is confided to the first lump. At this moment the condition (2) is satisfied. The condition (2) still be satisfied if no split of lump with a size less than 3 occurs.

3.2.5 Optimization

In this Section we study how the nodes can optimize the 'chain of lumps'. There is two different kinds of optimization, the first one that is presented in the Section 3.2.5.1, consists in making the nodes sharing the same number of keys. The second kind of optimization is to reduce the number of the sub-intervals of the 'chain of lumps'. The reduction of the number of sub-intervals is explained in the Section 3.2.5.2.

3.2.5.1 Load balancing

The load balancing consists in fairly distributing the <key, value> pairs over the nodes of the 'chain of lumps'. In order to do this, the sub-intervals must be fairly shared out over the set of the lumps. There exists three strategies in order to balance the load. In the first strategy the length of each sub-interval should be equal. In the second strategy, the cumulative length of the handled sub-intervals should be equal for each lump. In the third strategy, the cumulative length of the handled sub-intervals must be proportional to their size, i.e. the cumulative length of the sub-intervals handled by a lump multiplied by its size must be the same for each lump. The choice of the strategy must be done before that ROSA is initiated.

To ensure that the load balancing of the chosen strategy, each node looks down in its list of lumps for a pair of consecutive lumps. We denote this pair $<l_I, l_{I+}>$, where l_{I+} is the successor of l_I. Afterwards, the nodes checks that these two lumps are in accordance with the strategy. If it is not the case, the node have to find a pair of sub-intervals $<I$, $I+>$, I handled by l_I and $I+$ handled by l_{I+}. The pair $<I$, $I+>$, must satisfy the following conditions:

- $I+$ must be a successor of I
- The length of $I+$ must be greater than the length of I

If the node is unable to find such a pair, the load balancing process stops. Else if the node can find such pairs, the node selects a pair among all the possible pairs. The selection criteria depend of the chosen strategy. Once I and $I+$ are found, the node modifies the length of the sub-intervals I and $I+$ in accord with the chosen strategy. The modification of the sub-intervals length consists in setting the upper bound of the representation of I and the lower bound of the representation of $I+$ to a new value that depends on the chosen strategy. The node also has to reattributes the <key, value> pairs of the list of DHT entries of I and $I+$ in order to reflect these changes. Afterwards, the node sends to the nodes that compose l_I and l_{I+} the modifications done on I and $I+$.

The load balancing is done by the function loadBalance. This function has to be added to the added to the ROSA main loop. The pseudo code of the function loadBalance can be found in the Figure 3.15. The function reattributeDHTEntry takes two contiguous sub-intervals in argument and modifies their DHT entries lists in order to be compatible with the bounds of the sub-intervals. The function inAccordance checks if the two lumps given in arguments are in accord with the chosen load balancing strategy. The function findPairToAccord returns a pair of contiguous sub-intervals. This pair is the best pair among the possible ones according to the load balancing strategy. The function getNew-Bound returns the new value for the upper bound and the lower bound of respectively the

```
loadBalance
 1: for all lump_I ∈ lump_list do
 2:     for all lump_I+ ∈ lump_list do
 3:         if succeed(lump_I, lump_I+) then
 4:             if not inAccordance(lump_I, lump_I+) then
 5:                 return loadBalance(lump_I, lump_I+) ;

loadBalance(lump_I, lump_I+) (Input:lump, lump)
 6: toAccord ← findPairToAccord(lump_I, lump_I+) ;
 7: if toAccord = null then
 8:     return ;
 9: accordLength(toAccord, lump_I, lump_I+) ;
10: send UpdateSubInt(toAccord[0]) to lump_I ;
11: send UpdateSubInt(toAccord[1]) to lump_I+ ;

accordLength(toAccord, lump_I, lump_I+)
(Input:pair<subinterval, subinterval>, lump, lump )
12: newBound ← getNewBound(toAccord) ;
13: pair[0].upBound ← newBound ;
14: pair[1].lowBound ← newBound ;
15: reattributeDHTEntry(pair[0], pair[1]) ;

Messages handling
    ...
16: upon receive UpdateSubInt(subinterval_1) do
17:     for all lump ∈ lump_list do
18:         for all subinterval_2 ∈ lump.subInt_list do
19:             lump.subInt_list ← lump.subInt_list - subinterval_2 ;
20:             lump.subInt_list ← lump.subInt_list ∪ subinterval_1 ;
    ...
```

Figure 3.15: The load balancing function

first and the second element of the pair of sub-intervals given in argument. These three functions depend on the chosen strategy. Below, we explain the selection criteria and we detail the functions mentioned above for each strategy.

Lengths of sub-intervals equal

A lump l_I and its successor l_{I+} are not in accordance with this strategy if there exists a pair of sub-intervals $<I, I+>$, I handled by l_I and $I+$ handled by l_{I+} such the length of I is different from the length of $I+$ and greater that a given threshold. The threshold must be chosen in order to have a good load balancing without performing too many load balancing operations. The smaller the threshold is, the better the load balancing is but

the greater the number of load balancing operations needed is. If many pairs are found by the node that performs the load balancing, the node has to select one. The selected pair is the pair of sub-intervals with the difference of length that is the greater.

Once the pair $<I, I+>$ is selected, the value for the upper bound and the lower bound of respectively I and $I+$ is computed as:

$$\left\lfloor \frac{lowerBound(I) + upperBound(I+)}{2} \right\rfloor$$

where lowerBound(I) and upperBound($I+$) are respectively the lower bound of I and the upper bound of $I+$. The pseudo code of the functions inAccordance, findPairToAccord and getNewBound of the first strategy are shown in the Figure 3.16.

```
inAccordance(lump_I, lump_I+) (Input:lump, lump)
 1: for all pair ∈ getSuccSubIntPair(lump_I, lump_I+) do
 2:     if |pair[0].getLength() - pair[1].getLength()| ≥ threshold then
 3:         return false ;
 4: return true ;

findPairToAccord(lump_I, lump_I+)
(Input:lump, lump; Output:pair<subinterval, subinterval>)
 5: diffLength ← 0 ;
 6: toAccord ← null ;
 7: for all pair ∈ getSuccSubIntPair(lump_I, lump_I+) do
 8:     diffCurrLength ← |pair[0].getLength() - pair[1].getLength()| ;
 9:     if diffCurrLength > diffLength then
10:         diffLength ← diffCurrLength ;
11:         toAccord ← pair ;
12: return toAccord ;

getNewBound(toAccord, lump_I, lump_I+)
(Input:pair<subinterval, subinterval>, lump, lump; Output:integer)
13: return ⌊(toAccord[1].upBound + toAccord[0].lowBound)/2⌋ ;
```

Figure 3.16: Functions for the first load balancing strategy

Cumulative lengths of the lumps equal

A lump l_I and its successor l_{I+} are not in accordance with this strategy if the difference between the cumulative length of the sub-intervals handled by l_I and the cumulative length of the sub-intervals handled by l_{I+} is greater than a given threshold. As for the first strategy, the threshold must be chosen in order to have a good load balancing without performing to many load balancing operations.

The node that finds two lumps l_I and l_{I+} that are not in accordance has to find a pair of sub-intervals $<I, I+>$, I handled by l_I and $I+$ handled by l_{I+} such that $I+$ is

the successor of I and such that the length of $I+$ is greater than the length of I if the cumulative length of the sub-intervals handled by l_I is less the cumulative length of the sub-intervals handled by l_{I+} and such that the length of $I+$ is less than the length of I else. If the node is unable to find such a pair, the load balancing process stops. Else if the node can find such pairs, the node selects a pair among all the possible pairs. The selection criterion is that the difference between the length of two sub-intervals I and $I+$ must be the closer to the difference between the cumulative length of l_I and l_{I+}

```
inAccordance(lump_I, lump_I+) (Input:lump, lump)
 1: if |lump_I.getCSL() - lump_I+.getCSL()| ≥ threshold then
 2:     return false ;
 3: return true ;

findPairToAccord(lump_I, lump_I+)
(Input:lump, lump; Output:pair<subinterval, subinterval>)
 4: diffLength ← infinity ;
 5: toAccord ← null ;
 6: cl_I ← lump_I.getCSL() ;
 7: cl_I+ ← lump_I+.getCSL() ;
 8: diffCL ← |cl_I - cl_I+| ;
 9: for all pair ∈ getSuccSubIntPair(lump_I, lump_I+) do
10:     length_0 ← pair[0].getLength() ;
11:     length_1 ← pair[1].getLength() ;
12:     if length_0 < length_1 and cl_I < cl_I+ or
            length_0 > length_1 and cl_I > cl_I+ then
13:         diffCurrLength ← |length_0 - length_1| ;
14:         if |diffCurrLength - diffCL| < diffLength then
15:             diffLength ← diffCurrLength - diffCL;
16:             toAccord ← pair ;
17: return toAccord ;
```

Figure 3.17: Functions for the second load balancing strategy

Once the pair $<I, I+>$ is selected, the value for the upper bound and the lower bound of respectively I and $I+$ is computed as:

$$\left\lfloor \frac{lowerBound(I) + upperBound(I+)}{2} \right\rfloor$$

Where lowerBound(I) and upperBound($I+$) are respectively the lower bound of I and the upper bound of $I+$. One can remark that this is the same value as for the first load balancing strategy, consequently the pseudo code of the function getNewBound is not detailed. The pseudo code of the functions inAccordance, findPairToAccord of the second

strategy are shown in the Figure 3.17. The function getCSL returns the sum of the lengths of the sub-intervals handled by the lump caller and its pseudo code is in the Appendix.

Cumulative lengths of the lumps proportional to their size

To determine if a lump l_I and its successor l_{I+} are not in accordance with this strategy, a node has to compute the difference between the cumulative length ratio of l_I and the cumulative length ratio of l_{I+}. The cumulative length ratio of a lump is the ratio of the sum of the lengths of the sub-intervals handled by the lump over its size. If the difference is computed is greater than a given threshold, the node start the load balancing process. As for the two precedent strategies, the load balancing process consists in first finding pairs of sub-intervals $<I$, $I+>$, I handled by l_I and $I+$ handled by l_{I+} such that $I+$ is the successor of I and such that the length of $I+$ is greater than the length of I if the cumulative length ratio of l_I is less the cumulative ratio of l_{I+} and such that the length of $I+$ is less than the length of I else. If the node is unable to find such a pair, the load balancing process stops. Else if the node can find such pairs, the node selects a pair among all the possible pairs. The selection criterion is that the difference between the length of two sub-intervals I and $I+$ must be the greater possible. Once the pair $<I$, $I+>$ is selected, the value for the upper bound and the lower bound of respectively I and $I+$ is computed.

Let us consider the case where, the cumulative length ratio of l_I is greater than the cumulative length ratio of l_{I+}. Let cr_I and cr_{I+} respectively be the cumulative length ratio of l_I and the cumulative length ratio of l_{I+}. Let $length_I$ be the length of I and $length_{I+}$ be the length of $I+$. Since $cr_I > cr_{I+}$ the pair $<I$, $I+>$ was chosen such that $length_I > length_{I+}$. Let $upperBound_I$ be the upper bound of the sub-interval I, $size_I$ be the size of l_I and $size_{I+}$ be the size of l_{I+}.
The value of the new bound is denoted *newbound*.

$$newbound = upperBound_I + v_{add}$$

where v_{add} is the greatest value that satisfies:

$$\begin{aligned} v_{add} &< 1/3 length_{I+} && (l1) \\ v_{add} &\leq \frac{(size_I \times size_{I+})(size_{I+} \times cr_I - size_I \times cr_{I+})}{2} && (l2) \end{aligned}$$

The condition (l1) ensures that at most a third of the length of the sub-interval $I+$ can be given to the sub-interval I. If the cumulative length of the lump l_I still be less than the one the lump l_{I+}, the process will be repeated. The condition (l2) ensures that the length of the sub-interval $I+$ that is given to the sub-interval I does not make the cumulative length of the lump l_{I+} less than the one the lump l_I.

In the case where, the cumulative length ratio of l_{I+} is greater than the cumulative length ratio of l_I, v_{add} is computed in the same way but in inverting respectively $size_I$, cr_I and $length_I$ with $size_{I+}$, cr_{I+} and $length_{I+}$. The value of the new bound *newbound* is equal to $upperBound_{I+} + v_{add}$.

The pseudo code of the function inAccordance, findPairToAccorf and getNewBound are shown in the Figure 3.18.

inAccordance(lump$_I$, lump$_{I+}$) (**Input:***lump*, *lump*)

```
1: cr_I ← lump_I.getSCL() / lump_I.nodeId_list.size() ;
2: cr_I+ ← lump_I+.getSCL() / lump_I+.nodeId_list.size() ;
3: if |cr_I - cr_I+| ≥ threshold then return false ;
4: return true ;
```

findPairToAccord(lump$_I$, lump$_{I+}$)
(**Input:***lump*, *lump*; **Output:***pair<subinterval, subinterval>*)

```
5: diffLength ← 0 ;
6: toAccord ← null ;
7: cl_I ← lump_I.getCSL() ;
8: cl_I+ ← lump_I+.getCSL() ;
9: for all pair ∈ getSuccSubIntPair(lump_I, lump_I+) do
10:    length_0 ← pair[0].getLength() ;
11:    length_1 ← pair[1].getLength() ;
12:    if length_0 < length_1 and cl_I < cl_I+ or
           length_0 > length_1 and cl_I > cl_I+ then
13:       diffCurrLength ← |length_0 - length_1| ;
14:       if diffCurrLength > diffLength then
15:          diffLength ← diffCurrLength ;
16:          toAccord ← pair ;
17: return toAccord ;
```

getNewBound(toAccord, lump$_I$, lump$_{I+}$)
(**Input:***pair<subinterval, subinterval>*, *lump*, *lump*; **Output:***integer*)

```
18: cr_I ← lump_I.getSCL() / lump_I.nodeId_list.size() ;
19: cr_I+ ← lump_I+.getSCL() / lump_I+.nodeId_list.size() ;
20: if cr_I < cr_I+ then
21:    return getNewBound(pair<toAccord[1], toAccord[0]>,
                          lump_I+, lump_I) ;
22: limit_1 ← 1/3 toAccord[1].getLength();
23: limit_2 ← (lump_I.size() × lump_I+.size)
           (lump_I.size() × cr_I - lump_I+.size × cr_I+) / 2 ;
24: v_add ← 0 ;
25: while v_add < limit_1 and v_add ≤ limit_2 do v_add ← v_add + 1;
26: return lump_I.infBound + v_add ;
```

Figure 3.18: Functions for the third load balancing strategy

3.2.5.2 Reducing the number of sub-intervals

The number of hops needed to perform a lookup over a DHT determines the efficiency of this DHT. The number of hops needed to perform a lookup over the 'chain of lumps'

depends on the number of sub-intervals. Therefore, reducing the number of sub-intervals that compose the 'chain of lumps' is a good way to improve its efficiency.

Since sub-intervals cannot be suppressed the only way to reduce the number of sub-intervals is to merge some of them. We have already seen in the section dealing with the absorptions of a lump (Section 3.2.2.2) how two sub-intervals are merged. In order to increase the number of merges that occur a lump l with more than one sub-interval may confide one of its sub-interval I to another lump l' that satisfies the following conditions:

- l' can handle the sub-interval I ;
- l' handles another sub-interval I' such I and I' are contiguous (and consequently be merged) ;

Once I is confided to l' the sub-intervals I and I' are merged.

reduceSubIntNumber

```
1:  for all lump_I in lump_list do
2:      for all lump_I' in lump_list s.t. lump_I ≠lump_I' do
3:          if lump_I.subInt_list.size() > 1 then
4:              toConfide ← getSubIntToConfide(lump_I, lump_I') ;
5:              if toConfide ≠ null then
6:                  confide(lump_I, lump_I', toConfide) ;
7:                  send RemoveSubInt(lump_I.id, toConfide) to lump_I;
8:                  send AddSubInt(lump_I'.id, toConfide) to lump_I';
```

Messages handling

```
    ...
9:  upon receive RemoveSubInt(id, subint) do
10:     for all lump in lump_list do
11:         if lump.id = id then
12:             lump.subInt_list ← lump.subInt_list - subint ;
13: upon receive AddSubInt(id, subint_1) do
14:     if lump.id = id then
15:         for all subint_2 in lump.subInt_list do
16:             if contiguous(subint_1, subint_2) then
17:                 subint_1 ← merge(subint_1, subint_2) ;
18:                 lump.subInt_list ← lump.subInt_list - subint_2 ;
19:         subint_1.owner ← lump.id ;
20:         lump.subInt_list ← lump.subInt_list ∪ subint_1 ;
    ...
```

Figure 3.19: Reduction of the number of sub-intervals

A node that wants to reduce the number of sub-intervals looks down its list of lumps in order to find two lumps l and l' such that l handles more than one sub-interval. If the node finds such lumps, it look for two sub-intervals I and I', respectively handled by l and l' such that I and I' are contiguous and I can be handled by l'. If such I and I' are found, the node modifies the representations of l and l' in order to reflect that the sub-interval I is not handled anymore by l but by l'. The representation of l' is also modified in order to reflect that I and I' are merged. The node sends the new representation of l and l' to respectively the nodes that compose these lumps.

This reduction of the sub-intervals of the 'chain of lumps' is done with the help of the function reduceSubIntNumber. This function has to be added to the main loop of ROSA. The pseudo code of the function reduceSubIntNumber is in the Figure 3.19. The function getSubIntToConfide returns a sub-interval of the first lump given in argument. This subinterval is chosen in order to satisfy the conditions described above. If such a sub-interval cannot be found the function returns null.

Chapter 4

A reliable storage over ROSA

4.1 Introduction to distributed file storage systems

A distributed file storage system is a system that allows the participating nodes to store, retrieve and delete files. In the Section 4.1.1 we introduce all the notions necessary to the understanding of the distributed file storage systems. Then in the Section4.1.2 we detail the main properties of such systems. And since the file storage system deployed over ROSA will use the 'chain of lumps' we present, in the Section 4.1.3 some example of distributed file storage systems that use some DHTs as backbones.

4.1.1 Definition

Distributed storage systems became the preferential method of data storage for the distributed applications. Some big companies, like Google, Amazon and Yahoo!, use such systems for their web applications. The distributed storage systems are preferred over the traditional storage system because of their fundamental properties such the scalability, the availability and their failure tolerance.

A distributed storage system is composed of many memory units. The files stored over such a system are shared over these memory units. The advantages of a distributed storage system is that it does not have a single point of failure and the large number of memory units endows the storage system to an unlimited storage capacity.

On these systems, depending of the chosen solution, the file can be stored in one block of a single memory unit or can be split and distributed over many memory units. Since there is many memory units, a node that wants to retrieve a file must discover on which units the file or its different parts are stored. This implies that a distributed storage system must benefit from a file lookup mechanism. In order to be more reliable, some distributed storage system, stores many copies of the same file. Such storage systems have to be endowed with a mechanism that ensures the consistency of the system.

Some distributed storage systems use a DHT as backbone platform. CFS (Dabek et al. [2001]) deployed over Chord (Morris et al. [2001]), PAST (Druschel and Rowstron [2001]) deployed over Pastry (Rowstron and Druschel [2001]) and OceanStore (Kubiatowicz et al. [2000]) deployed over Tapestry (Zhao et al. [2004]) are the first storage systems relying on DHT based backbones. Using a DHT as a backbone is judicious because the concerned storage systems benefits of all the interesting properties of the supporting DHTs, such

the scalability and the load balancing. Moreover on the DHT based storage systems, the lookup mechanism is confided to the DHT.

4.1.2 Properties

In order to be effective a distributed storage system must have a fast data lookup. On these systems the data are distributed in diverse memory units and an user must be able to store and retrieve a file in a efficient way in term of bandwidth usage. The systems based on a DHT usually allow the user to store and retrieve usage with the same cost as the lookup mechanism of the DHT. The scalability of the distributed storage system is its capacity to store a large amount of file without the efficiency of the storage and retrieve mechanisms are deteriorated as well as its capacity to be composed by a large amount of memory units. The scalability is the other deciding factor in the performance of the system.

The persistence of the data stored over a reliable system is another important property. An efficient distributed storage system should be able to ensure a persistent access to data. Some mechanisms must ensure that in the presence of memory units failures, the data stored in the system are preserved and still be accessible to any user.

Another interesting property is the possibility for an user to modify the content of a file stored over the system. The storage systems that do not allow users to modify or update a stored file are told read-only, the storage systems that allow users to perform modification on stored files are read/write systems. Achieving the consistency of read/write storage systems that store many copies of the same file is a serious issue. Indeed, when an user performs a modification on a file stored on such system the modification has to be reverberated on all the copies.

4.1.3 Example

Since the reliable storage system deployed over ROSA is based on the 'chain of lumps' and in order to have a better understanding of how such storage systems works, we show in this section two examples of DHT based distributed storage systems. The first one, CFS (Dabek et al. [2001]) is based on Chord (Morris et al. [2001]). The second one is Past (Druschel and Rowstron [2001]). Past uses Pastry (Rowstron and Druschel [2001]) as backbone platform.

4.1.3.1 PAST

PAST is a large-scale storage utility developed by Microsoft Research in 2001. Past uses the Pastry lookup and routing mechanisms. Each node in PAST acts as client access point and memory unit and is identified by an unique 128-bit identifier. Each user of PAST has a storage quota. Each time that an user stores a file, this quota is decremented from the file size. An user is not allowed to store a file over PAST if its storage quota is less than the file size. PAST allows users to perform three operations, insertion, lookup and reclaim.

The insertion operation stores a file on PAST. The user must compute the 160-bit identifier of the file. This identifier is obtained by hashing the name of the file, the 128-bit identifier of the node inserting the file, the number of replica wanted and the file itself. The file is then routed by Pastry to the nodes whose identifiers are numerically closest to

the 128 most significant bits of the file identifier. This is an invariant of PAST, and must be maintained over the lifetime of a file, despite the arrival, failure and recovery of PAST nodes.

The file size is subtracted from the storage quota of the user for each stored replica. The files are inserted in a immutable way. It means that the same file cannot be inserted multiple time. This operation also ensures that the set of nodes storing the file is fairly chosen among the nodes of the network and therefore that the number of files assigned to each node is roughly balanced.

Given a file identifier, the lookup operation retrieves a copy of the file corresponding to the identifier if it exists in PAST. Consequently, a node that wants to retrieve a file needs the file identifier. The lookup request is confided to Pastry and is routed to the closest live node among the nodes that store a copy of the file to the PAST node issuing the lookup (in terms of the Pastry proximity metric).

Since an user is only allowed to store a limited amount of data over PAST, an operation allows users to reclaim the storage occupied by the copies of a file. An user that wants to perform such operation sends a request containing the identifier of the file. This reclaim request is routed using Pastry to the nodes storing the copies of the file.

4.1.3.2 CFS

CFS stands for Cooperative File System. It was developed at MIT in 2001. CFS allows any user to store and update their own file over the system, and provides read-only access to other users. CFS performs load balancing, is failure tolerant and has a restriction mechanism that controls the amount of data that an user is allowed to store.

CFS is not directly based on Chord but is based on DHash (Brunskill [2001]). DHash is a distributed block storage system built over Chord. DHash provides an interface that allows user to put and get blocks of data over Chord. The DHash lookup requests are confided to Chord.

To insert a data block over DHash under a given key, the key has to be hashed to produce a Chord identifier k. The block has to be stored by the node of Chord that is the successor of k. The lookup request is performed analogously, to retrieve a block stored under a given key, the key has to be hashed into the Chord identifier k. The lookup request is routed by Chord to the successor of k. The transfers of the data blocks between the nodes are accomplished by an RPC distinct of Chord.

In order to ensure a better reliability DHash stores each block of data associated with a given key not only on the direct successor of the key, but also on the next n successors. Where n is a parameter that is decided by the node that wants to store the data block. DHash also maintains the number of replicas despite the failures of nodes or the fact that some nodes join or leave the network.

The files stored in CFS are split in data blocks and the blocks are managed by DHash. A third layer, called FS, interprets the data blocks as files and provides a file system interface to the users and to the applications.

The three layers of CFS are represented in the Figure 4.1. On the left part of this figure one can see a node publishing a file over CFS. The file is confided by the node to the FS layer. The FS layer splits the file in data blocks and confides them to the DHash layer. The DHash confides each of these data blocks to one or many nodes of the Chord

layer. In the right part of the figure the inverse process is represented. A user asks the FS layer for a file. The FS layer asks for the data blocks to the DHash layer. The DHash layer retrieves the data blocks and gives them to the FS layer that returns the file to the user that requested it.

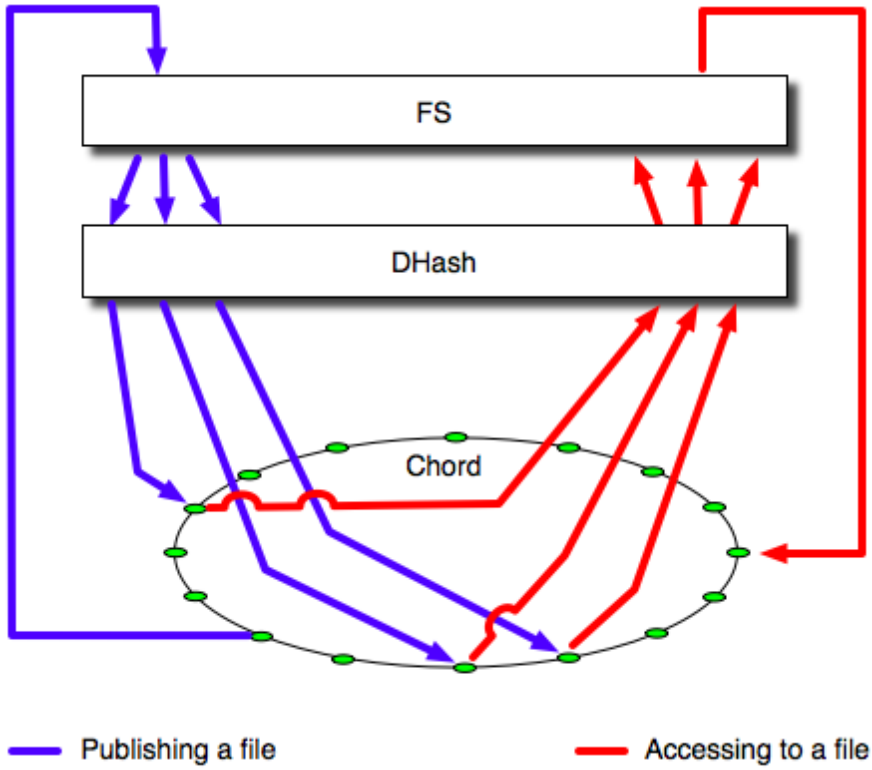


Figure 4.1: CFS

4.2 The ROSA reliable storage system

The ROSA protocol endowed with the 'chain of lumps' offers a resilient network topology and a resilient distributed HashTable. In this section we explain how a reliable file storage system can be built over the 'chain of lumps'. This storage system already benefits of the resiliency of the 'chain of lumps' and in order to ensure a better failure tolerance each file stored over ROSA is stored in many replicas. Each replica is stored by a node. The number of replicas is a parameter that has to be decided when the file is stored. In this way important files could be stored in many replicas while negligible ones in only few replicas. In order to allow a node to discover where the replicas of a given file are located, for each file an index is also stored over ROSA. The storage system of ROSA ensures a better reliability than CFS because the system also handles the transfer of the copies of the files, therefore the file transfer is not confided to an external RPC and benefits of the resiliency of ROSA. In Section 4.2.1 we explain what are the files indexes. In the Sections 4.2.2 and 4.2.3 we detail how a node stores and retrieves a file. In the Sections 4.2.4 and 4.2.5

we see how some nodes can update, modify or delete a file. In Section 4.2.6 we explain how the whole storage system is maintained despite the presence of failures.

4.2.1 Files indexes

The files indexes allow nodes to retrieve the location of the replicas of the files. The index of a file contains, the file identifier, some optional data about the file, a flag, the lists of the locations of the replicas and the list of the identifiers of the nodes that owns this file. A file index is described in Figure 4.2. The file identifier identify the file over the 'chain

File ID	Version	Flag
ID_{file}	Version nb	flag
File data		
File name, description, etc.		
Replicas		
$ID_{node1} : [a_1, b_1]; [a_2, b_2]; ...$		
$ID_{node2} : [a_3, b_3]; ...$		
...		
Owners ID		
ID_{owner1}		
ID_{owner2}		
...		

Figure 4.2: A file index

of lumps'. This identifier is an integer and must belong to the key space I_{init}. The data is optional, it could be a file name, a description of the file or some comments from the owners of the file. The flag contained in the index describes the state of the file. A file can be currently available, updated or deleted. The version of the file is an integer. Each time a file is updated the version is incremented. A location of a replica is constituted by the identifier and the bounds of the sub-intervals handled by the lumps whose the nodes that store the replicas belong. The list of the owners is the list of the identifiers of the nodes that are allowed to access, update and delete the file. If the owners list is void, it means that every participating node is an owner of the file. The index of a file is stored by the nodes that compose the lump that handles the file identifier. This lump will be called the lump index of this file.

4.2.2 Storing a file

A node that wants to store a file upon ROSA, computes the identifier of the file. This identifier is obtained by using a hash function over the name. Afterwards, the node determines the number of replicas needed and which nodes are allowed to access the file. To finish the node has to build a message *StrFile*. This message, as described in Figure 4.3, contains the sending node identifier and location, the file identifier, an optional description of the file, the number of replicas wanted, the list of the owners and the file itself.

Once the message is built, the node sends it over the DHT to the lump index of this file. The lump index of the file is the lump that handles the sub-interval that contains

StrFile	
Node ID	Node location
File ID	File description
Rep. nb	Owners
File	

Figure 4.3: A message *StrFile*

the file identifier. The node sends the message using the process described in Chapter 3, Section 3.2.3.3 with the message *StrFile* as a data packet.

When a node of the lump index receives a message *StrFile*, if a file with the same identifier is already stored, the node replies with a message *BadId*. This message contains the non suitable file identifier. Else, it builds and sends a message *StrIdx* to the nodes of the lump index. These nodes will store the index of this file. The message *StrIdx* contains the index of this file, built with the content of the message *StrFile* and where the Flag is set to 'available' and the Version to 1.

The node that receives the message *StrFile* also randomly chooses N elements of I_{init}, where N is the number of replicas wanted. We denote these elements $k_i, i \in [1, N]$. Afterwards, the node builds a message *ScttrRep* that contains the file identifier and the file itself. To conclude, the node sends a copy of the message *ScttrRep* to each one of the lumps that handles one of the k_i. These messages are also sent as data packets using the process described in the Chapter 3, Section 3.2.3.3.

The process ensures that the messages *ScttrRep* reach ordinary nodes of one of the lumps that handles one of the k_i. When one of these nodes receives such a message it randomly choose a node of the lump that handles the k_i and sends a message *StrRep* to this node. This message informs the receiving node that it has to store a replica of the file. A message *StrRep* contains the file identifier and a copy of the file.

When a node willing to store a file receives a message *BadId* it modifies the file name, computes a new file identifier and starts again the whole process.

The pseudo code of the store process is described in the Figure 4.4. A file possesses a name, a description and a size. The function computeId computes the identifier of a file. The functions buildStrFile, buildStrIdx, buildScttrRep, buildBadId and buildStrRep respectively build the messages *StrFile*, *buildStrIdx*, *buildScttrRep*, *badId* and *buildStrRep*. The function sendDHT is a function that, given a key in I_{init}, sends the data packet given in argument to a node of the lump handling the key. The function getKeyIn returns a key corresponding to the node location given in argument. The functions storeIndex and storeReplica respectively make the node stores the file index and replica given in argument.

4.2.3 Retrieving a file

A node that wants to retrieve a file must be aware of the file identifier. In order to retrieve a file, the node has to send a message *RtrvFile* to the lump index of the file using the process described in Chapter 3, Section 3.2.3.3. A message *RtrvFile* contains the identifier of the file to retrieve, the identifier and the location of the node that wants to retrieve the

```
storeFile(toStore, rep_nb) (Input:file, integer)
 1: file_id ← computeId(file) ;
 2: strFile_mess ← buildStrFile(node, file, rep_nb, file_id) ;
 3: sendDHT(strFile_mess) to file_id ;
Messages handling
   ...
 4: upon receiveDHT StrFile do
 5:     if alreadyStored(StrFile.content.file_id) then
 6:         key ← getKeyIn(StrFile.content.snd_loc) ;
 7:         return sendDHT(buildBadID()) to key ;
 8:     scttrRep_mess ← buildScttrRep(StrFile.content);
 9:     for all i ∈ [1, StrFile.content.rep_nb] do
10:         sendDHT(scttrRep_mess) to rand(0, 2^128 − 1) ;
11:     for all lump ∈ lump_list do
12:         if lump.handle(StrFile.content.file_id)) then
13:             return send(buildStrIdx(StrFile.content)) to lump ;
14: upon receiveDHT ScttrRep do
15:     for all lump ∈ lump_list do
16:         if lump.handle(ScttrRep.dst_key)) then
17:             return send(buildStrRep(ScttrRep.content)) to lump ;
18: upon receiveDHT StrIdx do
19:     storeIndex(StrIdx.content) ;
20: upon receiveDHT StrRep do
21:     storeReplica(StrRep.content) ;
   ...
```

Figure 4.4: The store process

file. A message *RtrvFile* is described in Figure 4.5.

When a node of the lump index receives such a message, it checks first that the file exists. The node performs this check by looking in its stored indexes if the appropriate index can be found. If the node cannot find such an index it replies with a message *FileNotFnd*. Else the node checks in the corresponding stored index that the sender identifier is in the owners list field or that the owners list is void. The node replies with a message *NotAllwd* if the sender is not an owner of the file. Else, the node replies with a message *Wait* if the flag in the index is not set to 'available' and with a message *RepLoc* in the other cases. A message *RepLoc* must contain the locations of the replicas and the current file version.

Once the node that wants to retrieve the file is knows the locations of the replicas, it selects the replica that is the closer (in the meaning of the 'chain of lumps') to itself and sends a message *RtrvRep*. This message contains the identifier of the file to retrieve. A

RtrvFile	
Node ID	Node location
File ID	

Figure 4.5: A message *RtrvFile*

node that receives such a message has to reply with a message *HereItIs* that contains the replica of the wanted file. If after sending a message *RtrvRep* and a chosen amount of time a node do not receive the message *HereItIs*, it selects another replica location and sends again a message *RtrvRep*.

The role of the messages *Wait*, that are sent when the flag of the file is not set to available, is to inform the node that wants to retrieve the file that it has to wait an indefinite amount of time before receiving the message *RepLoc*. We will see in the Section 4.2.4 and 4.2.5 in which cases it happens.

The pseudo code of the retrieve process is shown in the Figure 4.6. The function getIndex returns the index corresponding to the file identifier given in argument stored by the node. The function getCloserLocation takes a list of locations in argument and returns the one that is the closer in the 'chain of lumps' to the caller. The function isOwner return true if the identifier given in argument is in the owners list of the coressponding index or if this list is void. this function returns false in all the other cases. The functions buildRtrvFile, buildFileNotFnd, buildNotAllwd, buildWait, buildRepLoc, buildRtrvRep and buildHereItIs respectively build the messages *RtrvFile*, *FileNotFnd*, *NotAllwd*, *Wait*, *RepLoc*, *RtrvRep* and *HereItIs*.

4.2.4 Updating and modifying a file

Updating or modifying a file is basically the same process, the only difference resides in the fact that in the case of a complete update the whole file will be sent again. In the other case only the differences between the old version of the file and the new one have to be sent.

Since all the nodes that owned the same file are able to update or modify it, the first step for a node to update a file is to retrieve the last version of the file and the version number. This is done as described in the Section 4.2.3. Once in possession of a copy and of the version number of the file, the node performs the needed changes on the file then computes the differences between the retrieved file and the new one. These differences are obtained using the differential file comparison utility diff (Hunt and McIlroy [1976]). Afterwards, the node has to send a message *UpdtFile* to the lump index of the file. This message must contain the node sender identifier and location, the file identifier and the version number obtained when retrieving the file and the computed changes. This message is described in the Figure 4.7.

When a node of the lump index receives such a message, since the file may have been deleted, it checks first that the file still exist. The node performs this check by looking in its stored indexes if the appropriate index can be found. If the node cannot find such an index, it replies to the sender of the message *UpdtFile* with a message *FileNotFound*. If the file exist, the node checks that the update request is done by a node owner of the file. If the update request does not come from an owner the file, the node replies with a

```
retrieveFile(file_id) (Input:integer)
 1: rtrvFile_mess ← buildRtrvFile(node, file_id) ;
 2: sendDHT(rtrvFile_mess) to file_id ;
Messages handling
   ...
 3: upon receiveDHT RtrvFile do
 4:     key ← getKeyIn(RtrvFile.content.snd_loc) ;
 5:     file_index ← getIndex(RtrvFile.content.file_id) ;
 6:     if file_index = null then
 7:         return sendDHT(buildFileNotFnd()) to key ;
 8:     if not file_index.isOwner(RtrvFile.content.snd_id) then
 9:         return sendDHT(buildNotAllwd()) to key ;
10:     if file_index.flag ≠ available then
11:         return sendDHT(buildWait()) to key ;
12:     return sendDHT(buildRepLoc(file_index.replica_location)) to key ;
13: upon receiveDHT RepLoc do
14:     while RepLoc.content.size() > 0 and not received(HereItIs) do
15:         loc ← getCloserLocation(RepLoc.content) ;
16:         RepLoc.content ← RepLoc.content - loc ;
17:         key ← getKeyIn(loc) ;
18:         sendDHT(buildRtrvRep()) to key ;
19:         sleep(interval) ;
20: upon receiveDHT RtrvRep do
21:     sendDHT(buildHereItIs()) to getKey(RtrvRep.snd_loc) ;
22: upon receiveDHT HereItIs do
23:     extractFile(HereItIs.content) ;
   ...
```

Figure 4.6: The retrieve process

message *NotAllwd*. If the update request concerns a valid file and comes from an owner of the file, the node that received the message *UpdtFile* checks that the flag in the index is set to 'available' and that the version number contained in the message *UpdtFile* is the same that the version number of the index. If the flag is not equal to 'available' the node replies with a message *FileBusy* and if the file version numbers do not match the node replies with a message *FileMdfd*.

A message *FileBusy* indicates that another node is currently performing some operation on the file. The update request has to be repeated later. A message *FileMdfd* indicates that another node has modified the file, consequently the node that wants to update the file must retrieve the new version of the file and must start again the whole process.

If all the checks are successfully passed, the node that received the message *UpdtFile*

UpdtFile	
Node ID	Node location
File ID	File version number
Changes	

Figure 4.7: A message *UpdtFile*

sends a message *SetBusy* to all the nodes that compose the lump index of the concerned file. This message contains the file identifier. Each node receiving this message sets the flag in the corresponding index to 'busy'.

The node that received the message *UpdtFile* also sends a message *UpdtRep* to each node that stores a replica of the file. This message contains the file identifier and the changes to apply to the file that are contained in the message *UpdtRep*. The message is sent using the node identifier and the location of the replica contained in the file index. When a node of the lump handling the sub-interval corresponding to the location receives the message it forwards it to the node that stores the replica of the file (if this node is not storing the replica itself).

When a node that stores the replica receives the message *UpdtRep*, it updates the replica stored with the data contained in the message and sends a message *ACKUpdt* to all the nodes that compose the lump index. The messages *ACKUpdt* contains the file identifier. When a node of the lump index receives a message *ACKUpdt*, the node checks that it stores the index corresponding to the file identifier and if it is the case the node set the flag of the corresponding index to 'available'. If after a determined amount of time a message *ACKUpdt* still missing, the replica is declared lost and a node replica substitution process is engaged. This process is described in Section 4.2.6.3.

The pseudo code of the update process is shown in the Figure 4.8. In this figure, we assume that a copy and the version number of the file are already retrieved. The diff and patch utilities are used to compute and apply the difference between the old and the new version of the file. The function getReplica takes a file identifier in argument and returns the corresponding replica stored by the caller. If the caller does not store such a replica it returns null. The functions buildUpdtFile, buildFileBusy, buildFileMdfd, buildSetBusy and buildRepNotFnd respectively build the messages *UpdtFile*, *FileBusy*, *FileMdfd*, *SetBusy* and *RepNotFnd*.

4.2.5 Deleting a file

To delete a file stored over the 'chain of lumps', a node has to know the corresponding identifier. Once in possession of this identifier, the node sends a message *DltFile* to the lump index of the file. This message contains the identifier of the file. At the receipt of this message a node of the lump index checks if it store an index that corresponds to the identifier contained in the message. If such an index cannot be found the node replies with a message *FileNotFound*. Else it checks if the sender of the message *DltFile* is an owner of the file. If not, it replies with a message *NotAllwd*. If the delete request concerns a valid file and the requester is an owner of the file, the node that received the message *DltFile*

```
updateFile(file_id, old_file, new_file, vernum)
(Input:integer, file, file, integer)
 1: change ← diff(old_file, new_file) ;
 2: updtFile_mess ← buildUpdtFile(file_id, vernum, change) ;
 3: sendDHT(updtFile_mess) to file_id ;
Messages handling
    ...
 4: upon receiveDHT UpdtFile do
 5:     key ← getKeyIn(UpdtFile.content.snd_loc) ;
 6:     file_id ← UpdtFile.content.file_id ;
 7:     change ← UpdtFile.content.change ;
 8:     file_index ← getIndex(file_id) ;
 9:     if file_index = null then
10:         return sendDHT(buildFileNotFnd(file_id)) to key ;
11:     if not file_index.isOwner(UpdtFile.content.snd_id) then
12:         return sendDHT(buildNotAllwd()) to key ;
13:     if file_index.flag ≠ available then
14:         return sendDHT(buildFileBusy()) to key ;
15:     if UpdtFile.content.vernum ≠ file_index.vernum then
16:         return sendDHT(buildFileMdfd()) to key ;
17:     sendDHT(buildSetBusy(file_id)) to file_id ;
18:     updtRep_mess ← buildUpdtRep(file_id, change) ;
19:     for all loc ∈ file_index.replica_location do
            sendDHT(updtRep_mess) to getKeyIn(loc) ;
20: upon receiveDHT SetBusy do
21:     file_id ← SetBusy.content.file_id ;
22:     file_index ← getIndex(file_id) ;
23:     if file_index = null then
24:         return sendDHT(buildFileNotFnd(file_id)) to key ;
25:     file_index.flag ← busy ;
26: upon receiveDHT UpdtRep do
27:     file_id ← UpdtRep.content.file_id ;
28:     file_replica ← getReplica(file_id) ;
29:     if file_index = null then
30:         return sendDHT(buildRepNotFnd(file_id)) to key ;
31:     file_replica ← patch(file_replica, change) ;
    ...
```

Figure 4.8: The update process

waits until the flag of the index is set to 'available' and sends a message *DltIdx* to the other nodes that composes the lump index. These messages only contain the identifier of the file.

When a node of the lump index receives a message *DltIdx* it looks for the index corresponding to the identifier contained in the message in its stored index. If the node finds such an index it deletes it. The nodes that received the message *DltFile* also has to notice the nodes that store the replicas about the deletion of the file. In order to do that, with the help of the replica location contained in the index, the node sends messages *DltRep* to the nodes storing the replicas of the file. This message contains the identifier of the file. To finish this node has to delete his own copy of the index.

When a node of the lump handling the sub-interval corresponding to the location receives the message, it forwards it to the node that stores the replica of the file (if this node is not storing the replica itself). When such a node receives this kind of messages, it looks for the replica corresponding to the file identifier contained in the message. If the node finds such a replica among those that it stores, it deletes it.

The pseudo code of the delete process is shown in the Figure 4.9. The functions buildDltIdx and buildDltRep respectively build the messages *DltIdx* and *DltRep*.

```
updateFile(file_id) (Input:integer)
 1: sendDHT(buildDltFile(file_id)) to file_id ;
Messages handling
   ...
 2: upon receiveDHT DltFile do
 3:     key ← getKeyIn(DltFile.content.snd_loc) ;
 4:     file_id ← DltFile.content.file_id ;
 5:     file_index ← getIndex(file_id) ;
 6:     if file_index = null then
 7:         return sendDHT(buildFileNotFnd(file_id)) to key ;
 8:     if not file_index.isOwner(DltFile.content.snd_id) then
 9:         return sendDHT(buildNotAllwd()) to key ;
10:     while file_index.flag ≠ available do
11:         sleep(interval) ;
12:     sendDHT(buildDltIdx(file_id)) to file_id ;
13:     for all loc ∈ file_index.replica_location do
            sendDHT(buildDltRep(file_id)) to getKeyIn(loc) ;
14: upon receiveDHT DltIdx do
15:     removeIndex(DltFile.content.file_id) ;
16: upon receiveDHT DltRep do
17:     removeReplica(DltRep.content.file_id) ;
   ...
```

Figure 4.9: The delete process

4.2.6 Preservation of indexes and stored files

As shown in the previous Chapter 2, ROSA can be seen as an entanglement of lumps and these lumps can be split or absorbed by other lumps during the reconfiguration process or when failures occur. During these modifications of the set of lumps the sub-intervals could also be split or merged, and since the index of a file is stored by all the nodes of a lump these modifications affect the way of how the indexes of the files are stored.

The optimizations introduced in the Section 3.2.5.1 and in the Section 3.2.5.2 also modify the distribution of the sub-intervals handled by the lumps. Consequently it also affect the mapping of the file indexes over the lumps.

The failures can also affect the nodes that stores the replicas. If all the nodes that store the replicas of a file fail the file is lost. In order to avoid such losses, a mechanism checks at regular interval that any nodes storing a replica still able to work properly.

If the mechanism finds that a node storing a replica has failed, another mechanism is in charge of substituting the failing node by a node working correctly. This mechanism consist in modifying the corresponding file index in confiding a copy of the file to the new selected node.

In the Section 4.2.6.1 we detail how the indexes of the files are redistributed in order to reflect the changes on the set of lumps. In the Section 4.2.6.2 we see how the failures of the nodes that stores the replicas are detected. In the Section **??** we deal with the substitution of a failing node storing a replica by a new node that works correctly.

4.2.6.1 Preservation of indexes

When a lump is split into two new lumps, the sub-intervals that the lump handled are distributed to the two resulting lumps. Consequently, the indexes of the files that was stored by the split lump have to be distributed too. Some of these index will have to be stored by the first resulting lump and the other by the second one. Consequently, it may happen that a node stores the index of a file when it does not have to anymore. In this case this node has to discard this index.

When a lump is absorbed by another lump, the sub-intervals that the absorbed lump handled are confided to the absorbing lump. Consequently the indexes of the files that was stored by the absorbed lump must be stored by the absorbing one. It may happen that the nodes that compose the absorbing lump but not the absorbed one do not have the copy of some indexes. We have seen in the Section 3.2.2.2 that when a node detects an absorption, it notices the other concerned nodes of this absorption with a message *AbsorbLump*. In order to maintain the good mapping of the file indexes, the node that detects the absorption also has to send a message *AddIdx* to the nodes of the absorbing lump that did not compose the absorbed lump. This message contains the file indexes that was stored by the absorbed lump. When a node receives this message, it checks for each of the file indexes contained in the message if it has to store it and if it is the case it stores it.

When optimizations, such the load balancing or the reduction of the sub-intervals number, are used over the 'chain of lumps' the mapping of the files indexes may be inappropriate. As seen in the Section 3.2.5.1, during the load balancing a node may modify the lengths and the bounds of two contiguous sub-intervals, I and $I+$. A node is allowed to perform such a load balancing if the node belongs to the two lumps that handles the two

concerned sub-intervals, respectively l_I and l_{I+}. In order to maintain the good mapping of the file indexes, the node that performs the load balancing have to send messages *AddIdx* and *DltIdx* to the nodes that compose l_I and l_{I+}. The messages *AddIdx* that are sent to the nodes composing l_I and l_{I+} are respectively composed by the file indexes that have to be stored now by l_I and those that have to be stored by l_{I+}. The messages *DltIdx* sent to the nodes composing l_I and l_{I+} point out the indexes to delete. The reduction of the sub-intervals number shown in the Section 3.2.5.2 consists in a node confiding a sub-interval handled by a lump to another lump. Here again the mapping of the files indexes may be inappropriate. The node that confides the sub-interval to another lump must sens messages *DltIdx* to the nodes composing the lump that previously handled the sub-interval and messages *AddIdx* to the nodes of the lump that will have to handle the sub-interval. The messages *DltIdx* and *AddIdx* contains the file indexes that have to be reattributed. When a node receive a message *DltIdx* it deletes the corresponding file indexes contained in the message.

4.2.6.2 Preservation of stored files

The nodes storing the replicas of the files may have failures or leave the network, therefore the number of replicas requested for a file may not be maintained. These failures have to be detected in order to readjust the number of nodes that own a copy of the file.

A node that stores a replica of a file must periodically send a message *RepAlv* to the lump index of the corresponding file. This message contains the sender identifier, the sender location and the file identifier. This message possesses two roles. The first one is to notice the nodes that compose the lump index that the sender still works properly. The second role of this message is to keep up to date the list of the replicas location contained in the index of the file.

When a node of the lump index receives this message, it checks that it stores the index that corresponds to the file identifier contained in the message. If it cannot finds such an index it discards the message. If this condition is respected it replies to the sender with a message *ACKRepAlv*. Afterwards, the node that has received the message *RepAlv* forwards it to the other nodes of the lump index. These nodes do not have to send *ACKRepAlv*.

If, after a determined amount of time, a node of the lump index of a file has not received a message *RepAlv* from one of the node storing a replica, it considers that the node storing the replica is failing and a node replica substitution process is engaged. This process is described in the Section 4.2.6.3.

The pseudo code of this process is shown in the Figure 4.10. The function keepStorage-UpToDate is in charge of sending the messages *RepAlv* to the corresponding lump index for each replicas that the caller stores. This function is also in charge of checking for each stored file index that all the nodes storing a replica of the file still work correctly. The function notAlive returns true if the node given in argument has recently sent a message *RepAlv* and false else. The function substitute is detailed in the Section 4.2.6.3. Its role is to start the substitution process. The function updateIndex takes a index and a list of locations in argument, it updates the index with the locations. The functions buildRepAlv and buildACKRepAlv respectively build the messages *RepAlv* and *ACKRepAlv*.

It may also happen that during the deletion of a file (see Section 4.2.5), a node storing a replica did not receive the message *DltRep*. In this case, it may happen that a node

```
keepStorageUpToDate
1:  for all file ∈ file_stored do
2:      sendDHT(buildRepAlv()) to file.file_id ;
3:  for all index ∈ index_stored do
4:      file_id ← index.file_id ;
5:      for all replica ∈ index.replica do
6:          if notAlive(replica) then substitute(file_id) ;

Messages handling
    ...
7:  upon receiveDHT RepAlv do
8:      snd_id ← RepAlv.snd_id ;
9:      tgt_id ← RepAlv.key ;
10:     file_index ← getIndex(RepAlv.content.file_id) ;
11:     if file_index = null then return ;
12:     for all lump ∈ lump_list do
13:         if lump.handle(tgt_id) then
14:             loc ← RepAlv.content.loc ;
15:             if not lump.nodeId_list.contain(snd_id) then
16:                 sendDHT(buildACKRepAlv()) to getKeyIn(loc) ;
17:                 sendDHT(RepAlv) to tgt_id ;
18:             return updateIndex(file_index, loc) ;
    ...
```

Figure 4.10: Preservation of the stored files

stores a copy of a file whereas it does not have to do anymore. To prevent that waste files encumber the network, the nodes storing the replicas, which have not received a message *ACKRepAlv* since a determined amount of time, have to delete their replicas.

4.2.6.3 The node replica substitution process

When a node that stores the replica of a file does not periodically send messages *RepAlv* or does not reply to a message *ACKUpdt*, the node is considered as failing. The detection of such a failure can only be done by a node of the concerned lump index.

The node that detects this failure must engage a substitution process. To start the process, a node has to first sends a message *SetBusy* to the nodes that compose the lump index. At the receipt of this message the nodes of the lump index set the flag of the corresponding index to 'busy'. Afterwards, the nodes that performs the substitution process retrieves a copy of the file from the other nodes storing a replica using the process described in Section 4.2.3 .

If the node is unable to retrieve a copy of the file, it means that the file is lost, therefore the node stops the substitution process and starts a forced deletion process. The forced

deletion process is similar as the process described in the Section 4.2.5, where the message *DltFile* is replaced by the message *FrcDltFile*. The forced deletion process can only be started by a node that compose the lump index of the concerned file.

If the node performing the substitution process succeeds in retrieving the file, it randomly chooses an element of I_{init}. Then, it sends a message *ScttrRep* using the 'chain of lumps' with the chosen element as key. When one node of the lump that handles the key receives such a message it randomly choose a node of the lump and sends a message *StrRep* to this node. The message *StrRep* is described in the Section 4.2.2. This message informs the receiving node that it has to store a replica of the file.

```
substitute(file_id) (Input:integer)
 1: file ← retrieveFile(file_id) ;
 2: if file = null then
 3:     return sendDHT(buildFrcDltFile()) to file_id ;
 4: scttrRep_mess ← buildScttrRep(file, file_id) ;
 5: key ← rand(0, 2^128 − 1) ;
 6: sendDHT(scttrRep_mess) to key ;

Messages handling
    ...
 7: upon receiveDHT FrcDltFile do
 8:     key ← getKeyIn(FrcDltFile.content.snd_loc) ;
 9:     file_id ← DltFile.content.file_id ;
10:     file_index ← getIndex(file_id) ;
11:     if file_index = null then
12:         return sendDHT(buildFileNotFnd(file_id)) to key ;
13:     lump_index ← getLumpIndex(file_id) ;
14:     if lump_index = null then
15:         return ;
16:     snd_id ← FrcDltFile.content.snd_id ;
17:     if not lump_index.nodeId_list.contains(snd_id) then
18:         return sendDHT(buildNotAllwd()) to key ;
19:     sendDHT(buildDltIdx(file_id)) to file_id ;
20:     for all loc ∈ file_index.replica_location do
            sendDHT(buildDltRep(file_id)) to getKeyIn(loc) ;
    ...
```

Figure 4.11: The replica substitution process

The pseudo code of the substitution process is shown in the Figure 4.11. The function retrieveFile takes a file identifier as argument and returns the file if it can be retrieved using the process described in 4.2.3. The function returns null if the file cannot be retrieved. The function getLumpIndex returns the representation of the lump that stores the index corresponding to the file identifier given in argument. If such a lump cannot be found in

the list of lumps of the caller the function returns null. The function buildFrcFile builds a message *FrcFile*.

Chapter 5

Routing from node to node

The ROSA protocol endowed with the 'chain of lumps' offers a resilient network topology and a resilient distributed HashTable. We have seen in the previous section how a reliable storage system can be built using the 'chain of lump'. In the Section 5.1 we detail how we can also build a resilient and routing service over the 'chain of lumps'. And in the Section 5.2 is described another way to compute the nodes identifiers in order to be compatible with the identification system of the network over which ROSA is deployed.

5.1 Description

As described in the Chapter 3, the 'chain of lump' allows any node of ROSA to send data packets to the nodes of the lump handling a given key. The 'chain of lumps' can be directly used in the case where some network services are confided to lumps. The nodes can access to these services by sending requests to the concerned lumps. When a node wants to communicate with the node that has a given identifier, it has to use the routing service. This routing service uses the 'chain of lumps' and is based on a routing tables systems. The routing tables are stored by the lumps and are accessed via the chain of lumps. In the Section 5.1.1 we detail how the routing tables are built and maintained. In the Section 5.1.2 the routing process itself is described. And to finish we describe, in the Section 5.1.3, how the nodes that do not handle any key can send data packets to a node or receive some from another nodes.

5.1.1 Building and maintaining routing tables

Each node of ROSA that wants to receive data packets form other nodes must build and maintain its own routing table. The routing table of a node is composed of its identifier, its location and a flag. We recall that the identifier of a node is a 128 bit value. We also recall that the node location is the bounds of the sub-internals handled by the lumps whose the node belongs.

A routing table is described in the Figure 5.1. In this table we can see the identifier of the node (ID_n), a flag and the node locations. If the flag is set to TRUE it specifies that the node handles a key of the 'chain of lumps'. If the flag is set to FALSE it means that the node does not handle any key. The Section 5.1.3 is consecrated to this last case.

When a node connects to ROSA it has to build its initial routing table. This initial

Node ID	sub-interval flag	Sub-intervals
ID_n	TRUE	$I_1 = [a_1, b_1]$
		$I_2 = [a_2, b_2]$
		...

Figure 5.1: The routing table used for routing to the node n.

routing table contains the bounds of the sub-intervals handled by the first lump that the node joins. Once that the node has built the table, it includes it in a message *StrTable*. The node sends this message, using the process described in Section 3.2.3.3, to the nodes composing the lump that handles the key corresponding to the node identifier. The node also keeps a copy of its current routing table.

```
storeTable()
 1: table ← buildTable(node.id, node.lump_list) ;
 2: strTable_mess ← buildStrTable(table) ;
 3: sendDHT(strTable) to node.id ;
 4: node.currentTable ← table ;
updateTable()
 5: table ← buildTable(node.id, node.lump_list) ;
 6: if table ≠ node.currentTable then
 7:     updtTable_mess ← buildUpdtTable(table, node.currentTable) ;
 8:     sendDHT(updtTable) to node.id ;
 9:     node.currentTable ← table ;
10:     return ;
11: sendDHT(buildTableAlv()) to node.id ;
Messages handling
    ...
12: upon receiveDHT StrTable do
13:     storeTable(StrTable.content.table) ;
14: upon receiveDHT UpdtTable do
15:     updateTable(UpdtTable.content) ;
    ...
```

Figure 5.2: The store and update table process

When a node of the concerned lump receives the message *StrTable*, it extracts the table from the message and stores it. Consequently, each node that composes this lump stores the table.

A node that wants to receive data packets also has to keep its table up to date. In order to do so, at regular time interval the node looks down its list of lumps and checks, by comparison with the currently stored table, if its table has to be updated. The table

of a node must be updated when the node joins or leaves a lump, or when the set of the sub-intervals handled by the lump whose the node belongs is modified. If the table has to be updated the node builds a message *UpdtTable* and sends it to the lump that is in charge of its table. This message contains all the necessary information to update the table. When these nodes receive the message *UpdtTable*, they update the corresponding table. If the table does not need to be updated, the node must sent a message *TableAlv* to the lump that is in charge of storing its table. This message is used to notice the nodes storing the table that the table must not be deleted.

The pseudo code corresponding to the store and update table process is described in Figure 5.2. The function updateTable has to be added to the ROSA main loop. This function performs the check and the update of the routing tables when it is necessary. The function buildTable builds the table of the caller node. This function uses the node identifier and list of lumps to build the table. The function fills the entry of the table corresponding to the location of the node according to the sub-intervals handled by the lumps of the list. If no such a sub-interval is found the function lets this entry empty and sets the flag to FALSE. Else the function sets the flag to TRUE. The functions storeTable and updateTable respectively store and update the table contained in the messages given in argument. The functions buildStrTable, buildUpdtTable and buildTableAlv respectively build the messages *StrTable*, *UpdtTable* and *TableAlv*.

The fact that the routing tables are stored by the lumps, implies that when the lumps are modified the tables have to be redistributed. When a lump is split, and therefore some sub-intervals are split too, the tables have to be redistributed. Since after a split the list of lumps of the nodes is modified, it may happen that a node stores the table of a node with an identifier that is not anymore contained in the handled sub-intervals. In this case, this node has to discard this table. When a lump is absorbed, tables have to be redistributed too. When a lump is absorbed, some nodes become members of lumps that possess sub-intervals that were not possessed by the lumps of their old lumps set. The corresponding tables have to be forwarded to these nodes by the other nodes.

The tables have also to be redistributed when the 'chain of lumps' is modified during an optimization. When optimization, such the load balancing or the reduction of the sub-interval number, are used over the 'chain of lumps' the distribution of the tables may be inappropriate. As seen in the Section 3.2.5.1, during the load balancing a node may modify the lengths and the bounds of two contiguous sub-intervals. The reduction of the sub-intervals number shown in the Section 3.2.5.2 consists in a node confiding a sub-interval handled by a lump to another lump. In order to maintain the good distribution of the tables the node that performs these optimizations send messages *AddTable* and *DltTable*. These messages are equivalent to the messages *AddIdx* and *DltIdx* shown in the Section 4.2.6.1 dealing with the preservation of the file indexes of the storage system.

In order to avoid that the tables of the nodes that have failed or left the network, or that the table of the node that does not anymore want to publish their locations encumber the network, the routing system is endowed with a cleaning mechanism. The node that stores the table of a node and that have not received messages *UpdtTable* or *TableAlv* since a given amount of time must delete the corresponding table.

5.1.2 Routing a message to a node

Let id_1 be the identifier of a node n_1 of ROSA. When a node n_2 with the identifier id_2 wants to send a data packet to the node n_1, it first encapsulates the data packet into a message *SendToTable*. This message in addition to the data packet contains the identifiers of the sending node id_2 and of the targeted node id_1. Afterwards, the node n_2 sends the message *SendToTable* to the lump that handles the key corresponding to id_1.

The nodes that compose this lump may store the routing table of the node n_1, if n_1 has wished publish it. When a node that composed the lump supposed to store the routing table of n_1 receives the message *SendToTable*, it checks that the node n_1 has published its routing table. If not, it replies with a message *TableNotFound*. If the node n_1 has published its routing table, the node that received the message *SendToTable* looks down the node location list contained in the table and determines the one that is the closer to itself in term of 'chain of lumps'. Once this node location is determined, the node that received the message *SendToTable*, extracts a key from this node location and sends a message *SendToNode* to the lump handling the extracted key. This node n_1 is in this lump, since the node locations that are in the routing table of n_1 corresponds to the bounds of a sub-interval handled by a the lumps that n_1 composes. The message *SendToNode* contains the data packet,the identifiers of the sending node id_2 and of the targeted node id_1

Upon the receipt of the message *SendToNode*, a node of the targeted lump sends it to the node n_1 (if the node that received the message is not n_1 itself). Since all the nodes that compose a lump are neighbors, the last step is not a problem except if the table is not up to date or if n_1 does not handle any key. The last case will dealt in the Section 5.1.3.

It may happen that the table is not up to date if the node n_1 has a failure, leaves the network, etc. In this case, the node that received the message *SendToNode* sends a message *NodeNotFnd* to the lump that stores the routing table of n_1. Once this message reaches a node of this lump, the node sends a message *DltLoc* to the nodes of the lump and checks if there is another node location in the table of n_1. If this location exists then the message is sent again using the new node location, else the message is discarded. A message *DltLoc* contains a node identifier and a node location. A node receiving a message *DltLoc* checks if it stores the routing table corresponding to the node identifier contained in the message. If it does, it removes the node location from the routing table.

The pseudo code of this process is described in the Figure 5.3. The function getTable returns the table corresponding to the node identifier passed in argument. If the caller does not store such a table, the function returns null. The function removeLoc removes the location corresponding to the key given in argument from the table. The functions buildTableNotFnd, buildSendToTable, buildDltLoc respectively build the messages *TableNotFnd*, *SendToTable* and *DltLoc*.

The average number of hops needed for a node to send a data packet to a another node is obviously equals to twice the average number of hops needed for a node to send a data packet to a specified lump as it is described in the Figure 5.4. In this figure the node shortcuts are deliberately ignored in the purpose of more clarity. In this figure, a node wants to send a data packet to the node that has 276 as identifier. The node sends through the 'chain of lumps' the data packet encapsulated in a message *SendToTable* to the lump handling the routing table of the node 276. When the message reaches the lump, the closest node location is chosen and a message *SendToNode* is sent using the 'chain of

```
sendToNode(id, packet) (Input:integer, data)
1: sendDHT(buildSendToTable(packet, id, node.id)) to id ;
Messages handling
   ...
2:  upon receiveDHT SendToTable do
3:      table ← getTable(SendToTable.key) ;
4:      if table = null then
5:          key ← SendToTable.content.send_id ;
6:          return sendDHT(buildTableNotFnd()) to key ;
7:      loc ← getCloserLocation(table.loc_list) ;
8:      key ← getKeyIn(loc) ;
9:      sendDHT(buildSendToNode(packet, id, node.id)) to key ;
10: upon receive SendToNode do
11:     if node.id = SendToNode.content.dest_id then
12:         extractDataPacket(SendToNode.content) ;
13: upon receiveDHT SendToNode do
14:     dest_id ← SendToNode.content.dest_id ;
15:     for all lump ∈ lump_list do
16:         if lump.handle(SendToNode.key) then
17:             if lump.nodeId_list.contains(dest_id) then
18:                 send(SendToNode) to dest_id ;
19:     sendDHT(builDltLoc(SendToNode.key)) to dest_id ;
20: upon receiveDHT DltLoc do
21:     table ← getTable(DltLoc.key) ;
22:     if table = null then
23:         key ← DltLoc.content.send_id ;
24:         return sendDHT(buildTableNotFnd()) to key ;
25:     table.removeLoc(DltLoc) ;
26:     loc ← getCloserLocation(table.loc_list) ;
27:     key ← getKeyIn(loc) ;
28:     sendDHT(buildSendToTable(packet, id, node.id)) to key ;
   ...
```

Figure 5.3: Routing a data packet from a node to another node

lumps' to the lump that corresponds to this node location. Once the message reaches the lump, the data packet is sent to the node 276.

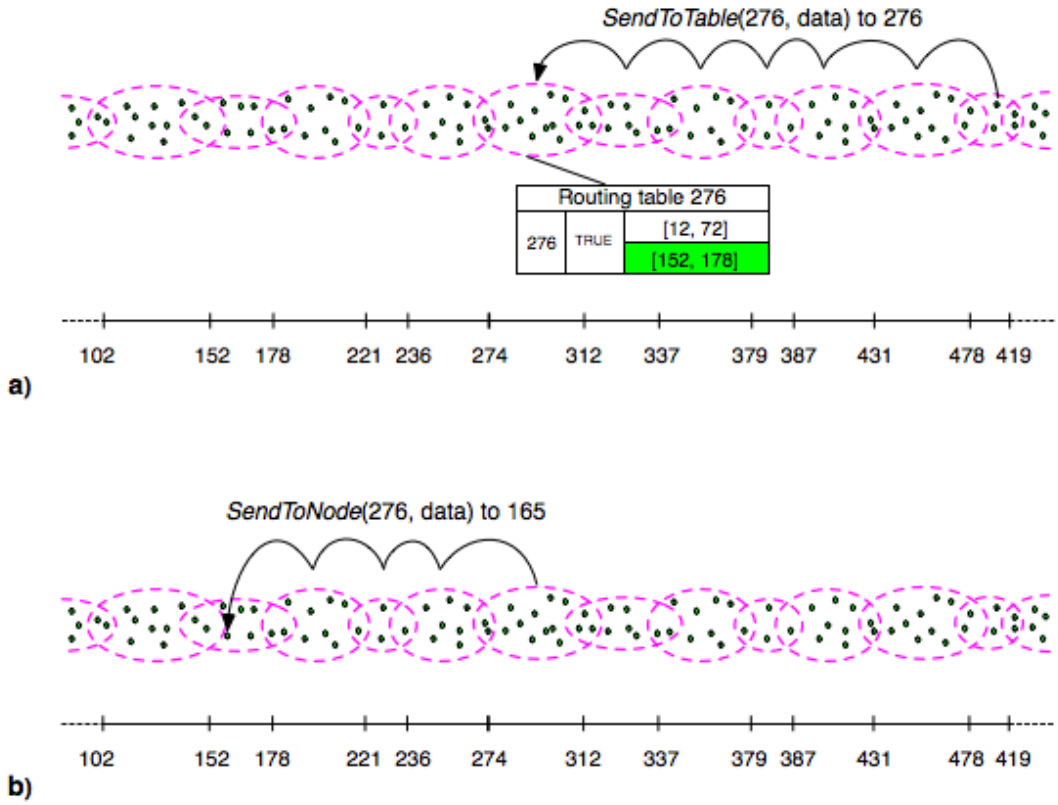


Figure 5.4: A node to node routing example

5.1.3 Dealing with nodes that does not handle any key

It may happen that after the split of a lump, in the worst case, one of the resulting lumps does not handle any sub-interval and it may also happen that a node only belongs to lumps without sub-interval.

This is very rare case but it is interesting that these nodes can also send and receive data packets.

To build and to maintain its routing table, a node that does not handle any key has to periodically send a message *updateTable* to its neighbors. Let the identifier of this node be *id*. This message only contains the node identifier, no sub-intervals and the flag set to FALSE. This message is forwarded from node to node using a random walk algorithm until the message reaches a node that handles a key. When such a node receives the message, it completes it with its own location and sends it using the 'chain of lumps' to the lump that handles *id*. The *updateTable* message is then handled as any other *updateTable* message.

To allow a node that does not handle any key to receive a data packet, a similar process than the one described in the Section 5.1.2 is used. The message *SendToNode* is replaced with a message *SendToNoKeyNode*. This message is sent by the sender node to the lump that corresponds to the node location contained in the routing table. The first node of the lump that receives this message sends it to its neighbors. Then the neighbor that receives this message and that does not handle any key checks if it is the destination node and

if it is not, it sends it to their neighbors. This process does not ensure that destination node will be reached by the message. It depends if its routing table is up to date. If the destination node is not reached the message is lost.

5.2 Finding an appropriate node identifier

Since the nodes of ROSA are already identified by an physical address, it could be interesting to define the nodes identifier in such a way that given a node identifier one is able to retrieve the corresponding physical address. Consequently, to compute the identifier of a node one must use a reversible function. This function has also to ensure a fair distribution of the node identifier among I_{init}. The reversible mixing functions are functions that can be used to obtain the nodes identifiers. In the Section 5.2.1 we present the concept of reversing mixing functions. Then we detail, in the Section 5.2.2, how we can compute the nodes identifiers compatible with the IP identification.

5.2.1 Introduction to mixing functions

The mixing functions are the main component of the hash function. The purpose of a mixing function is to take an input, to mix and transform the bits, and to return an output of the size of the input. A good mixing function have for attributes an uniform distribution and a strong collision resistance. If the mixing function is reversible, it is obvious that collisions are guaranteed not to happen.

A mixing function is reversible if it is a combination of reversible operations. In the Table 5.1 are listed some reversible and non reversible operations on unsigned n-bit integer. These operation must be performed modulo 2^n. The bitwise left and right shift operations and respectively noted $<<$ and $>>$ and c is a constant.

Non-reversible operations	Reversible operations	
output = input & c	output = input $\oplus$ c	
output = input $<<$ c	output = input + c	(1)
output = input $>>$ c	output = input $-$ c	(2)
output = input | c	output = input $\times$ (2 $\times$ c + 1)	
output = input $\times$ (2 $\times$ c)	output = input $\oplus$ (input $<<$ c)	(3)
output = input / c	output = input $\oplus$ (input $>>$ c)	(4)
output = input % c	output = input $\oplus$ (input $<<$ c)	
output = input + (input $>>$ c)	output = input + (input $<<$ c)	

Table 5.1: The non reversible and reversible operations

In the mixing function that is proposed in Section 5.2.2.1 we use the reversible operations (1), (2), (3) and (4). In the Table 5.2 are shown the corresponding inverse operations. In this table n refers to the n bits of the input and c is a constant.

A mixing function achieves an uniform distribution if it satisfies the avalanche criterion of degree 1 also called strict avalanche criterion. The strict avalanche criterion of degree 1 is respected if whenever a single input bit is complemented, each of the output bits changes with a probability of one half. This criterion was introduced by Webster and Tavares in

Operations	Inverse operations
output = input + c	output = input − c
output = input − c	output = input + c
output = input $\oplus$ input << c	output = $\bigoplus_{\substack{i=0 \\ i<n/c}}$ input << $c \times i$
output = input $\oplus$ input >> c	output = $\bigoplus_{\substack{i=0 \\ i<n/c}}$ input >> $c \times i$

Table 5.2: The chosen reversible operations and their inverse operations

1985 (Webster and Tavares [1986]). It is not possible to prove for large inputs that a mixing function satisfies the strict avalanche condition. For large inputs only probabilistic evaluations are possible.

5.2.2 Computing the ROSA node identifier

In this Section and according to the definition of the mixing function described in Section 5.2.1, we propose the reversible mixing function that is used to compute the nodes identifier. We also evaluate if the proposed reversible mixed function satisfy the strict avalanche condition.

5.2.2.1 The chosen mixing function

To build the mixed function we take inspiration from the article 'Reversible data mixing procedure for efficient public-key encryption' (Matyas et al. [1998]). The mixing process described in this article consists of dividing the block of data into a left-hand part L and a right-hand part R. It is preferable from a security point of view that these parts are equal-sized. Then the mixing process consists in performing some mixing operations on one of the part and Exclusive-ORing it with the other part. Four iterations of this mixing process are sufficient to ensure that the strict avalanche condition is satisfied. To finish the two parts are concatenated in order to obtain the final output.

Our reversible mixed process takes an IP address in input and must return a 128-bit output. An IP address is composed of 4 8-bit unsigned integer. Each of these unsigned integer are transformed into 32-bit unsigned integers. These 4 32-bit integers will be the four parts that will manipulate our mixed function. Given an IP address A.B.C.D the resulting parts will respectively be P_A, P_B, P_C and P_D. As for the previously described mixing process described only four steps are sufficient to ensure that the strict avalanche condition is satisfied.

The four steps of our mixing process are:

- P_A is mixed to obtain P_A^1. The function used to mix P_A is a reversible mixing function and will be described further in this Section. Then P_B, P_C and P_D are Exclusive-ORed with P_A^1 to respectively obtain P_B^1, P_C^1 and P_D^1.
- P_B^1 is mixed to obtain P_B^2. Then P_A^1, P_C^1 and P_D^1 are Exclusive-ORed with P_B^2 to respectively obtain P_A^2, P_C^2 and P_D^2.

- P_C^2 is mixed to obtain P_C^3. Then P_A^2, P_B^2 and P_D^2 are Exclusive-ORed with P_C^3 to respectively obtain P_A^3, P_C^3 and P_D^3.
- P_D^3 is mixed to obtain P_D^4. Then P_A^3, P_B^3 and P_C^3 are Exclusive-ORed with P_D^4 to respectively obtain P_A^4, P_B^4 and P_C^4.

Since the function used to mix P_A, P_B^1, P_C^2 and P_D^3 is a reversible function. The unmixing process exists and consists in simply the inverse of the mixing process and will not be detailed. The mixing process is schematized in the Figure 5.5.

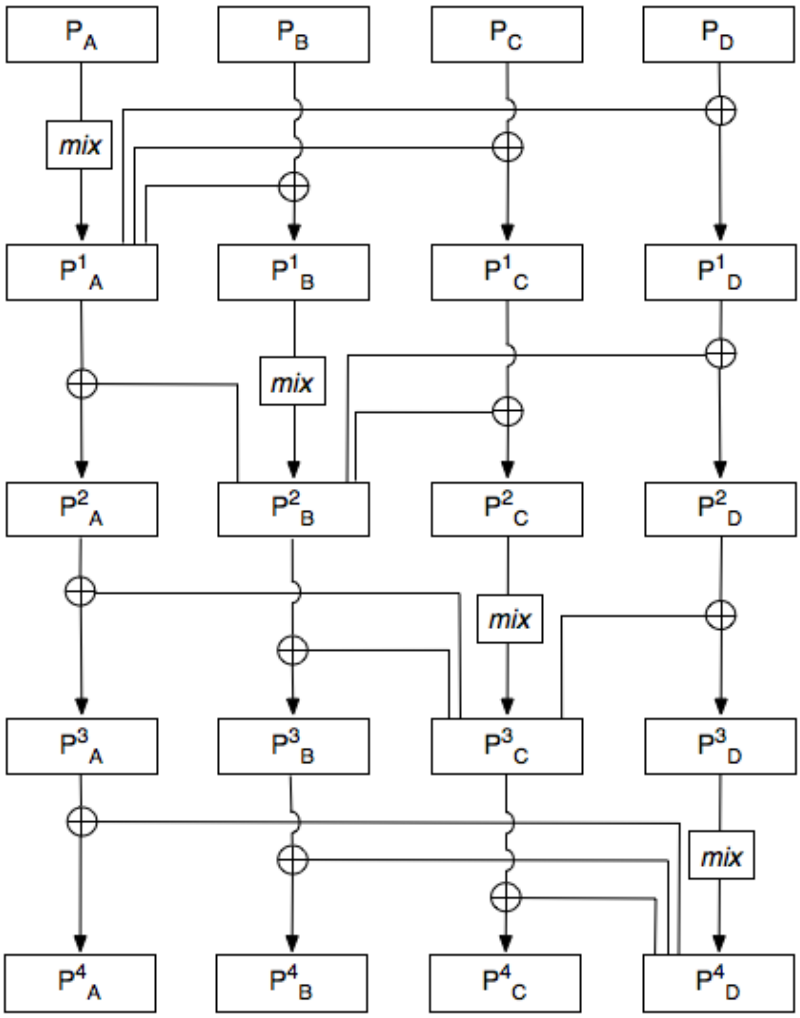


Figure 5.5: Mixing process

In order to be reversible, the function *mix* used to mix P_A, P_B^1, P_C^2 and P_D^3 must be a combination of reversible operations on unsigned 32-bit integer. The reversible operations chosen are the operations (1), (2), (3) and (4) of the Table 5.2. The algorithm of the *mix* function is represented in the Figure 5.6. This function was found empirically. Many different functions were tested until one makes the whole mixing process satisfy the strict avalanche condition.

mix(part) (**Input:***integer*; **Output:***integer*)

```
1: result ← part + 944 ;
2: result ← result ⊕ result << 1 ;
3: result ← result − 66240 ;
4: result ← result ⊕ result >> 2 ;
5: result ← result + 439654944 ;
6: result ← result ⊕ result << 4 ;
7: result ← result − 1937066240 ;
8: result ← result ⊕ result >> 8 ;
9: result ← result + 419501280;
10: result ← result ⊕ result << 16 ;
11: result ← result − 2516448256 ;
12: result ← result ⊕ result >> 1 ;
13: result ← result + 1402087808 ;
14: result ← result ⊕ result << 2 ;
15: result ← result − 2699718400 ;
16: result ← result ⊕ result >> 4 ;
17: result ← result + 2132352512 ;
18: result ← result ⊕ result << 8 ;
19: result ← result − 1240660480 ;
20: result ← result ⊕ result >> 16 ;
21: result ← result + 4009274112 ;
22: result ← result ⊕ result << 3 ;
23: result ← result − 129839136 ;
24: result ← result ⊕ result >> 9 ;
25: result ← result + 234628412 ;
26: result ← result ⊕ result << 9 ;
27: result ← result − 56931536 ;
28: result ← result ⊕ result >> 3 ;
29: return result ;
```

Figure 5.6: The *mix* function

5.2.2.2 Evaluating the mixing function

Since the size of the input is equal to 128 bits the avalanche behavior cannot be formally determined. In order to estimate it for the proposed mixed function we randomly choose an important number of input values. For each of these values, we first compute the output value, then toggle each bits of this input value and for each of the new input values obtained we compute the output value and compare it to the initial output value. The result for each randomly chosen input is stored. When a sufficient number of input values are tested we have a good estimation of the probabilities that a given input bit affects a given output bit. The number of random input values used in our evaluation is equal to 10^9.

The Figure 5.7 represents the evaluation of the probabilities of the proposed mixing function. The figure is composed of a grid of 128 coloured squares of width and 128 coloured squares of height. The color of the square located at (i,j) on the grid corresponds to the probability that the bit i^{th} bits of the input affects bit j^{th} bit of the output. One can remark that all these probabilities are included between 49,4% and 50,6% and that the majority of these probability are nearly equal to 50%. Consequently, the proposed mixing function is an appropriate mixing function for our purpose.

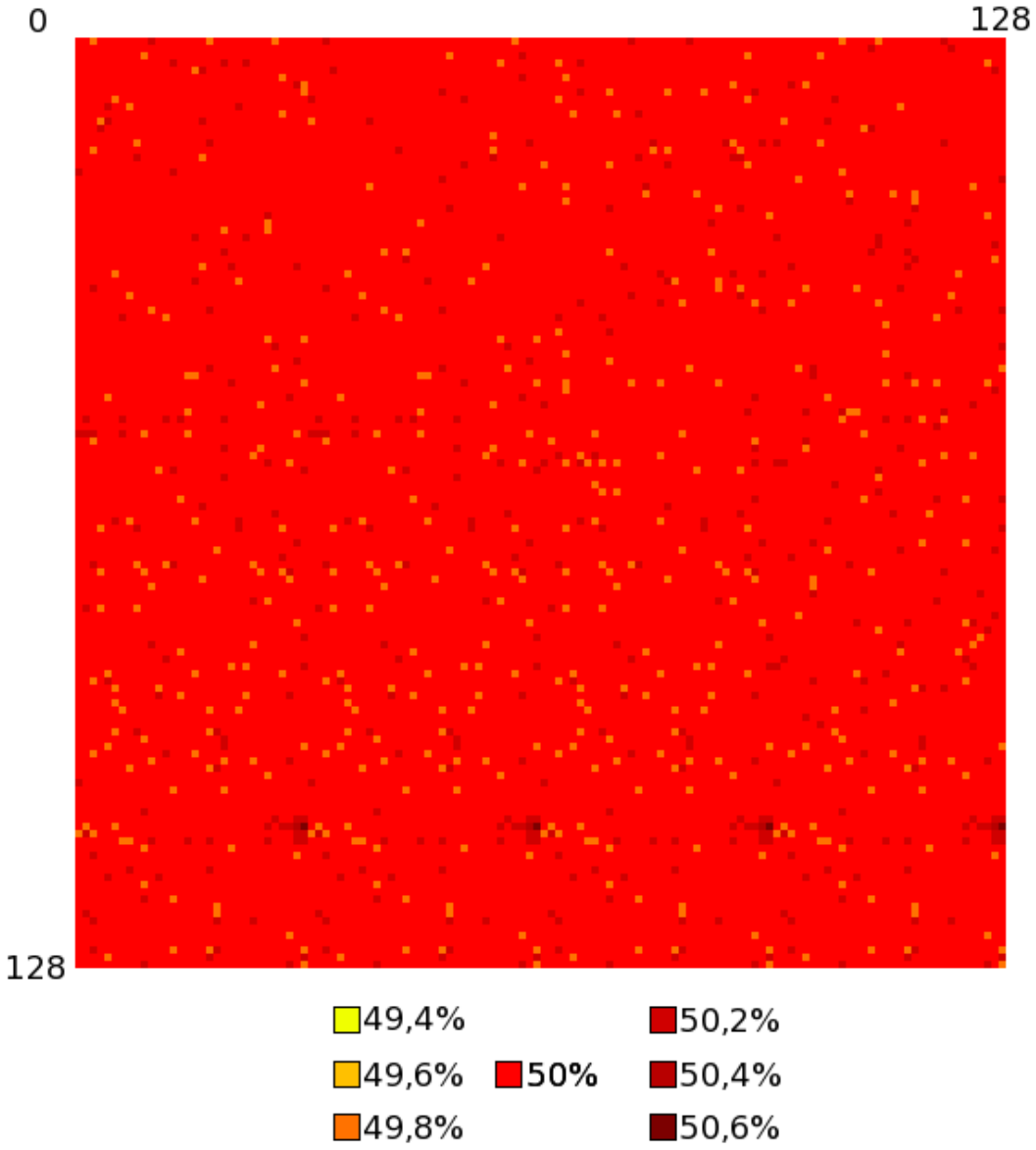


Figure 5.7: Propagation of the mixing function

Chapter 6

Density

In this Chapter we deal with the density. We introduce the notion of density and recall when and why the density is necessary in the Section 6.1. Afterwards we present the three densities that was conceived during this thesis. In Section 6.2 we detail the default density. In Section 6.3, we detail the density used to deploy ROSA over an IP network and offer a strong tolerance to underlying network failures. And in Section 6.4 we present a density that can be used to deploy ROSA over large and dense mobile network.

6.1 Introduction

We have seen in the Chapter 2 that the topology management of ROSA consists in the nodes computing the density of lumps which they belong. Each node sends to its neighbors the representation of the lump of that has the lowest density in its list of lumps. Consequently, every node of ROSA is aware of some lumps that have low densities. According to this knowledge about the densities of the lumps, the nodes leave lumps with high densities in order to join and to increase the density of the lumps with low densities.

The density is the parameter that defines the behavior of ROSA. If two instances of ROSA were deployed over the same network (i.e same nodes, same failures and same network topology) but with two different definitions of the density, the nodes of each instance will choose which lumps to leave or to join according to the definition of the density of the instance. Consequently, the set of lumps of these instances of ROSA will be completely different.

The density is the most important parameter of the ROSA protocol, the density of an instance of ROSA must be chosen to reflect the goal whose ROSA is intended to achieve. During this thesis, three definitions of the density were proposed, they will be presented further in this Chapter. But one can define a new density in order to use ROSA for another purpose. ROSA can be adapted to many usages simply in modifying the definition of the density, this is a very strong point of ROSA

6.2 Default density

The default density of a lump is equal to size of the lump. This is the simplest density. It was first used for the purpose of testing how the topology management process of ROSA

behaves. Nevertheless, this density can be used to average and maximise the number of nodes per lump. This density has three advantages:

- it does not require any additional data to allow nodes to compute it ;
- it is easily computable ;
- it can be used by ROSA on any types of networks.

We performed a simulation to see how this density affects the network. The simulation shows how the average number of nodes per lump evolves from the initialization of ROSA to a state of stability (i.e. a state where if no failure occur no more changes happen in the topology). The time in this simulation is measured in ROSA cycles. A ROSA cycle is the time that is necessary to the nodes to run the ROSA main loop described in the Section 2.4.7. The number of nodes involved in this simulation is equal to 100 and the maximum number of nodes that can compose a lump is equal to 10. The first 100 cycles correspond to the initialization of ROSA. During this initialization a node joins every cycle. The results of this simulation are represented in the graphics Figure 6.1 and Figure 6.2. The first graph shows the average number of nodes of the lumps in function of the number of ROSA cycles elapsed. The second graph shows the distribution of lumps in percentage in function of the number of nodes that compose these.

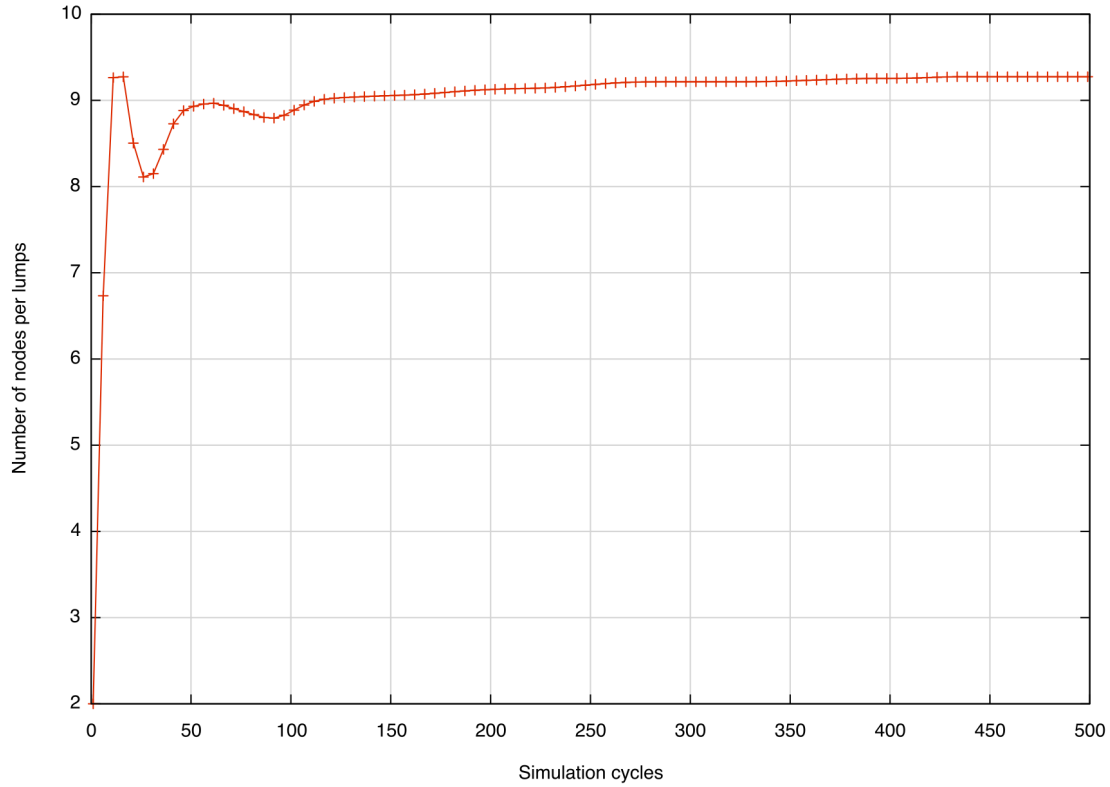


Figure 6.1: Average number of nodes per lumps

In the Figure 6.1, one can see that the final average number of nodes per lump is about 9.3. One can remark that the process, described in the Section 2.4.11, in charge of

optimizing the average density of the lumps of ROSA (in the present case the number of nodes per lump) works well since from the ROSA cycle 100 to the ROSA cycle 500 the average number of nodes per lump increases from 8.8 to 9.3. The fluctuations occurring in the first 100 ROSA cycles are due to the nodes connecting to ROSA.

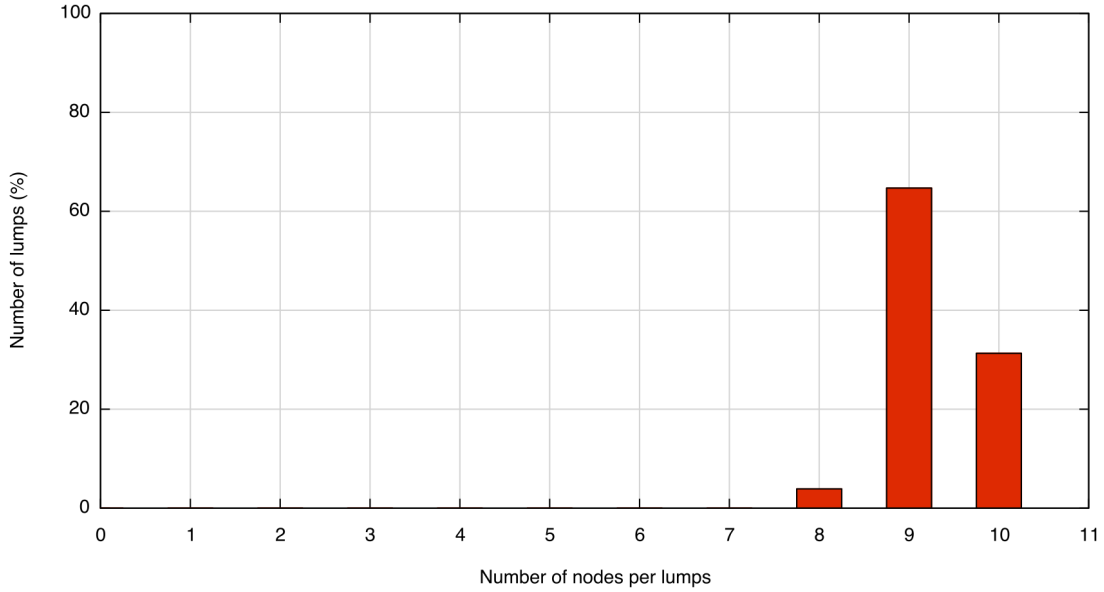


Figure 6.2: Distribution of the lumps according to their number of nodes

In the Figure 6.2 one can see that the minimum number of nodes per lump is equal to 8 and that only 3.9% of the lumps are only composed of 8 nodes. Most of the lumps, 64.7%, are composed by 9 nodes. And 31.3% of the lumps are composed by 10 nodes, i.e. the maximum number of nodes possible.

This simulation confirms the fact that this density can be used to maximize the number of nodes per lump and therefore the number of neighbors per node (the maximum number of neighbors per node is equal to the maximum number of nodes per lump multiplied by the maximum number of lumps per node).

6.3 Resilient density

The purpose of the resilient density is to make ROSA extremely resistant to failures. Since computer-based systems are nowadays present in the economy, medical processes and equipment, power and communication infrastructures, it is essential to protect these systems from network failures. Studies from Paxson [1998] and Zhang et al. [2000] have demonstrated that Internet connectivity failures are not rare. In 1.5% to 3.3% of time, failures prevent pair of hosts from communicating.

Since BGP Rekhter and Li [1995] can take much time, many minutes sometimes, to discover a failure and to reorganize the routes in a consistent form Labovitz et al. [2000], we have defined a density in order to make ROSA able to assure a resilient routing to these computer-based systems. It means that ROSA will be deployed over an IP network.

A lot of Overlay Networks have already dealt with the problem of resilient routing. The most famous is undoubtedly RON (Resilient Overlay Network) Andersen et al. [2001]. RON maintains a virtual link between all pairs of nodes. Each node in RON checks the availability and the capacity of the virtual links between itself and the other nodes. Each node decides then, based on its knowledge, if it should let packets flow directly to other nodes, or if some indirections via other RON nodes could be useful. But since each node in RON is linked with all the other nodes, RON cannot exceed more than one hundred of nodes.

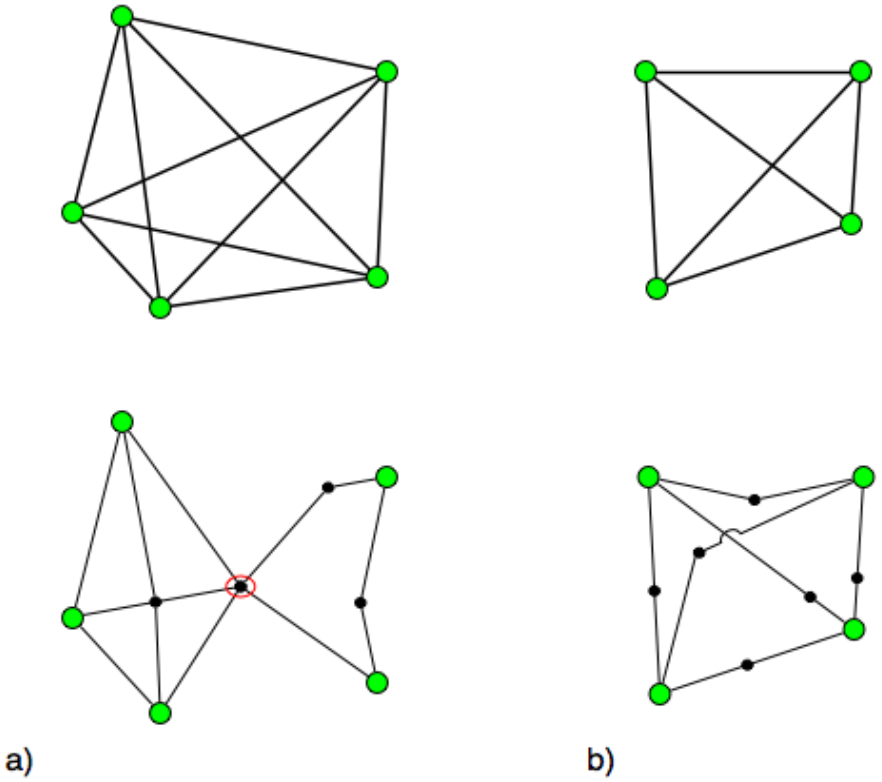


Figure 6.3: To examples of resilient density for two lumps

This limit on the maximum number of nodes does not allow to use RON as a backbone for large network. To improve RON scalability, the solution chosen by DG-RON Qazi and Moors [2007] consists in splitting the network into logical zones in such a way that each node has to maintain and exchange information only with nodes of its logical zone. But this solution does not take into account of the specificities of the topology of the underlying network. A solution proposed in Akihiro et al. [2006] consists in exploring the underlay network topology in order to make the nodes efficiently choose these neighbors. Nevertheless, the algorithm proposed can only be used to construct static network. The static network are not able to react to failures and therefore are not very resilient.

From these observations, we decided to define a density that will force the node of ROSA to dynamically reorganizes their neighbors sets according to:

- the topology of the network which ROSA is deployed on ;

- the failures on the elements of the links between two nodes.

Let l be a lump, the density of l is the quantification of its capacity to maintain a path between all the nodes that compose if despite the presence of virtual link failures. We assume that each node is able to identify and distinguish the elements of the underlying network composing the virtual links that connect it to its neighbors. If ROSA is deployed on an IP network the `traceroute` utility (Jacobson [1989]) allows any node to do this.

The density is defined as the minimal number of failures on the elements of the virtual links of the lump that are necessary to isolate a node of the lump. In such a way that if the number of failures is less than the density, we can affirm that there exist a path between any two nodes of the lump and that the nodes of the lump still able to communicate each other. One can remark that if all the links between the nodes of a lump are disjoints, the density of the lump is simply equal to the cardinal of the lump minus one, i.e. #l - 1.

In the Figure 6.3 are represented two lumps and the graph of the elements of the underlying network that correspond to these lumps. Each lump is displayed above its corresponding underlay graph. The green circle are the nodes, they are present in the lumps and in the underlay graph. The little black dots are the underlay elements. One can see that the size of the first lump (a)is equal to 5 and the size of the second (b) is equal to 4. The density of the first lump is equal to 1 since the failures of the underlay red circled element failure are sufficient to isolate the nodes of the left to the nodes of the right. The density of the second lump is equal to 4 since all the links are disjoints.

In order to compute the density of a lump, a node needs to know the graph of the underlay elements that compose the links between the nodes of the lump. That is why when a node joins a lump, it explores the underlay topology of the links between itself and the other nodes of the lump. This information about the topology of the links is added to the representation of the lump when the node joins. Since as it was shown in the Section 2.4.4, this representation is sent to the other nodes of the lump during the join, any node of the lump has the capability to compute the density.

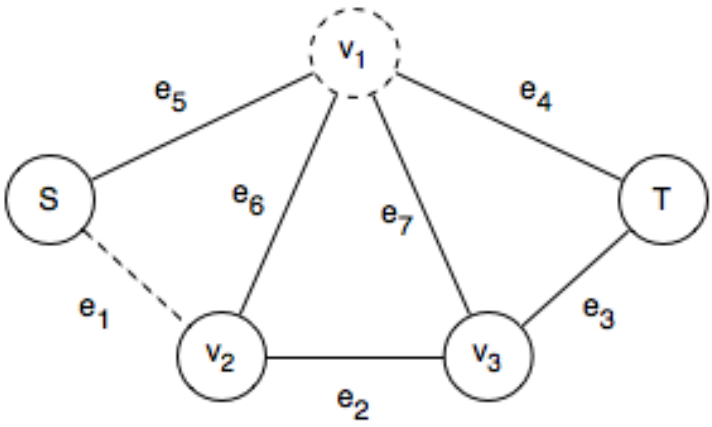


Figure 6.4: An example of s-t cutset

A node that wants to compute the density of a lump must build the underlay graph corresponding to the information about the topology of the links of the lump. If ROSA is

deployed over an IP network, the vertices of the resulting graph are the router, gateway, etc, and the edges of the resulting graph are the cables linking the vertices. Once the graph is built, if we consider that both the vertices and the edges fail, to compute the density of the lump one must first compute the minimal s-t cutsets separating every pair of nodes of the lump. The density is equal to the cardinal of the smallest of the computed minimal cutsets.

The minimal s-t cutset, separating two vertices source and terminal, is the minimal set of vertices and edges whose removal disconnects the vertex source from the vertex terminal. In the Figure 6.4, is shown a graph, one of the minimal s-t cutset separating the vertices S and T is equal to $\{v_1, e_1\}$ (dashed in the figure).

We have looked, in the literature dealing with the graph theory, for a solution to, given a graph, a source and a terminal vertex, compute the cardinal of the minimal s-t cutset separating the source from the terminal. Since in a graph composed of n vertices there are $2^n - 1$ minimal s-t cutsets, finding the minimal cutsets of a graph is an hard problem. Many solutions exist in the literature but the one proposed by Fard and Lee Fard and Lee [1999] retained our attention.

```
computeDensity(lump) (Input:lump; Output:integer)
 1: graph ← computeGraph(lump) ;
 2: density ← ∞ ;
 3: for all n1 ∈ lump.nodeId_list do
 4:     for all n2 ∈ lump.nodeId_list such n1 ≠ n2 do
 5:         tmp_density ← getCardMinSTCutSet(n1, n2, graph) ;
 6:         if tmp_density < density then density ← tmp_density ;
 7: return tmp_density ;
getCardMinSTCutSet(id1, id2, graph)
(Input:integer, integer, graph; Output:integer)
 8: st_min_edge_cutset_list ← getMinEdgeCS(id1, id2, graph) ;
 9: st_min_cutset_list ← void ;
10: for all cs ∈ st_min_edge_cutset_list do
11:     st_min_cutset_list ← st_min_cutset_list ∪ getMinCS(id1, id2,
        graph, cs) ;
12: cardinal ← ∞ ;
13: for all cs ∈ st_min_cutset_list do
14:     if #(cs) < cardinal then cardinal ← #(cs) ;
15: return cardinal ;
```

Figure 6.5: Computing the resilient density

The first step of this solution consists in enumerating all the minimal s-t edge cutsets that separates the source from the terminal. A minimal s-t edge cutset, separating two vertices is the minimal set of edges whose removal disconnects these two vertices. In the

Figure 6.4 $\{e_1, e_5\}$, $\{e_3, e_4\}$ and $\{e_1, e4, e_6, e_7\}$ are s-t edge cutsets that separate the vertex S from the vertex T (there exist many other). To enumerate these s-t edge cutsets one can use the algorithm proposed in 1990 by Ahmad S. [1990].

Once the minimal s-t edge cutsets are enumerated, the second step is to use the algorithm presented by Fard and Lee to build the minimal s-t cutsets from the minimal s-t edge cutsets previously computed. The concept used to build the minimal s-t edge cutsets is that *the failure of a vertex inhibits the working of all the edges incident from it.* Therefore, given a minimal s-t edge cutsets the algorithm builds a set of minimal s-t cutsets by adding one or many vertices and removing the edges incident from these. The vertices to add are combination of vertices that are linked by the edge of the minimal s-t edge.

The pseudo code of the function that given a lump computes the density of this lump is shown in the Figure 6.5. The function computeGraph returns the graph corresponding to the links between the nodes of the lump given in argument. The function getMinEdgeCS enumerates the s-t edge cutsets between the nodes given in argument. This is the algorithm described in S. [1990]. The function getMinCS generates the s-t cutset from the s-t edge cutset passed in argument. This is the algorithm presented in Fard and Lee [1999].

In the case where only the vertices are susceptible to fail, the density can be computed as the vertex connectivity of the graph corresponding to the links of the lump. The vertex connectivity of a graph is the smallest number of vertices whose the deletion separates the graph. An efficient algorithm to compute the was proposed by Henzinger, Rao and Gabow in 2000 Henzinffer et al. [1996].

6.4 Mobile density

The mobile density was created in order to adapt ROSA to dense mobile Ad-Hoc network.

ROSA cannot be deployed on every Ad Hoc networks. The Ad Hoc network on which ROSA can be deployed must:

- be dense
- possess a Ad Hoc layer protocol that takes in charge the neighbor discovery and a resilient node to neighbor routing process

Dense means that any node has a large number of potential neighbors. If the network is not dense enough ROSA may fail to constitute lumps big enough to support simultaneous communication failures (that can happen with a non negligible probability on the Ad Hoc network). The inner protocol of the Ad Hoc network on which ROSA can be deployed must be incharge of the discovery of the Ad Hoc neighbors of a node in such a way that any node of ROSA can choose a subset of the Ad Hoc neighbors as the ROSA neighbors. The inner protocol of the Ad Hoc network must also handle the temporary disruption of communication between a node and one of its ROSA neighbors.

In fact, only few Ad Hoc networks possess the necessary capacities to support the ROSA protocol. We show in this Section of these Ad Hoc network, the Aeronautical network.

The mobile density is designed to consider the characteristics of mobile nodes. These characteristics are namely their relative positions, speed and direction. We introduce the concept of the viability of the link between two nodes. The viability of the link between

two mobile nodes is a measure that provides information on the quality and durability of the connection between these two nodes. For example, two mobile nodes going in opposite directions will not be within communication range as long as two near mobile nodes following the same way.

The viability of the link between two mobile nodes n_1 and n_2 is described in the formula below:

$$viability(n_1, n_2) = 1 - \left(\frac{1 - cos(\widehat{\vec{s_1}, \vec{s_2}})}{2} + \frac{d^2}{d_{max}^2} + \frac{||\vec{s_1} - \vec{s_2}||^2}{(2.v_{max})^2} \right) / 3$$

Where $\vec{s_1}$ and $\vec{s_2}$ are respectively the state vector representing the direction and velocity of the nodes n_1 and n_2. The distance between the two nodess is noted d and d_{max} represents the distance beyond which the two mobile nodes cannot communicate and v_{max} represents the maximum speed that a node can reach.

The usage of mobile density assumes that the nodes are able to compute their state vector. This can be done with the help of a embedded GPS (Global Positioning System see Botton et al. [1997]).

The first part of the formula: $\frac{1 - cos(\widehat{\vec{s_1}, \vec{s_2}})}{2}$ varies from 1 if the mobile nodes go in opposite direction to 0 if the nodes go in the same direction. The second part: $\frac{d^2}{d_{max}^2}$ is the ratio of the distance between the nodes distance over the maximum distance. The more this value is close to 0, the more the nodes are close and the more it is close to 1, the more the nodes are distant. The last part: $\frac{||\vec{s_1} - \vec{s_2}||^2}{(2.v_{max})^2}$ compares the relative speeds of the nodes. The more this value is close to 1, the more the difference between the speeds is high and when this value approaches 0 the speeds are equivalent.

A node joining a lump computes the viability of the links between itself and the other nodes of the lumps. These information are added to the representation of the lump. Any node of a lump is then able to compute its density. The density of a lump is the sum of the viability of the communication between the nodes composing the lump. Let l be a lump the density is equal to:

$$density(l) = \sum_{a_1}^{a_1 \in l} \sum_{a_2}^{a_2 \in l - \{a_1\}} viability(a_1, a_2)$$

The Figure 6.6 shows 2 lumps and their relative densities.
On the lump of the left, the four nodes go in the same direction and their relative speeds are equal to 0. The density of this lump is equal to 5.93. On the lump on the right, one node goes in the opposite direction therefore the density of this lump must theoretically be less than the density of the previous lump. Indeed the density of the lump on the right is equal to 3.94.

The aeronautical network is an Ad Hoc network on which ROSA can be deployed. Three types of devices composed an aeronautical network. We have by altitude order (from bottom to top), airports, aircrafts and satellites (that can be replace by High Altitude Platforms Grace et al. [2005]). Airports and satellites compose the backbone network. Each of this device has its own coverage and as show in the Figure 6.7 there is some areas

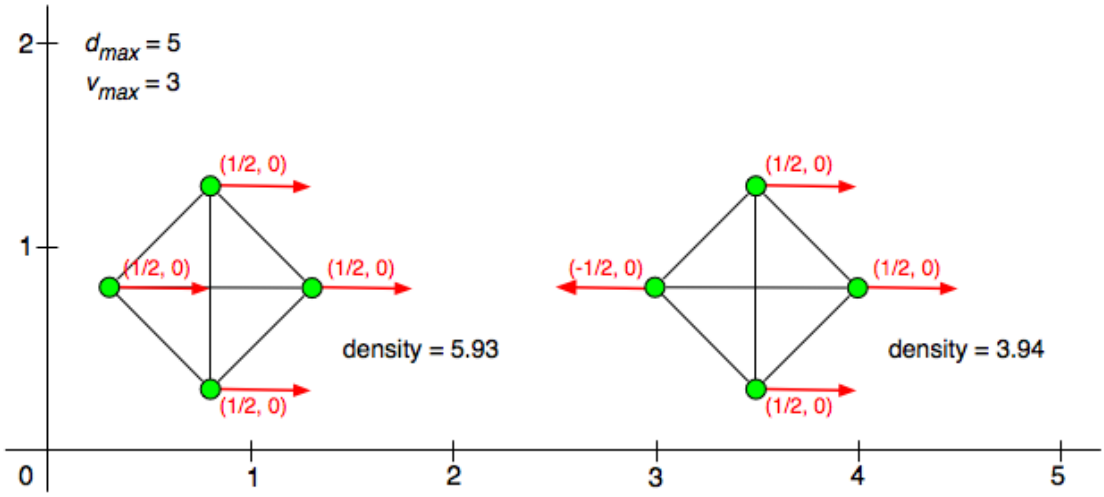


Figure 6.6: Two examples of mobile densities

that are not covered by any airport or any satellite. To connect the aircrafts cruising in this uncovered area to the backbone network, paths among these aircraft have to be maintained and consequently these aircrafts compose a mobile Ad Hoc network.

This aeronautical mobile Ad Hoc network are composed by a large number of aircrafts connected by VHF (Very High Frequency) links. The range of the VHF links may reach many hundreds of kilometers. It means that this Ad Hoc network is dense, indeed, the aircrafts have potentially a large number of Ad Hoc neighbors. But the aircrafts are only equipped with a bounded number of antennas and so an aircraft can only have an effective limited number of neighbors. These neighbors have to be appropriately chosen according to the distance between each aircrafts, their relative speeds and directions. ROSA endowed with the mobile density is especially adapted for this purpose.

The discovery of the potentially Ad Hoc neighbors of an aircraft is achieved with a non directional antenna and an existing protocol specific to the aeronautical network. Moreover the aircrafts are endowed with the ADS-B system (CASCADE Project Eurocontrol [2008]), that is a system that returns the state vector of an equipped aircraft.

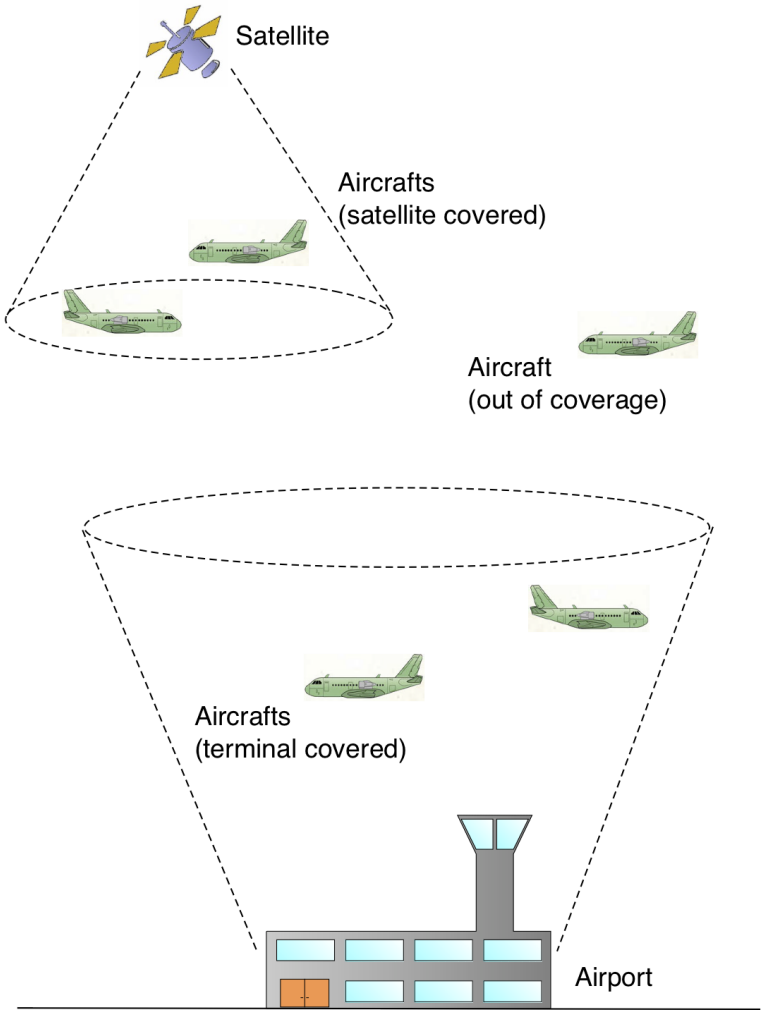


Figure 6.7: The aeronautical network with the three types of devices: airports (ground), aircrafts (mid altitude) and satellite (high altitude).

Chapter 7

Analysis

In this Chapter we analyse the performances of ROSA and the 'chain of lumps'. In the Section 7.1 we study the scalability of ROSA and show that the bandwidth used by a node does not depend on the number of the nodes involved in ROSA. In the Section 7.2 we focus on the efficiency of the routing algorithm of the routing over the 'chain of lumps'. Finally, in the Section 7.3 we deal with the parameters that impact the load of the nodes.

7.1 Scalability of ROSA

ROSA ensures its scalability by bounding the number of neighbors per node by a constant. This bound is equal to the maximum number of nodes per lump multiplied by the number of lumps per node.

The size of the messages exchanged for the inner functioning of ROSA is also bounded. The messages with the highest size bound are the messages *AbsorbLump*. A message *AbsorbLump* contains the representation of a lump, two identifiers of lumps and the identifier of the node that sends the message. The size of an identifier is equal to 128 bits (for the lumps and for the nodes). If we refer to the definition of the representation of a lump given in the Section 2.1, the bound on the size of the representation of the lump is equal to:

$$\begin{aligned} bound &= LumpSizeLimit \times (sizeof(node_{id}) + sizeof(node_{phy})) + sizeof(data_{density}) \\ bound &= 256\ bits \times LumpSizeLimit + sizeof(data_{density}) \end{aligned}$$

where *LumpSizeLimit* is the maximum number of nodes per lump. The value of the size of $data_{density}$ depends on the definition of the density. If the definition of the density used is equal to the default density (i.e. the number of nodes that compose a lump) then the bound on the size on the representation of a lump is equal to $8\ bytes \times LumpSizeLimit$. One can also notice that in order to ROSA be scalable, the definition of the density used over ROSA must allow to compute it with a bounded size of $data_{density}$. Indeed, if $data_{density}$ depends on the number of nodes, ROSA is not scalable anymore.

We have performed simulations in order to verify the scalability of ROSA. These simulations consisted in running many different instances of ROSA. Each of these instances of ROSA has a different number of nodes. Then we measure the average number of bytes that the nodes of ROSA send at each interval. The Figure 7.1 shows the result of one of these simulations.

For the simulation corresponding to the Figure, the maximum number of nodes per lump is set to 10, the maximum number of lumps per node is set to 2 and the density is the default density.

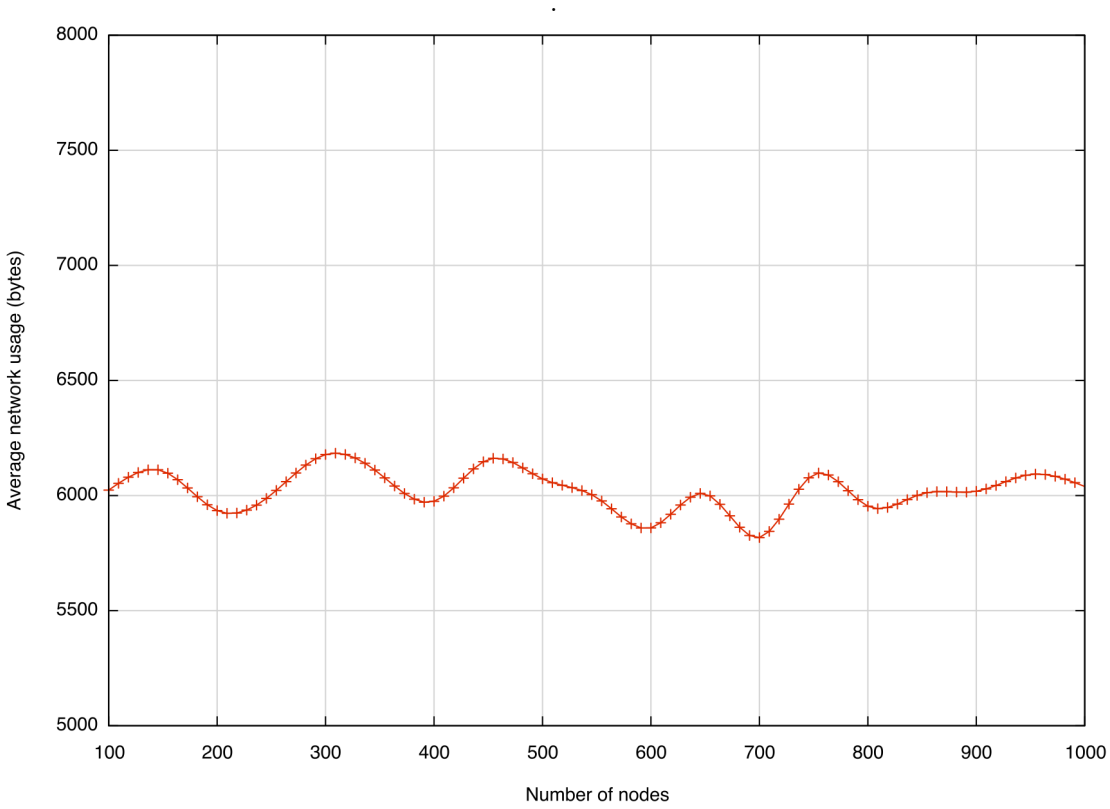


Figure 7.1: Average number of bytes sent by a node in function of the number of nodes

One can see in the Figure that the average size of the data sent by a node during a time interval does not depend on the number of nodes of ROSA. The other simulations performed show the same fact. One can consider the management of the topology of ROSA as a scalable process.

7.2 Efficiency of the routing over the 'chain of lumps'

In this section we show the performances of the routing algorithm based on the 'chain of lumps'. In the Section 7.2.1 we present a formal analysis of the efficiency of the routing algorithm in the worst case. We show that the routing algorithm is in $O(N)$ where N is the number of sub-intervals that are implied in the 'chain of lumps'. In the Section 7.2.2 we present the results of some simulations. These simulations aim to reflect the behavior of the routing algorithm in real cases. Consequently, the simulations aim to show the real efficiency of the routing algorithm based on the 'chain of lumps'. The efficiency of the routing algorithm is measured by the number of hops that a data packet must do when a random node wants to send this data packet to a random lump.

7.2.1 Formal analysis of the worst case

The 'chain of lumps' is composed of lumps that share a initial interval I_{init} into sub-intervals such:

- All the sub-intervals are allocated ;
- Two lumps that possess contiguous sub-intervals share at least a common node ;
- Each lump has at least one sub-interval.

Two lumps that share contiguous sub-intervals are called predecessors and successors accordingly to the position of their sub-intervals on the 'chain of lumps'. In order to be more efficient the last and first lumps of the chain are respectively predecessor and successor of each other. This way the lumps form a circular chain. There exists shortcuts in the chain since the node are allowed to compose more than one lump.

We call distance between a node n and a lump l, the minimum number of lumps that separate n from l in the 'chain of lumps'. The worst case for the efficiency of the 'chain of lumps' is the lack of shortcuts. In this worst case, the distance between a node and a lump is equal to the number of hops that a data packet sent by the node would do to reach the lump.

Let, now and until the end of this Section, us consider the worst case. In order to determine the efficiency of the routing algorithm, one has to determine the probability that a node is at a given distance from a lump. This distance is at most equal to $\lfloor N/2 \rfloor$. The probability that a randomly chosen node is at a distance d from a randomly chosen lump on a 'chain of lumps' without shortcuts and composed by N sub-intervals is noted $P_N(d)$. To determine these $P_N(d)$, we have to consider two cases, N is even or N is odd. If N is even, let $p = N/2$, we have :

$$P_N(d) = \begin{cases} 1/N & \text{if } d = 0 \\ 2/N & \text{if } 1 \leq d < p \\ 1/N & \text{if } d = p \end{cases}$$

If N is , let $p = (N-1)/2$, we have :

$$P_N(d) = \begin{cases} 1/N & \text{if } d = 0 \\ 2/N & \text{if } 1 \leq d \leq p \end{cases}$$

The efficiency of the routing algorithm is the number of hops that a data packet, sent by a randomly chosen node to a randomly chosen lump, will do. Since there is no shortcuts in the 'chain of lumps' in the worst case, the number of hops is equal to the distance between the randomly chosen node and the randomly chosen lump. This number of hops on a 'chain of lumps' composed by N sub-intervals is noted $Hop(N)$. This number of hops $Hop(N)$ is equal to the sum, for all the possible values that the distance between the node and the lump can takes, of the value multiplied the probabilities that the distance between the node and the lump equals this value:

$$Hop(N) = \sum_i P_n(i) \times i$$

Since the probabilities depend of the parity of N, we have to consider two cases: N is even or N is odd.
If N is even, let $p = N/2$, and :

$$\begin{aligned} Hop(N) &= \sum_{i}^{p-1} P_n(i) \times i + \frac{p}{N} \\ Hop(N) &= \sum_{i}^{p-1} \frac{2}{N} \times i + \frac{p}{N} \\ Hop(N) &= \frac{2}{N} \times \frac{p(p-1)}{2} + \frac{p}{N} \\ Hop(N) &= \frac{\frac{N}{2}(\frac{N}{2}-1) + \frac{N}{2}}{N} \\ Hop(N) &= \frac{N}{4} \end{aligned}$$

If N is odd, let $p = (N-1)/2$, and :

$$\begin{aligned} Hop(N) &= \sum_{i}^{p} P_n(i) \times i \\ Hop(N) &= \sum_{i}^{p} \frac{2}{N} \times i \\ Hop(N) &= \frac{2}{N} \times \frac{p(p+1)}{2} \\ Hop(N) &= \frac{\frac{N-1}{2}\frac{N+1}{2}}{N} \\ Hop(N) &= \frac{N}{4} - \frac{1}{4N} \end{aligned}$$

We can conclude that in the worst case, the number of hops that a data packet does between a random node to a random lump is less or equal to $N/4$ where N is the number of the sub-intervals that compose the 'chain of lumps'. The routing algorithm using the 'chain of lumps' is in $O(N)$.

To verify experimentally that the number of hops to route a data packet from random node to a random lump is less or equal to $N/4$, some simulations were performed. In these simulations, the number of nodes per lump is equal to 15 and the number of lumps per node is set to 1. Consequently, there is no shortcuts in the 'chain of lumps' and the simulations were performed in the worst case. Then we built several instances of ROSA with a 'chain of lumps' composed by more and more sub-intervals (these instances are obtained by adding more and more nodes to an initial instance of ROSA). Then a node and a lump are randomly chosen and the number of hops needed to route a data packet from the node to the lump is measured. The result of these simulations are shown in the

Figure 7.2. One can see that the theoretical number of hops corresponds to the number of hops obtained with the simulation.

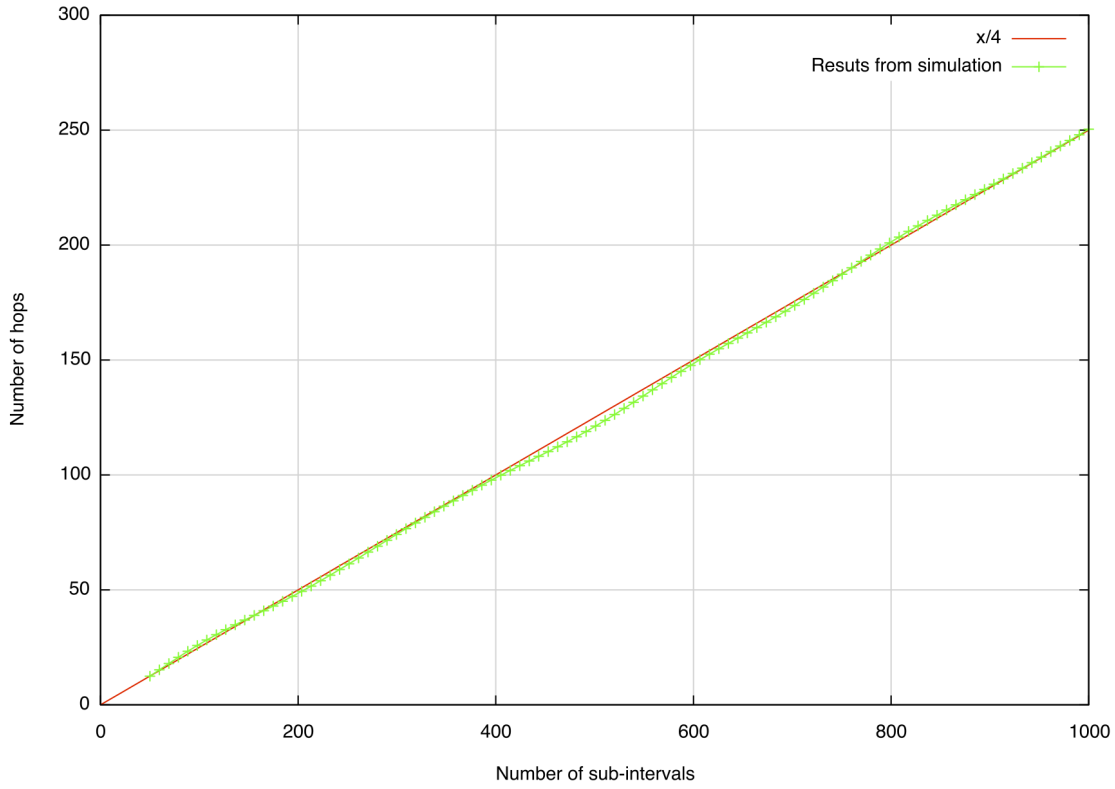


Figure 7.2: The average number of hops in function of the number of sub-intervals in the worst case

7.2.2 Simulation of real cases

In this Section we simulate the behavior of the 'chain of lumps' in conditions near to those encountered in the reality. We study the impact, on the efficiency of the routing algorithm of two parameters. These parameters are the number of sub-intervals that compose the 'chain of lumps' and the number of shortcuts present on the 'chain of lumps'. The influence of the number of sub-intervals is shown on the Section 7.2.2.1. The influence of the number of shortcuts is shown in the Section 7.2.2.2.

7.2.2.1 Influence of the number of sub-intervals

In order to understand how the number of sub-intervals influences the number of hops needed to route a message from a random node to a random lump we performed many simulations.

In the first simulation, the number of nodes per lump is equal to 20 and the number of shortcuts per node is equal to 0.5. It means that a node belongs to 1.5 lumps in average. This is due to the fact that some nodes are unable to join more that one lump according

to the network topology. Then we built several instances of ROSA with a 'chain of lumps' composed by more and more sub-intervals by adding more and more nodes to ROSA. To give an order of height, the number of nodes to obtain 500 sub-intervals is about 1300 nodes.

We measure the average number of hops needed to route a message from a node to a lump. The node and the lump are randomly chosen. The results of this simulation are shown in the Figure 7.3. On this graph is also represented the standard deviation corresponding to the average number of hops and the theoretical worst case routing bound.

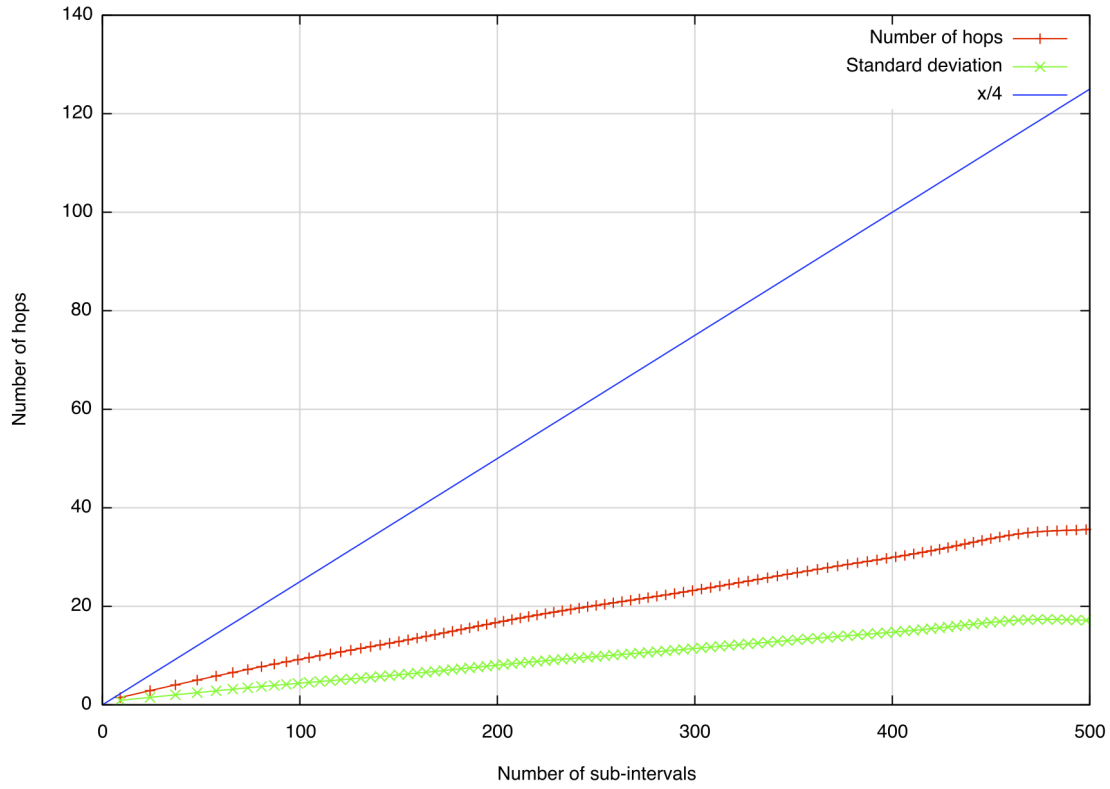


Figure 7.3: The average number of hops in function of the number of sub-intervals with 0.5 shortcuts per node

One can see in this Figure that the average number of hops needed to route a message is less than theoretical worst case bound and seems to linearly grows. One can also see that the standard deviation is important. The average number of hops needed to route messages in a 'chain of lumps' with 500 sub-intervals is about 37.

The parameters of the second simulation differ from those of the first one in the number of nodes par lump that is equal to 15 and in the number of shortcuts that is equal to 1.5 (the nodes belong to 2.5 lumps in average). The results of this simulation are shown in the graph of the Figure 7.4.

One can see in this Figure that the average number of hops needed does not grow linearly in function of the number of sub-intervals. It seems to be due to the number of shortcuts in the 'chain of lumps'. The average number of hops needed to route messages

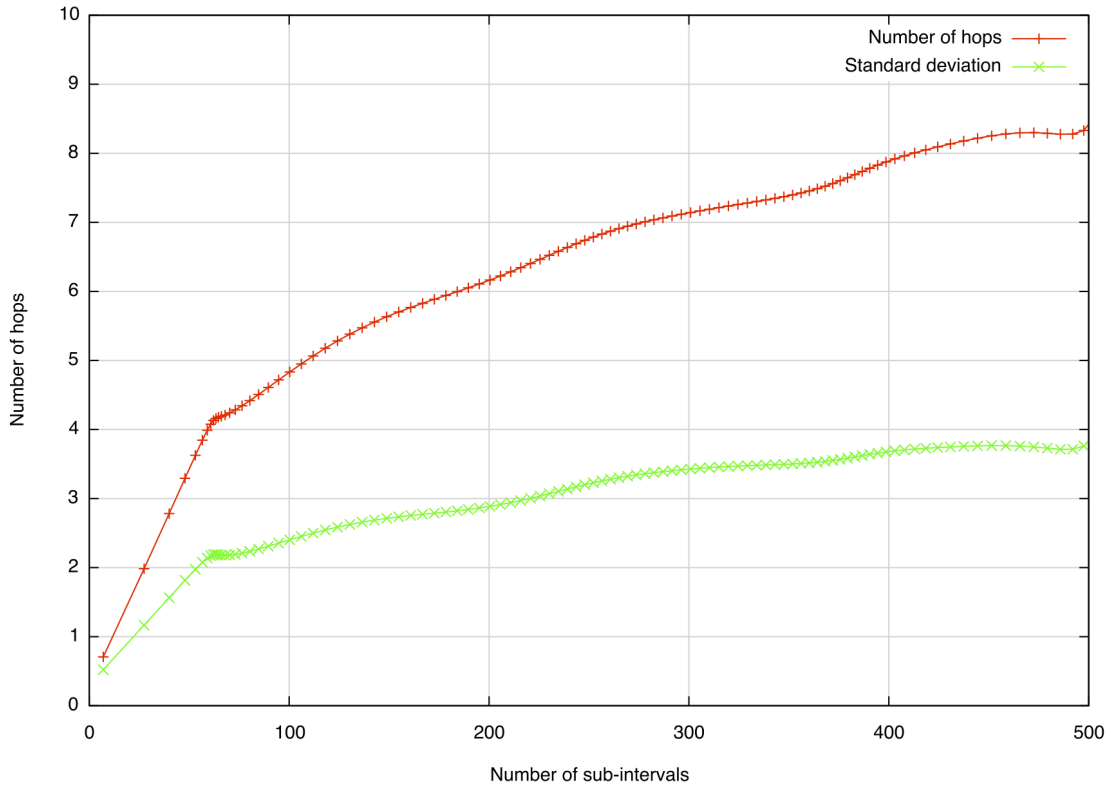


Figure 7.4: The average number of hops in function of the number of sub-intervals with 1.5 shortcuts per node

in a 'chain of lumps' with 500 sub-intervals is about 8.3.

The parameters of the third simulation are the same than those of the precedent simulations excepted for the number of nodes per lump and the number of shortcuts. In this simulation the number of nodes per lump is set to 10 and the number of shortcuts is set to 2.5 (the nodes belong to 3.5 lumps in average). The results of this simulation are shown in the graph of the Figure 7.5.

One can see in this Figure that the average number of hops needed first grows until 300 sub-intervals and then stagnates at a average number of hops about to 5.25 number of hops. This simulation shows that if the number of shortcuts is sufficient the number of sub-intervals in the 'chain of lumps' does not influence anymore the number of hops needed to route messages.

One can also remark that the maximum number of neighbors per node that is equal to the maximum number of nodes per lump multiplied by the maximum number of lumps per node is nearly the same for the three simulations ($1.5 \times 25 = 37.5$, $2.5 \times 15 = 37.5$ and $3.5 \times 10 = 35$). One can remark that this number of neighbors is small compared of the total number of nodes needed to reach 500 sub-intervals in the 'chain of lumps'.

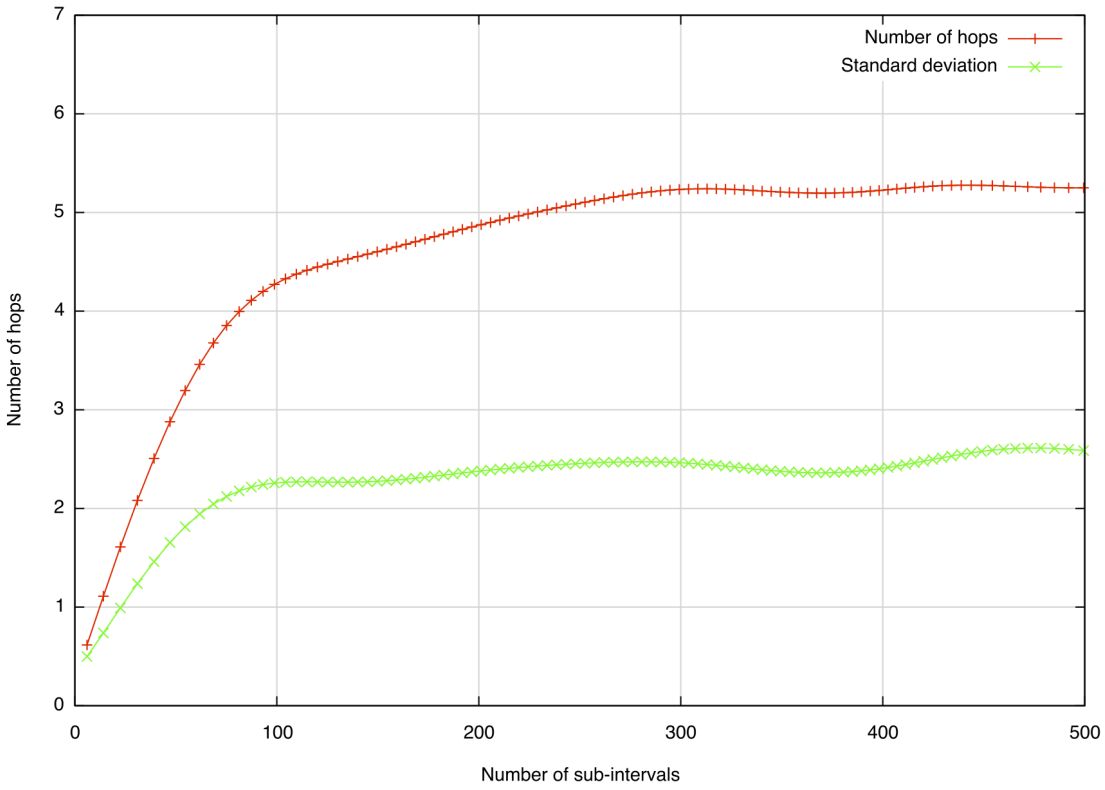


Figure 7.5: The average number of hops in function of the number of sub-intervals with 2.5 shortcuts per node

7.2.2.2 Influence of the number of shortcuts per node

Let us recall first that the number of shortcuts is a function of the number of lumps per node, but also depends of the topology of the network on which ROSA is deployed. Indeed, it is not sure that given a topology the nodes are able to join a number of lumps equals to the maximum number of nodes per lump allowed. In a general way, one can increase the number of shortcuts per node by increasing the number of lumps per node. Nevertheless, one cannot predict the number of shortcuts per node that it will obtain. Consequently the number of shortcuts per node has to be empirically determined. It is however possible to determine the average number of lumps per node given the number of shortcuts per node: the average number of lumps per node is equal to the number of shortcuts per lumps plus one.

One could think that in order to benefit of a really efficient routing protocol, it is sufficient to increase the number of shortcuts per node. Nevertheless, increasing the number of shortcuts per node also means increasing the number of the representation of lumps that a node must store. Consequently, the gain for the routing is obtained to the detriment of time that each node needs to execute the main loop of ROSA and to the detriment of the amount of data that each node must store.

In order to determine the ideal number of lumps per nodes we have performed many simulations to see the influence of the number of shortcuts on the number of hops needed

to route messages on the 'chain of lumps'. These simulation consists in running different instance of ROSA endowed with a 'chain of lumps' composed by the same number of sub-intervals but with different number of shortcuts for the 'chain of lumps' of the different instances of ROSA.

The results of this simulation with 500 sub-intervals in the 'chain of lumps' are shown in the graph of the Figure 7.6.

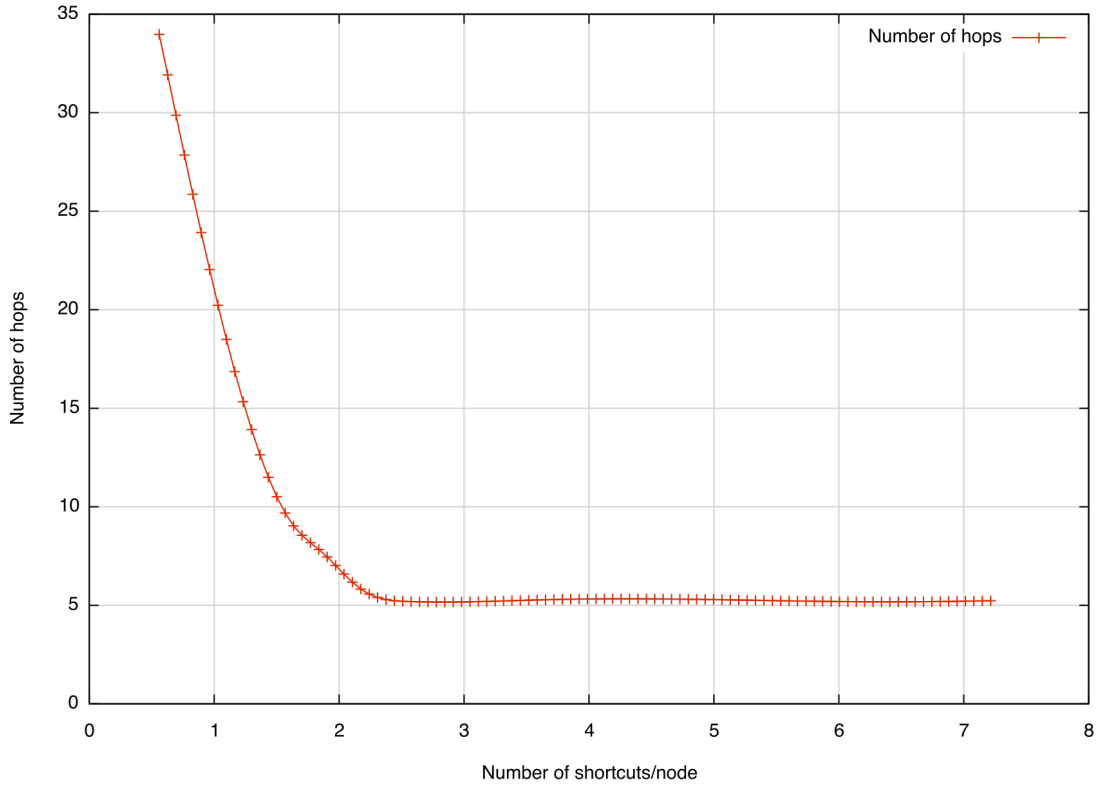


Figure 7.6: The average number of hops in function of the number of shortcuts per node

One can see that as the number of shortcuts in the 'chain of lumps' increases the average number hops exponentially decreases and finally stagnates around 5.25 hops. Consequently, having a number of shortcuts per node greater than 2.5 (i.e. number of 3.5 lumps per nodes) does not bring anymore gain in the routing. The simulations performed with other values for the number of sub-intervals gave nearly the same results.

The phenomenon appearing in the results of these simulations is similar to a small-world phenomenon. In a network under the influence of a small-world phenomenon, most of the nodes are not neighbors, but most nodes can reach every other nodes by a small number of hops. A network under the influence of a small-world phenomenon is called small world network.

The small-world networks are based on the six degrees of separation concept. This concept affirms that if a person is one step away from each person they know, and if a person is at two steps away from each person who is known by people that are at one step away from the person then each person on earth is at most six steps away from any other

person on Earth.

The concept of six degrees of separation were popularized, in the context of the computers network, first in 2001 by Duncan Watts. It performs a mail based experiments that consists of an email message that has to be delivered to 19 targets. The network used in this experiments consist in 48000 senders shared over 157 countries, each sender knowing only a small set of the other senders. Watts founds during this experiments that the number of intermediaries sender was around 6. This concept has been revised to the seven degrees of separation since a 2008 study (Leskovec and Horvitz [2008]).

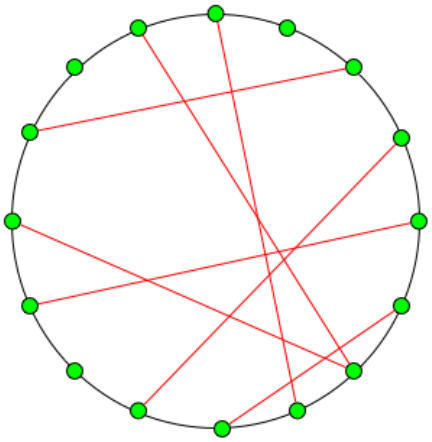


Figure 7.7: A Watts and Strogatz network

The 'chain of lumps' can be compared to the Watts and Strogatz small world model (Watts and Strogatz [1998]), that is a network where the nodes are organized to form a circle. Some random long links connect pair of nodes that are not neighbors on the circle. The Figure 7.7 represents the Watts and Strogatz small world network. The long links are represented in red in this Figure.

As for the Watts and Strogatz network our simulations show that the distribution of the shortcuts depends of the length of these shortcuts. There are many shortcuts with small length and only few with long length. In the Figure 7.8 are shown the results of these simulations. On the y-axis are represented the number of shortcuts in percentage in function of the x-axis where are represented the lengths of the shortcuts. There is two curves one corresponding to a simulation involving a 'chain of lump' with 500 sub-intervals and one corresponding to a simulation involving a 'chain of lumps' with 800 sub-intervals.

One can remark, on this Figure, that the number of sub-intervals that compose the 'chain of lumps' does not impact the distribution of the lengths of the shortcuts. This fact was verified by more of our simulations.

The next simulation aims to show the influence of the number of shortcuts per node of the distribution on the lengths of the shortcuts. On the Figure 7.9 three curves are represented. These curves correspond to three instances of ROSA endowed with a 'chain of lumps' composed of 500 sub-intervals with respectively 0.5, 2.5 and 5.5 shortcuts per nodes.

This Figure shows the fact of passing from a value of 0.5 to a value of 2.5 shortcuts per node changes the distribution of the lengths of the shortcuts. Indeed the higher the value

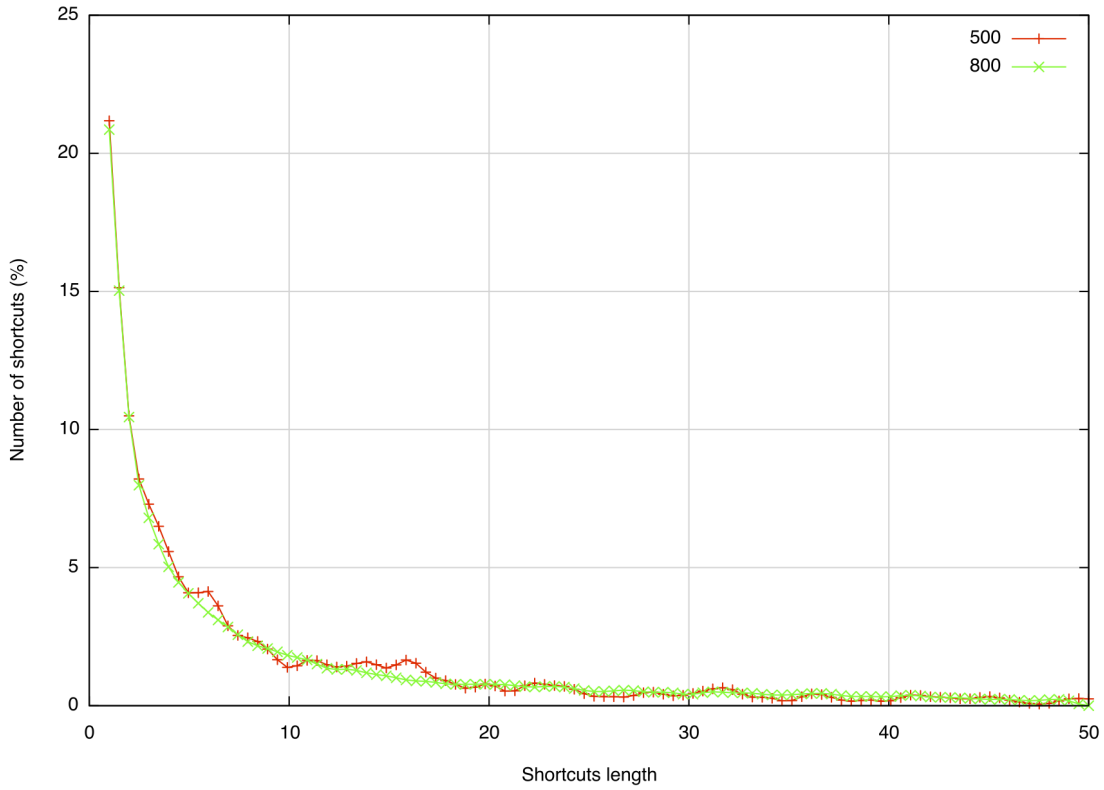


Figure 7.8: Distribution of the shortcuts according to the shortcuts length

of the shortcuts per node is, the higher the proportion of long shortcuts is. One can also see, on this Figure, that once the value of 2.5 shortcuts per node is reach, increasing the number of shortcuts per node does not impact a lot the distribution of the length of the shortcuts. It explain why, as shown in the Figure 7.6, increasing the number of shortcuts per nodes above a given value is not interesting from a routing point of view.

7.3 Load of a node

The load of a node in a ROSA network endowed with a 'chain of lumps' is the cumulative length of the sub-intervals handled by the lumps to which the node belongs. The load of the nodes is an important parameter. If we consider the case where the reliable file storage of ROSA is used, a node with a important load will have to theoretically handles and stores an important number of indexes and files. Consequently the load of the nodes must be as low as possible.

In this Section we study the load of the nodes according the number of sub-intervals. We performed few simulations. On these simulations we first study the impact of the variation of the number on the loads of the nodes. We also study the impact of the number of lumps per node on the load of the nodes.

The Figure 7.10 shows how the average load of the nodes evolved in function of the number of the sub-intervals that compose the 'chain of lumps'. On the x-axis are repre-

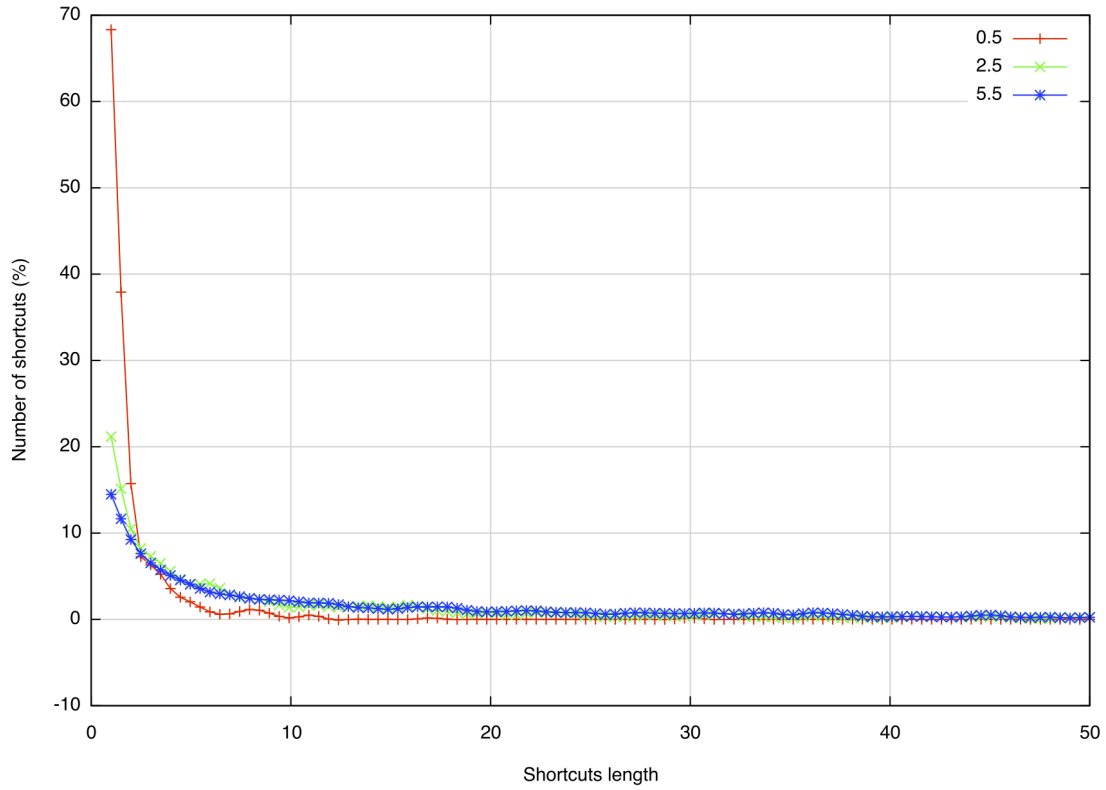


Figure 7.9: Distribution of the shortcuts according to the shortcuts length for three values of shortcuts per node

sented the number of sub-intervals and on the y-axis are represented the average load of the nodes in percentage of the length of the initial interval (I_{init}). One can see on this Figure the curves corresponding to the load of the nodes in function of the number of sub-intervals for three different number of lumps per node (1.5, 2.5 and 3.5).

Two facts were revealed by these simulations. The first one is that the average load of the nodes decreases as the number of sub-intervals in the 'chain of lumps' increase. It is an obvious fact since in the initialization of ROSA, there is a single lump and a single sub-interval, and therefore all the nodes of the first lump have an load equal to 100%. One can also remark that the higher the number of lumps per node is, the higher the average load of the nodes is.

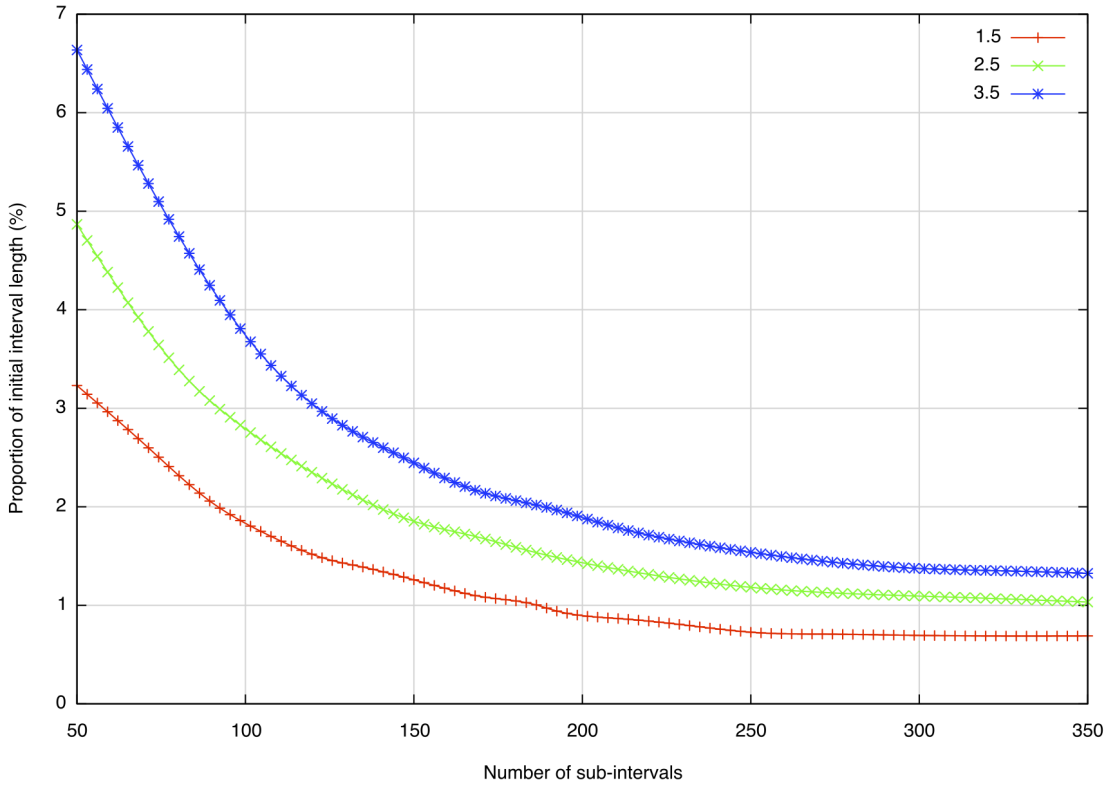


Figure 7.10: Average load of a node in function of the number of sub-intervals for three values of shortcuts per node

Chapter 8

ROSA in a real case

In this Chapter we present a distributed monitoring tool that uses ROSA as backbone. In the Section 8.1 we introduce the distributed monitoring tool. The monitoring tool is based on the computation of the Security Assurance of the different entities of the monitored network. The Section 8.2 explains how the Security Assurance of each kind of entities is computed. The Section 8.3 deals with an implementation and an experimentation of the monitoring tool over a real network.

8.1 Introduction

ROSA and the resilient density were used in within the context of the DESEREC project[1]. This project aims to provides methods and tools to monitor, analyze, design, model, simulate and plan the optimized configuration of Communications and Information Systems (CIS).

The DESEREC framework has to perform 3 tasks that are:

- Modelling. This task consists in planning and defining the optimal operational configuration of the CIS. This allow defining coherent and operational mode of the CIS. By the way, this allow the detection of the attacks or failures by comparison with the optimal operational mode of the CIS.
- Detection and Prevention. To obtain the current operational mode of the CIS, some sensors must allow to measure the characteristic of the CIS. The sensors must detect the forerunners of a failure or an attack.
- Reaction. Some computer-aided and automated counter-measures initiatives have to be taken when a failure or an attack is detected. These responses must be quick and appropriate to the kind of detected incidents.

The monitoring tool proposed within the context of DESEREC consists in a distributed set of multi-technology sensors and a set of detection mechanisms to detect attacks, failures or services bugs that can occur in the system. When such an incident is detected, the tool must respond in a quick and appropriate way according to a security policy. The security policy may imply system and services reconfiguration. The tool proposed during the DESEREC project is able to identify malicious events and to isolate the suspected entities to avoid the propagation of threats or a cascading effect.

[1] IST FP6 DESEREC Project (CN: 026600) of the European Union.

Since this tool is destined to be deployed on the network of a company and since such network could be shared over many countries and therefore over many autonomous systems and subnetworks we cannot tolerate that some failures prevent the sensors shared by these different subnetworks from communicating. The tool proposed by DESEREC, needs to have guarantees on the efficiency of the routing of its data. The sensors must communicate using a resilient network.

A set of sensors is installed on every measurable entities of the network. The sensors installed on a given entity depends on the type of this entity. The sensors may measure the load of the processor, the disk usage, the use of network interfaces, the output of anti-virus softwares, the errors in the services log files and so on.

The monitoring tool uses the sensors output data to compute a Security Assurance (SA) Values. Many kind of entities can compose a network, entities can be workstations, gateways, printers, routers, subnetworks and so on. Each kind of entities has its own definition of the SA value. The SA value of a entity can be computed locally or distantly depending on its kind. The SA value of a printer cannot be locally computed since a printer does not have computing capabilities. The SA value of some entities, such as subnetworks, has to be computed in a distributed way.

The monitoring tool must detect quickly the attacks, failures or services bugs, it implies that the SA values of the different entities must be permanently recomputed. The frequency of computation of the SA value of an entity depends on the kind of the entity. The SA values of critical entities or those of the entities with state that change quickly must be recomputed very often while the SA values of entities with low impact on the network, such as the printers, could be computed less often. The SA values of an entity is computed from data retrieved from the sensors. The data needed to compute the SA value of a given entity are also retrieved at different time interval according to the type of data. The Table 8.1 shows the different time intervals for some data that have to be retrieved to compute the SA value of a workstation.

The Figure 8.1 schematizes the monitoring tool and the network whose the tool is deployed. On the bottom of the figure, one can see the sensors as blue polygons, the entities as green circles and the Ethernet link between the entities. The sensors returns some values that are interpreted by the monitoring tool. The monitoring tool is represented on the top of the figure. One can see the lumps of ROSA (supporting the tool) as dashed purple shapes and the computed SA values as orange circles.

Once the SA values are computed the Security Policy has to be applied. Nowadays the Security Policy consists in two rules that are:

- If the SA value of a routing entity (router or a gateway) goes under a given threshold, then the monitoring tool tries to find an alternative routing entity for all the subnetworks that depends of the entity with a low SA value. If no alternative routing entity can be found, for a given subnetwork, the subnetwork is isolated until the SA value of the entity is above the threshold. If an alternative routing entity is found the routing table of the concerned entities is modified in order to use the alternative entity. This rule supposes that if the SA value of a routing entity is low, this routing entity may be under attacks and consequently an eavesdropper could inject malicious data or spy the communications that transit through the routing entity.
- If a SA value of a subnetwork goes under a given threshold this network is isolated.

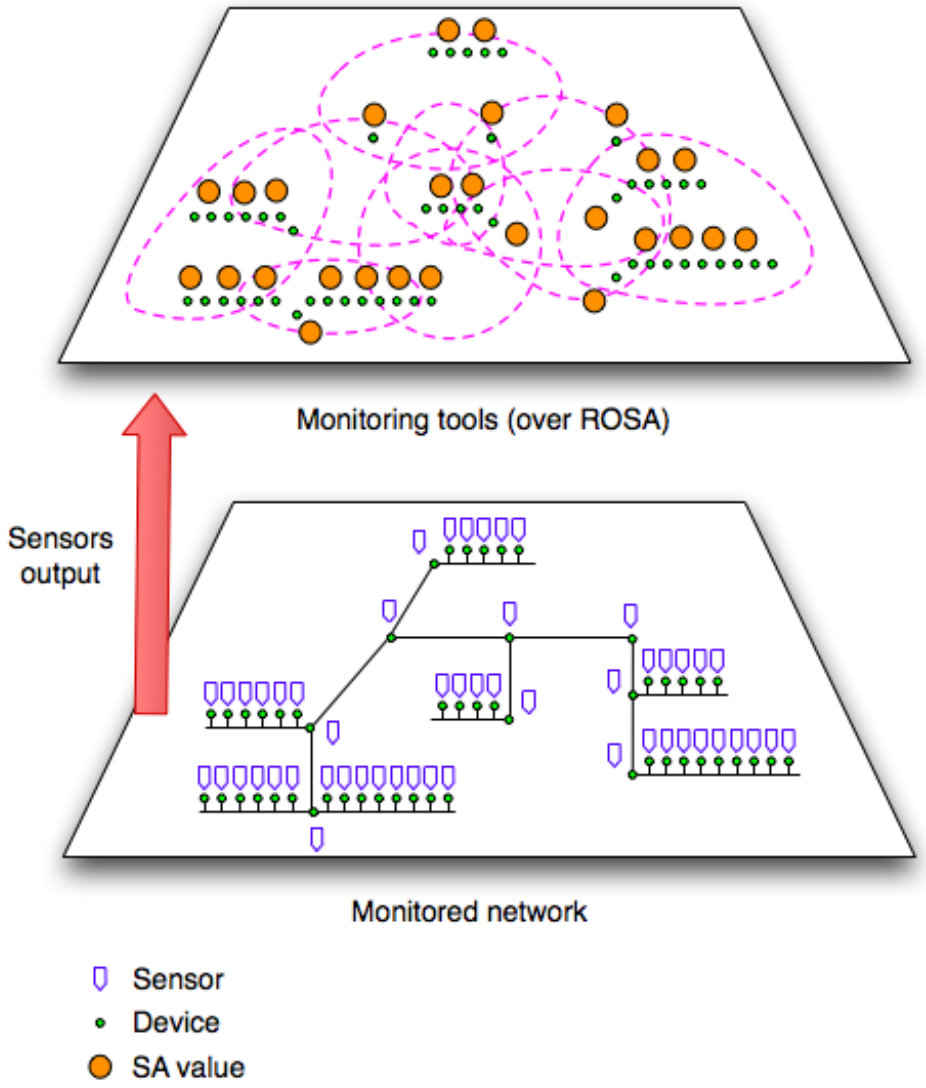


Figure 8.1: The DESEREC tool retrieving the sensors output and computing the SA values

It means that all the entity of the network must discard the data packet emanating from the subnetwork with the low SA value. This rule supposes that if the SA value of a subnetwork is low, the entities that compose the subnetwork may be victims of a virus. The isolation of the subnetwork mitigates the propagation of the virus. Once the virus is removed, the SA value of the subnetwork goes above the threshold and the subnetwork is not maintained in quarantine anymore.

The thresholds of the SA values below which the Security Policy requires a reaction have to be experimentally determined by calibrating the sensors of the monitored network during a period where no attacks or no failures are encountered. This period must be sufficiently long in order to obtain significant thresholds.

Many other rules could be added to the Security Policy according to the services running

Retrieved data	Frequency
Input packet rate	10 seconds
Input packet error rate	10 seconds
Output packet rate	10 seconds
Output packet error rate	10 seconds
Packet collision rate	10 seconds
Number of UDP ports	30 seconds
Number of datagrams discarded because no route could be found rate	30 seconds
Number of received UDP datagrams for which there was no applications at the destination port rate	30 seconds
Number of TCP connections rate	1 minutes

Table 8.1: The retrieved data and their corresponding frequencies for a workstation

on the monitored network. Let us imagine that a web server is running on the monitored network and that alternative web servers exist. If the SA value of the main web server decreases too much, one of the alternative web server is activated by the monitoring tool while the one with the low SA value is deactivated. One can imagine as many rules that one wants.

The Figure 8.2 illustrates the applications of the rules of the Security Policy according to the SA value computed by the monitoring tool. One can see that that is the monitoring tool that is in charge to propagation the reconfiguration commands.

ROSA and the resilient density are used as the backbone of this tool. Its self-healing resilient topology and its reliable routing bring a good insurance that the different devices of the monitoring tool are able to communicate despite the failures, the attack and since the monitoring tool may modify the routing tables of the entities of the monitored network, despite the decisions taken by the monitoring tool itself.

8.2 Computing the SA values of the entities of a network

The computation of the SA values of the different entities of the monitored network is not in the scope of this thesis. This part of the work was done by Nguyen Pham at Telecom Paristech. A short summary of this work is presented in this Section. This summary is based on Pham and Riguidel [2007] and Pham et al. [2008].

8.2.1 Definition

The direct measurement of SA value of a network entity is desirable, but not always directly possible. In some cases, the network entity has to be decomposed into network

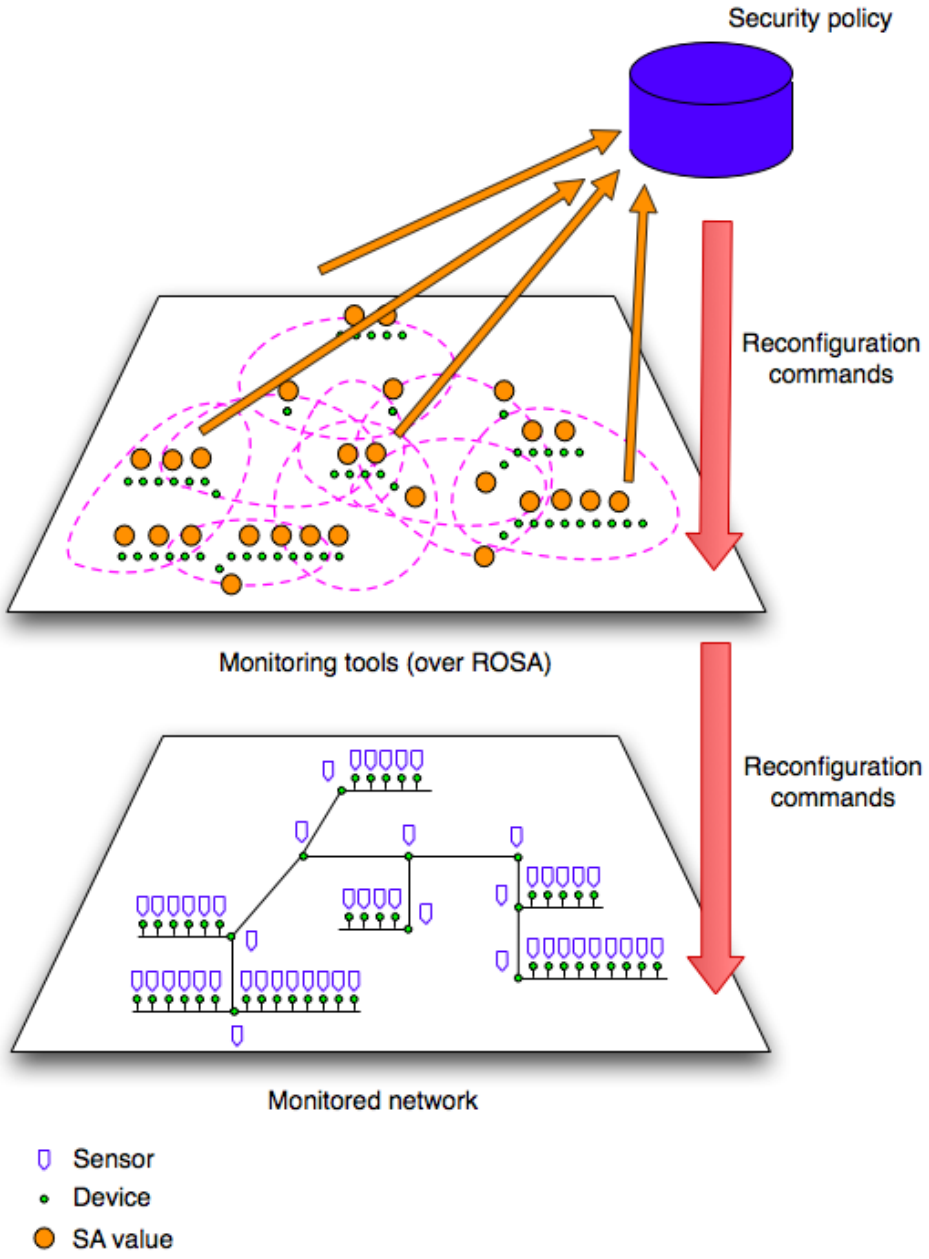


Figure 8.2: The DESEREC tool interpreting the SA values and applying the Security Policy

subentities. Then the SA value of the network entity is computed from the SA values of the subentites obtained by the decomposition of the initial entity using aggregation methods. An aggregation method combines the SA values of the subentities by taking account of the relations between these subentities to obtain the desired SA value. Consequently, the computation of the SA value of an entire network consists in 5 steps that are:

- Modelling. It consists in decomposing the network into irreducible entities. An entity is irreducible if it is possible to compute its SA value without needing the aggregation of the SA values of other entities.

- Metrics assignment. It consists in determining which metrics has to be observed to compute the SA values of each kind of entities.

- Measurement. It consists in measuring the SA values of the irreducible entities.

- Aggregation. It consists in computing the SA values of non irreducible entities using the aggregation methods and the already computed SA values.

- Evaluation and Interpretation. It consists in computing the SA values of the entire network and interpreting the resulting values to determine the overall assurance level of the network.

8.2.2 Evaluating the SA value of the network components

In this section we present how the SA value of each kind of entity is evaluated. Within the context of the monitoring tool that we propose, we define the SA values for 5 kinds of network entities that are defined as below:

- Workstation. A workstation is a microcomputer designed for technical or scientific applications. It is intended to be used by one person at a time.

- Server. A server is a computer designed to provide services to clients.

- Router. A router is a computer that connects two or more computers or other electronic devices to each other.

- Subnetwork. A subnetwork is a set of computers that have a common, designated IP address routing prefix.

- Network. A network is the reunion of all those network entities.

The SA value of a network entity is calculated through a combination of its local processing capability (e.g. CPU utilization, allocated memory), its network capability (e.g. number of established connections, number of packets received). Since the failure of the local processing capability or the network capability will result in the inability of the entity to carry out its functionalities, we take the min operator to represent this relation, therefore the general formula for the SA values (SAV) of an entity is:

$$SAV_{entity} = \min(SAV_{local}, SAV_{network})$$

The data needed to evaluate the SAVs are collected using SNMP or some local shell scripts. The data collected have to be normalized before that can be used to compute SA values.

8.2.2.1 Workstation

Each workstation computes its own SAV according to the formula:

$$\begin{aligned} SAV_{local} \quad = \quad & \text{min(CPU Utilization, Used disk space, Allocated memory)} \times \\ & \text{min(Number of process, Number of users)} \end{aligned}$$

and the formula:

$$\begin{aligned} SAV_{network} = \text{min(} & \text{Number of input packets, Input packet error rate,} \\ & \text{Number of output packets, Output packet error rate,} \\ & \text{Packet collision rate, Number of possible routes,} \\ & \text{Number of links to server)} \times \\ \text{min(} & \text{Number of TCP connections, Number of UDP connections,} \\ & \text{Number of discarded datagrams, Number of UDP datagrams)} \end{aligned}$$

8.2.2.2 Server

The SAV of a server is computed in the same way that the SAV of a workstation. However, in the case of a server that can have many IP addresses, the $SAV_{network}$ is calculated with the following formula:

$$SAV_{network} = SAV_{tmp} \times (1 \text{ / Number of IP addresses})$$

where SAV_{tmp} is computed with the formula used to compute the $SAV_{network}$ of a workstation. The SAV of a server is locally computed by the server itself.

8.2.2.3 Router

The SAV of a router are distantly computed. Since each entity of the network is susceptible to fail or be attacked at every moment, it is not judicious to confide the evaluation of the SAV of the routers to a dedicated workstation or server. We have chosen to use the following solution: each workstation and server, at each given interval of time, is open to compute the value of a router with a probability of 1 / (Number of stations + Number of servers). This assures, in a probabilistic way, that the SAVs of a routers is computed at each interval of time.

The formula used to compute the SAV_{loceal} of a router is:

$$SAV_{local} = \text{min(CPU utilization, Allocated memory)}$$

and the formula used to compute $SAV_{network}$ of a router is:

$$\begin{aligned} SAV_{network} \quad = \quad & \text{min(Input packet, Input packet error rate,} \\ & \text{Output packet rate, Output packet error rate,} \\ & \text{Packet collision rate, Number of network interface)} \times \\ & (1 \text{ / Number of IP addresses)} \end{aligned}$$

8.2.3 Evaluating the SA value of the sub networks

As for the routers and for the same reasons each workstation and server at each given interval of time, is open to compute the value of a sub network with a probability of 1 / (Number of stations + Number of servers). The SAV of a sub network is computed according to the formula:

$$\begin{aligned} SAV_{subnet} = \min(&\text{Average workstations,} \\ &\text{Average perimeter servers and router,} \\ &SAV_{network}) \end{aligned}$$

where "Average stations" is the average value of SA values of all workstations, "Average perimeter servers and router" is the average value of SA values of all the servers and routers belonging to the subnetwork.

The $SAV_{network}$ is computed as it follows:

$$\begin{aligned} SAV_{network} = \min(&\text{Input packet rate, Input packet error rate,} \\ &\text{Output packet rate, Output packet error rate,} \\ &\text{Packet collision rate)} \end{aligned}$$

In order to compute the SAV of a sub network, a workstation (or a server) needs to have the SA values of all the workstations and servers that belong to the subnetwork.

8.2.4 Evaluating the SA value of the whole network

To decide the SA value of the whole network, we apply the weighted-sum operator to the values of subnetworks that compose it. Therefore, the SAV of the whole is computed according to the formula:

$$SAV_{network} = \sum_{i=1}^{5} w_i SAV_i$$

where

$$w_i = \frac{\text{Number of machines in subnetwork i}}{\text{Total number of machines in the whole network}}$$

The value SAV_i is the SAV of the subnetwork i.

8.3 Experimentation on the Telecom ParisTech network

The monitoring tool has been implemented and deployed on the Computer Sciences and Networking department network at TELECOM ParisTech. This network has around 50 workstations, several CISCO routers and is divided into several subnetworks.

In this experimentation, we have used true sensors except for the Virus Sensor since we do not want to really propagate virus or worms on this operational network. We are able to simulate the existence of a worm attack or the failure of a network element. Concerning the Security Policy, only the two rules already mentioned in the above sections have been implemented. The application of Security Policy Rules, that is the reconfiguration commands, have been virtually implemented since, once again, we did not want to interfere with the department operational network.

Each workstation and servers of the monitored network acts as a node of ROSA and a node of the monitoring tool. All the communications between the nodes of the monitoring tool transit through ROSA. Each node of ROSA computes its SA value according to its type and the formulas presented in the Section 8.2.

When a node wants to compute the SA value of the subinterval, it broadcasts a request to retrieve the SA value of all the workstations and servers of the subnetwork. These workstations and servers reply to the requesting node with their SA values. Once the node that wants to compute the SA value has received the SA values of the devices of the subnetwork, it computes the SA value of the sub network and broadcast it to all the nodes of the monitoring tool. This way if the SA value of the sub network is under the threshold, each node of the monitoring tool can apply the appropriate rule of the Security Policy.

When a node wants to compute the SA value of a router, it retrieves the needed values using the SNMP protocol. This protocol exposes management data in the form of variables which describe the device configuration. These variables can then be queried. A SNMP daemon runs on each router of the monitored network. Once the necessary values are in possession of the node, it computes the SA value of the router. This SA value is broadcasted to the devices of the subnetwork that use this router. If the SA value of the router is low, the nodes of the monitoring tool try to find an alternate router and modify the routing table of the devices according to the Security Policy.

When a node wants to compute the SA value of the whole network it simply applies the formula of the Section 8.2 since it already possesses the SA values of all the subnetworks.

The result is that the system is able to quickly react to failures and attacks. The CPU usage is negligible on each computing device. The network usage can also be considered as negligible. While testing the system, we detected unpredicted events such as the system engineer reconfiguring the servers. From this point of view, the experimentation was a success.

In order to complete the monitoring tool and allow some demonstrations a control tool has been implemented. This control tool is called the security cockpit. The security cockpit consists in a java applet that display a representation of the monitored network. This applet is running on a server of the network. On this applet are also displayed the virtual links of ROSA and the SA values of the different network entities. The Figure 8.3 is a screenshot of the security cockpit. One can see on this Figure the workstations as green squares, the servers as purple squares and the routers as oranges squares. The bottom red squares corresponds to the different subnetworks of the monitored network.

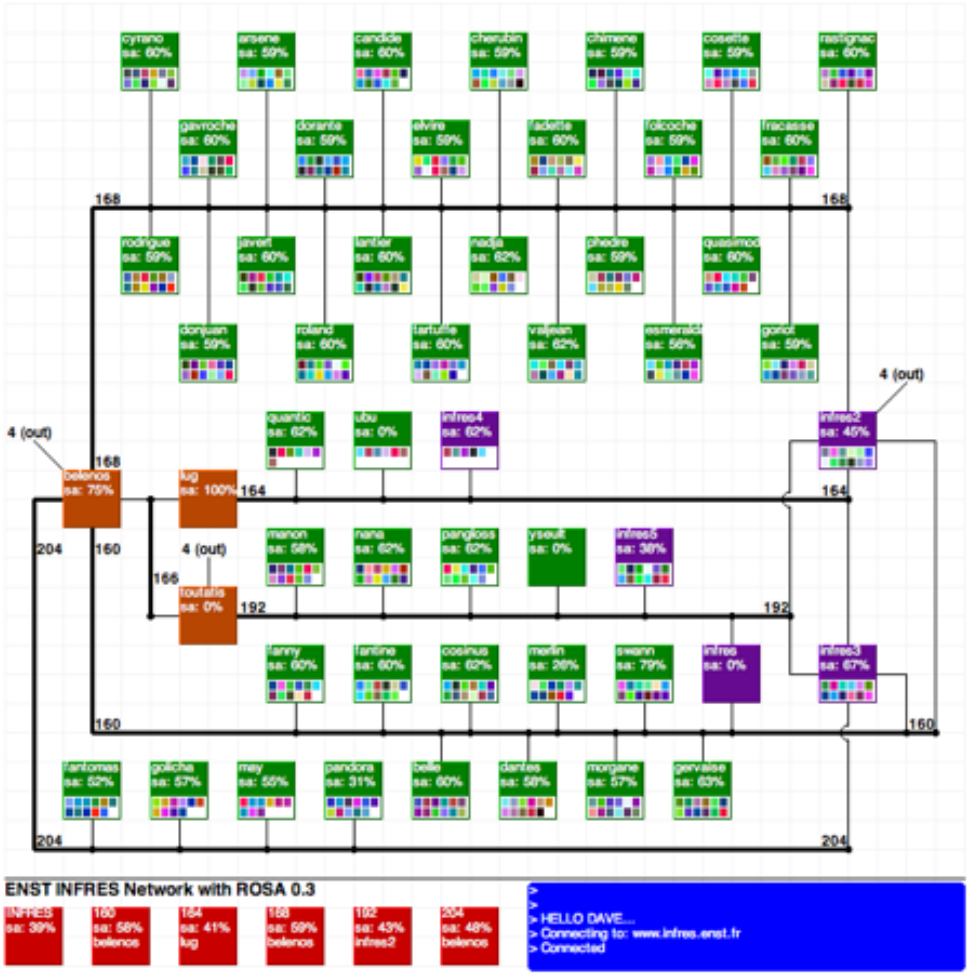


Figure 8.3: The screenshot of the DESEREC tool cockpit control

Conclusion

Original contribution

The PhD research done bring some contributions in two fields. These fields are the overlay networks and the Distributed HashTable. This PhD thesis also proposed two services that can be deployed over ROSA. These services take advantage of the properties of ROSA.

In the field of the overlay network this PhD thesis proposes a new overlay network called ROSA. ROSA oversteps the view of the overlay network as graph with nodes and links by using an abstraction layer: in ROSA, the nodes are organized into cluster called lumps. A lump is a set of fully connected nodes and ROSA can be represented by a entanglement of lumps. ROSA is a self-organizing, parametrizable, self-healing and scalable overlay network.

Self-organizing
Each node of ROSA belongs to at least one of the lumps. Each of the lumps of ROSA is associated with a metric called density. In ROSA, the nodes calculate the density of lumps which they belong, share with their neighbors about the knowledge of the low densities lumps, and leave lumps having high densities to increase the lumps with low densities. The density of the overall network is homogenized and maximized.

Parametrizable
The nodes homogenizes and maximizes the density of the lumps of ROSA. Consequently, one can change the behavior of ROSA by simply changing the definition of the density. In this thesis we proposed three kinds of density, the default density, the resilient density and the mobile density. Each of these densities has a purpose and an application field. The default density maximizes the number of nodes per lump and can be applied to all the networks. The resilient density id designed to be applied on IP networks and make ROSA resilient to the underlying network failures. The mobile density maintains and optimizes the connectivity between the nodes of ROSA in large and dense Ad Hoc networks.

Self-healing
ROSA possesses a process that detects the failure of the nodes. When a node detects a failure the entanglement of lumps is modified in order to reflect this failure and the nodes that was neighbors with the failing node may choose another node as neighbors. This

modification of the set of neighbors of the nodes in done during the optimization of the densities of the lumps.

Scalability
The number of neighbors per node is bounded by a constant. Consequently, the deployment, the self-organization and of the self-healing of ROSA do not depend either on the number of nodes that participates to ROSA. The scalability of ROSA with regard to the Bachmann-Landau notation is in $O(1)$. Simulations have demonstrated this result. All these properties of ROSA designs it as a good candidate to be used as the virtual layer in the virtualization of network model proposed in the introduction.

In the field of the DHT this PhD thesis proposes the 'chain of lumps'. The 'chain of lumps' is a DHT built over ROSA. This 'chain of lumps' must satisfy some conditions: the 'chain of lumps' must not reduce the scalability of ROSA and must not modify the topology of ROSA. The 'chain of lumps' is built over the entanglement of lumps. The lumps are organized in order to form a chain, this chain is complemented with shortcuts since the nodes may belong to more than one lump.

Scalability
In order to let ROSA as scalable that it is without the 'chain of lumps' (i.e. $O(1)$), the building and the maintenance of the 'chain of lumps' is achieved by simply adding some additional data in the message already exchanged by the nodes of ROSA. These data have a bounded size. The 'chain of lumps' allows any nodes of the network to send data packet to any lump of the 'chain of lumps'. This process is also scalable. Despite the fact that the theoretical average number of hops needed to route data packets is in $O(N)$ where N is the number of lumps that composes the 'chain of lumps', the simulations have shown that if the number of shortcuts per node is sufficient (around 2.3 shortcuts per nodes), the number of hops effectively needed to route data packets is in $O(1)$. This is due to a small world phenomenon.

Passive building
The 'chain of lumps' can be opposed to the other existing DHTs in the way of how it is built. Usually the nodes participating to a DHT choose their neighbors in order to make the topology of the network adequate for the DHT. The DHT is actively built to obtain the desired topology. In ROSA the set of the neighbors of each node is chosen in order to maximize the densities of the lumps. Consequently, the 'chain of lumps' is passively built despite the topology of ROSA.

Reliability
The 'chain of lumps' is a reliable DHT. Since any pair of <key, value> stored over the 'chain of lumps' is stored by all the nodes of the lumps responsible for the key. The nodes that joins a lump will have to store the <key, value> pairs that the lump stores. This implies that unless a great number of simultaneous failures occurs, the 'chain of lumps' still will be consistent. These properties of the 'chain of lumps' fulfills the conditions mentioned above.

This PhD thesis proposed two services built over ROSA endowed with the 'chain of lumps'. A resilient routing service and a reliable file storage service. These services take advantage of the properties of ROSA and of the 'chain of lumps'.

The resilient routing service
The 'chain of lumps' allows any node of ROSA to send data packet to a given lump. The resilient routing service allows any node to communicate directly with any other node of ROSA. This services is based on a set of routing tables used to locate the nodes. Each node maintains its routing table that is stored in a resilient way over the 'chain of lumps'. This routing service benefits of the reliability of the 'chain of lumps'.

The reliable storage service
This service allows node to store files over ROSA and the 'chain of lumps'. In order to ensure a reliable storage the files are stored by many nodes of ROSA. An index is stored on the 'chain of lumps' in order to locate the multiples replicas of a file stored on ROSA. This storage service benefits of the reliability of the 'chain of lumps'

The PhD research led to the publication of four articles:

- L. Baud, N. Pham, and P. Bellot. Robust Overlay Network with Self-Adaptive Topology: Protocol Description. In *2008 IEEE International Conference on Research, Innovation and Vision for the Future (RIVF 2008)*, Vietnam, Ho Chi Minh City, July 2008. Baud et al. [2008]

- L. Baud. Robust overlay network with self-adaptive topology: The reliable file storage layer. In *The 2009 IEEE - RIVF International Conference on Computing and Communication Technologies (RIVF 2009)*, Da Nang, Viet Nam, July 2009. Baud [2009]

- L. Baud and P. Bellot. Robust overlay network with self-adaptive topology: The chain of lumps structure. In *2009 International Workshop on Peer-To-Peer Networking (P2pNet 2009)*, St. Pertersburg, July 2009. Baud and Bellot [2009b]

- L. Baud and P. Bellot. The ROSA protocol adapted to aeronautical mobile ad-hoc network. In *8th Innovative Research Workshop & Exhibition (INO 2009)*, Brétigny sur Orge, France, Dec. 2009. Baud and Bellot [2009a]

Perspectives

Concerning a global point of view, the research perspectives that income from the works done during this thesis are the realization of the of the model described in the introduction of this thesis. It implies finding a definition of the density adapted to non dense Ad Hoc network and one adapted to the GSM/GPRS networks. It also implies finding a way to produce mixed densities for the lumps composed by nodes corresponding to devices

from different physical networks. The implementation of ROSA for each kind of physical networks has also to be considered in order to use ROSA as the virtual layer of the virtualization network model.

Concerning the Distributed HashTable point of view, the research perspectives that income from the works done during this thesis are the study and the creation of the passively built DHT. Indeed, the 'chain of lumps' has shown that it is possible to passively built DHTs. A passive built of a DHT consists in not modifying the neighbors sets of the nodes and simply maintaining the consistency DHT by adding some additional data to the message already exchanged by the nodes. The authors of the most known DHTs (Pastry, Chord, CAN, Kademlia, etc.) have not considered this possiblity and in the overlay network literature DHTs built in a passive way cannot be found.

Concerning the ROSA point of view, the research perspectives that income from the works done during this thesis are the development of new services over ROSA. Some seriously envisaged services are a fully distributed computing service over ROSA and a anonymous web-like service. The fully distributed computing service could perform a calculus as follow: the calculus is 'hashed' to obtain its identifier, the lumps handling the identifier of the calculus is in charge to distribute the tasks to the other lumps of ROSA. Each node of a lump that has to calculate a part of the initial calculus, performs the partial calculus and return the result to the lump that distributes the tasks. The lump that distributes the tasks would receive many partial results for the same part of the initial calculus. It would ensure that a malicious node cannot distort the calculus. The anonymous web-like service could be based on a similar system that Tor (Dingledine et al. [2004]) uses. Circuit of lumps could be created to hide the identity of the nodes publishing or consulting a content over ROSA.

Appendix

Node.initROSA

```
1: lump ← new lump ;
2: lump.nodeId_list ← id ;
3: lump.nodePhy_list ← phy ;
4: lump_list ← lump ;
5: connected ← true ;
```

Figure 4: The initROSA function

Node.getWeakestLump (**Output:***lump*)

```
1: result ← null ;
2: for all lump ∈ lump_list do
3:     if lump = null or lump.density() < result.density() then
4:         result ← lump ;
5: return result ;
```

Figure 5: The getWeakestLump() function

Node.getSMD(lump,id) (**Input:***lump*, *integer*; **Output:***float*)

```
1: lump1 ← lump.without(node.id) ;
2: lump2 ← lump.without(id) ;
3: return min(lump1.density(), lump2.density()) ;
```

Figure 6: The getSMD(lump,id) function

Node.checkLinks(lump) (**Input:***lump*; **Output:***integer*)

```
1: result ← 0 ;
2: for all phy ∈ nodePhy_list do
3:     if ping(phy) then
4:         result ← result + 1 ;
5: return result ;
```

Figure 7: The checkLinks(lump) function

Lump.without(nodeId) (**Input:***integer*; **Output:***lump*)

```
1: result ← Lump() ;
2: result.id ← rand(0, 2^128 − 1) ;
3: for 0 ≥ i < nodeId_list.length do
4:     if nodeId_list[i] ≠ nodeId then
           result.nodeId_list ← result.nodeId_list ∪ nodeId_list[i] ;
           result.nodePhy_list ← result.nodePhy_list ∪ nodePhy_list[i] ;
           /*
           adding the data concerning the node to result.data_density
           */
       result.subInt_list ← subInt_list ;
5: return result ;
```

Figure 8: The without(lump) function

Lump.contiguous(subInt$_1$, subInt$_2$)
(**Input:***subinterval*, *subinterval*; **Output:***boolean*)

```
1: if subInt_1.infBound = subInt_2.supBound or
      subInt_1.supBound = subInt_2.infBound
   then
2:     return true ;
3: return false ;
```

Figure 9: The contiguous(subInt$_1$, subInt$_2$) function

Lump.merge(subInt$_1$, subInt$_2$)
(**Input:***subinterval*, *subinterval*; **Output:***subinterval*)

```
1: if subInt1.infBound > subInt2.infBound then
2:     return merge(subInt2, subInt1) ;
3: if subInt1.upBound ≠ subInt2.lowBound then
4:     return null ;
5: result ← Subinterval() ;
6: result.lowBound ← subInt1.lowBound ;
7: result.upBound ← subInt2.upBound ;
8: result.predId_list ← subInt1.predId_list ;
9: result.succId_list ← subInt2.succId_list ;
10: return result ;
```

Figure 10: The merge(subInt$_1$, subInt$_2$) function

Node.getRMDJ(lump) (**Input:***lump*; **Output:***float*)

```
1: hyp_lump ← joinLump(lump) ;
2: if hyp_lump.size() < LumpSizeLimit then
3:     return hyp_lump.density ;
   else
4:     split_hyp_lump ← splitLump(hyp_lump) ;
5:     return min(split_hyp_lump[0].density, split_hyp_lump[1].density) ;
```

Figure 11: The getRMDJ(lump) function

Lump.canHandle(subint) (**Input:***subinterval*; **Output:***boolean*)

```
   pred ← false ;
   succ ← false ;
1: for all id ∈ nodeId_list do
2:     if id ∈ subint.predId then pred ← true ;
3:     if id ∈ subint.succId then succ ← true ;
4: return pred AND succ ;
```

Figure 12: The canHandle(subint) function

Interval.splitOver(lump$_1$, lump$_2$) (**Input:***lump*, *lump*);

```
1: subint_pair ← subint.split() ;
2: if lump1.canHandle(subint_pair[0]) then
3:     lump1.subInt_list ← lump1.subInt_list ∪ subint_pair[0] ;
4:     lump2.subInt_list ← lump2.subInt_list ∪ subint_pair[1] ;
   else
5:     lump1.subInt_list ← lump1.subInt_list ∪ subint_pair[1] ;
6:     lump2.subInt_list ← lump2.subInt_list ∪ subint_pair[0] ;
```

Figure 13: The splitOver(lump, lump) function

Interval.canSplitOver(lump$_1$, lump$_2$) (**Input:***lump*, *lump*);

```
1: subint_pair ← subint.split() ;
2: return (lump1.canHandle(subint_pair[0]) AND
           lump2.canHandle(subint_pair[1])       ) OR
          (lump1.canHandle(subint_pair[1]) AND
           lump2.canHandle(subint_pair[2])       ) ;
```

Figure 14: The canSplitOver(lump, lump) function

Node.checkPredList(lump$_I$, lump$_{I+}$, subInt_pair)
(**Input:***lump*, *lump*, *<subinterval, subinterval>*; **Output:***boolean*)

```
1: if lump_I.nodeId_list ≠ subint_pair[0] then
2:     return false ;
3: return true ;
```

Figure 15: The checkPredList(lump, lump, <subint, subint>) function

Node.checkSuccList(lump$_I$, lump$_{I+}$, subInt_pair)
(**Input:***lump*, *lump*, *<subinterval, subinterval>*; **Output:***boolean*)

```
1: if lump_I+.nodeId_list ≠ subint_pair[1] then
2:     return false ;
3: return true ;
```

Figure 16: The checkSuccList(lump, lump, <subint, subint>) function

Lump.getCSL (**Output:***boolean*)

```
1: result ← 0 ;
2: for all subint ∈ subInt_list do
3:     result ← result + subint.getLength() ;
4: return result ;
```

Figure 17: The getCSL() function

Lump.getKeyIn (**Input:***node_location*;**Output:***integer*)

```
1: for all subint ∈ node_location do
2:     return getIntegerBetween(subint.lowBound, subint.upBound) ;
```

Figure 18: The getKeyIn() function

Bibliography

K. Aberer. P-grid: A self-organizing access structure for p2p information systems. In *In CoopIS*, pages 179–194, 2001.

A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, Jon, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *In Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI*, pages 1–14, 2002.

Akamai. http://www.akamai.com/, 1998.

N. Akihiro, P. Larry, and B. Andy. Scalable routing overlay networks. *SIGOPS Oper. Syst. Rev.*, 40:49–61, 2006.

D. G. Andersen, H. Balakrishnan, and G. Andersen. Resilient overlay networks. In *Symposium on Operating Systems Principles*, pages 131–145, 2001.

D. P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10. IEEE Computer Society, 2004. ISBN 0-7695-2256-4.

M. anonymous remailer. http://mixmaster.sourceforge.net/.

S. Baset and H. Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. In *Proceedings of the 25th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies*, 2006.

L. Baud. Robust overlay network with self-adaptive topology: The reliable file storage layer. In *The 2009 IEEE - RIVF International Conference on Computing and Communication Technologies*, Da Nang, Viet Nam, July 2009.

L. Baud and P. Bellot. The rosa protocol adapted to aeronautical mobile ad-hoc network. In *8th Innovative Research Workshop & Exhibition (INO 2009)*, Brétigny sur Orge, France, Dec. 2009a.

L. Baud and P. Bellot. Robust overlay network with self-adaptive topology: The chain of lumps structure. In *2009 International Workshop on Peer-To-Peer Networking*, St. Pertersburg, July 2009b.

L. Baud, N. Pham, and P. Bellot. Robust Overlay Network with Self-Adaptive Topology: Protocol Description. In *2008 IEEE International Conference on Research, Innovation and Vision for the Future (RIVF 2008)*, Vietnam, Ho Chi Minh City, July 2008.

BitTorrent. http://www.bittorrent.com/, 2005.

S. Bohacek, J. Hespanha, K. Obraczka, J. Lee, and C. Lim. Enhancing security via stochastic routing. In *Computer Communications and Networks, 2002. Proceedings. Eleventh International Conference on*, pages 58–62, 2002.

L. Bölöni, D. Turgut, and D. C. Marinescu. n-Cycle: a set of algorithms for task distribution on a commodity grid. In *IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, volume 2, pages 615–622, May 2005.

L. Bölöni, D. Turgut, and D. C. Marinescu. Task distribution with a random overlay network. *Future Gener. Comput. Syst.*, 22(6):676–687, 2006. ISSN 0167-739X.

S. Botton, F. Duquenne, Y. Egels, and P. Wiliis. *GPS: localisation et navigation.* Edition Hermès, 1997.

R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. RFC 1633 (Informational), 1994.

E. Brunskill. Building peer-to-peer systems with chord, a distributed lookup service. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 81, Washington, DC, USA, 2001. IEEE Computer Society.

T. Bu, S. Norden, and T. Woo. A survivable dos-resistant overlay network. *Comput. Netw.*, 50(9):1281–1301, 2006.

A. R. Butt, R. Zhang, and Y. C. Hu. A self-organizing flock of condors. *J. Parallel Distrib. Comput.*, 66(1):145–161, 2006. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2005.06.022.

P. Cappello and D. Mourloukos. Cx: A scalable, robust network for parallel computing. *Sci. Program.*, 10(2):159–171, 2002. ISSN 1058-9244.

CASCADE Project Eurocontrol. Preliminary safety case for enhanced air traffic services in non-radar areas using ads-b surveillance. 2008.

M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. *SIGOPS Oper. Syst. Rev.*, 36:299–314, 2002. ISSN 0163-5980.

Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like p2p systems scalable. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 407–418, 2003.

I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46+, 2001.

F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 202–215, New York, NY, USA, 2001. ACM. ISBN 1-58113-389-8.

R. Dingledine, M. J. Freedman, and D. Molnar. The free haven project: Distributed anonymous storage service. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, volume 2009 of *Lecture Notes in Computer Science*, pages 67–95. Springer, 2000.

R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *In Proceedings of the 13th USENIX Security Symposium*, pages 303–320, 2004.

C. U. Dnad. On the feasibility of distributed intrusion detection, 2004.

J. R. Douceur. The sybil attack. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer-Verlag, 2002.

P. Druschel and A. Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *In HotOS VIII*, pages 75–80, 2001.

A. Dufour and L. Trajković. Improving gnutella network performance using synthetic coordinates. In *QShine '06: Proceedings of the 3rd international conference on Quality of service in heterogeneous wired/wireless networks*, page 31. ACM, 2006.

H. Eriksson. Mbone: the multicast backbone. *Commun. ACM*, 37(8):54–60, 1994. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/179606.179627.

N. S. Fard and T.-H. Lee. Cutset enumeration of network systems with link and node failures. *Reliability Engineering and System Safety*, 65(2):141–146, 1999.

G. Fedak, C. Germain, V. Neri, and F. Cappello. Xtremweb: A generic global computing system. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 582. IEEE Computer Society, 2001.

Gnutella. The annotated gnutella protocol specification v0.4., 2003.

Google. http://www.google.com, 1998.

D. Grace, M. Oodo, and P. Mitchell. An overview of the capanina project and its proposed radio regulatory strategy for aerial platforms. 2005.

G. Gu, P. Fogla, W. Lee, and D. Blough. DSO: Dependable Signing Overlay. In *Proceedings of the 2006 International Conference on Applied Cryptography and Network Security (ACNS'06)*, June 2006.

S. Gunther, K. Christopher, and K. Engin. Overbot - a botnet protocol based on kademlia. In *SecureComm 2008, 4th International Conference on Security and Privacy in Communication Networks*, 09 2008.

I. Gupta, K. P. Birman, P. Linga, A. J. Demers, and R. van Renesse. Kelips: Building an efficient and stable p2p dht through increased memory and background overhead. In *Proceedings of the Peer-to-Peer Systems II, Second International Workshop, IPTPS 2003*, pages 160–169, 2003.

M. Hefeeda, A. Habib, B. Botev, D. Xu, and B. K. Bhargava. Promise: peer-to-peer media streaming using collectcast. In *Proceedings of the Eleventh ACM International Conference on Multimedia*, pages 45–54, 2003a.

M. Hefeeda, A. Habib, D. Xu, B. Bhargava, and B. Botev. Collectcast: A peer-to-peer service for media streaming. *ACM Multimedia 2003*, 11:68–81, 2003b.

M. Henzinffer, S. Rao, and H. Gabow. Computing vertex connectivity: new bounds from old techniques. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:462, 1996.

J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Technical Report CSTR 41, Bell Laboratories, Murray Hill, NJ, 1976.

iVisit. http://www.ivisit.com/, 1997.

V. Jacobson. traceroute: ftp://ftp.ee.lbl.gov/traceroute.tar.gz, 1989.

R. Janakiraman, M. Waldvogel, and Q. Zhang. Indra: A peer-to-peer approach to network intrusion detection and prevention. In *In Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security*, pages 226–231, 2001.

J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole, Jr. Overcast: reliable multicasting with on overlay network. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 14–14, Berkeley, CA, USA, 2000. USENIX Association.

X. Jiang, Y. Dong, D. Xu, and B. Bhargava. Gnustream: A p2p media streaming system prototype. In *Proceedings of the International Conference on Multimedia and Expo (ICME*, pages 325–328, 2003.

H. Johansen, A. Allavena, and R. van Renesse. Fireflies: scalable support for intrusion-tolerant network overlays. *SIGOPS Oper. Syst. Rev.*, 40(4):3–13, 2006. ISSN 0163-5980.

D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *In ACM Symposium on Theory of Computing*, pages 654–663, 1997.

A. D. Keromytis, V. Misra, and D. Rubenstein. Sos: secure overlay services. In *Proceedings of the ACM SIGCOMM 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 61–72, 2002.

X. G. Klara, X. Gu, K. Nahrstedt, R. N. Chang, and Z. yin Shae. An overlay based qos-aware voice-over-ip conferencing system. In *IEEE Intern. Conf. on Multimedia and Expo (ICME2004*, pages 27–30, 2004.

J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceeedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, pages 190–201, November 2000.

C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. Delayed internet routing convergence. In *in Proc. ACM SIGCOMM*, pages 175–187, 2000.

L. Lao, S. S. Gokhale, and J.-H. Cui. Distributed qos routing for backbone overlay networks. In *Networking*, volume 3976 of *Lecture Notes in Computer Science*, pages 1014–1025. Springer, 2006.

J. Leskovec and E. Horvitz. Planetary-scale views on an instant-messaging network, 2008. URL `http://www.citebase.org/abstract?id=oai:arXiv.org:0803.0939`.

Z. Li and P. Mohapatra. Qron: Qos-aware routing in overlay networks. *Selected Areas in Communications, IEEE Journal on*, 22(1):29–40, Jan. 2004.

J. Liang, R. Kumar, and K. W. Ross. The kazaa overlay: A measurement study. *Computer Networks Journal (Elsevier*, 2005.

M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - a hunter of idle workstations. In *ICDCS*, pages 104–111, 1988.

F. Luis, B. Viktors, A. Jonathan, K. Mike, N. Andreas, M. Takagi, B. Richard, A. Adeeb, M. Ryo, H. Olegario, M. James, and B. Norbert. *Introduction to grid computing with globus*. IBM Corp., 2003.

D. Malkhi, O. Rodeh, M. K. Reiter, and Y. Sella. Efficient update diffusion in byzantine environments. In *SRDS*, pages 90–98. IEEE Computer Society, 2001.

M. Matyas, M. Peyravian, A. Roginsky, and N. Zunic. Reversible data mixing procedure for efficient public-key encryption. *Computers & Security*, 17(3):265–272, 1998.

P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, London, UK, 2002. Springer-Verlag. ISBN 3-540-44179-4.

Meebo. http://www.meebo.com/, 2005.

M. Misha. Cascade : an attack resistant peer-to-peer system. 2003.

R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM 2001*, pages 149–160, 2001.

A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: a read/write peer-to-peer file system. *SIGOPS Oper. Syst. Rev.*, 36(SI):31–44, 2002. ISSN 0163-5980.

M. Muuss. ping, 1983.

Napster, 1999.

K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474 (Proposed Standard), 1998.

V. E. Paxson. *Measurements and analysis of end-to-end Internet dynamics*. PhD thesis, 1998.

D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. Almi: an application level multicast infrastructure. In *USITS'01: Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems*, pages 5–5, Berkeley, CA, USA, 2001. USENIX Association.

N. Pham and M. Riguidel. Security assurance aggregation for it infrastructures. In *ICSNC '07: Proceedings of the Second International Conference on Systems and Networks Communications*, page 72. IEEE Computer Society, 2007.

N. Pham, L. Baud, P. Bellot, and M. Riguidel. Towards a security cockpit. *isa*, 0:374–379, 2008.

G. Pierre and M. van Steen. Globule: a collaborative content delivery network. *IEEE Communications Magazine*, 44(8):127–133, 2006.

C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 311–320. ACM, 1997.

S. Qazi and T. Moors. Scalable resilient overlay networks using destination-guided detouring. In *Proceedings of the IEEE International Conference on Communications (ICC)*, 2007.

S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, 2001.

Y. Rekhter and T. Li. A border gateway protocol 4 (bgp-4), 1995.

E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching architecture, 2001.

A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Lecture Notes in Computer Science*, pages 329–350, 2001.

A. H. S. Enumeration of minimal cutsets of an undirected graph. *Microelectronics and reliability*, 30(1):23–26, 1990.

S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, and J. Zahorjan. Detour: Informed internet routing and transport. *IEEE Micro*, 19(1):50–59, 1999.

B. Schneier. *Secrets and Lies: Digital Security in a Networked World.* John Wiley & Sons, 2004.

SecondLife. http://www.secondlife.com, 2003.

A. Singh, T.-W. Ngan, P. Druschel, and D. S. Wallach. Eclipse attacks on overlay networks: Threats and defenses. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–12, 2006.

L. Subramanian, I. Stoica, H. Balakrishnan, and R. Katz. OverQoS: An Overlay Based Architecture for Enhancing Internet QoS. In *1st Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, March 2004.

C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *In Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 224–237, 1997.

J. Touch and S. Hotz. The x-bone. In *Proceedings of the 3rd Global Internet Mini-Conference/Globecom*, 1998.

J. D. Touch, Y.-S. Wang, V. Pingali, L. Eggert, R. Zhou, and G. G. Finn. A global x-bone for network experiments. In *Proceedings of the First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, pages 194–203, 2005.

Twinverse. http://twinverse.com/, 2008.

D. J. Watts and S. H. Strogatz. Collective dynamics of'small-world'networks. *Nature*, 393 (6684):409–10, 1998.

A. F. Webster and S. E. Tavares. On the design of s-boxes. In *CRYPTO '85: Advances in Cryptology*, pages 523–534, London, UK, 1986. Springer-Verlag. ISBN 3-540-16463-4.

Wikipedia. http://www.wikipedia.org, 2001.

L. Xiong, L. Liu, and I. C. Society. Peertrust: Supporting reputation-based trust for peer-to-peer electronic communities. *IEEE Transactions on Knowledge and Data Engineering*, 16:843–857, 2004.

V. Yegneswaran, P. Barford, and S. Jha. Global intrusion detection in the domino overlay system. In *In Proceedings of Network and Distributed System Security Symposium (NDSS*, 2004.

X. Zhang, J. Liu, B. Li, and T.-S. P. Yum. Coolstreaming/donet: a data-driven overlay network for peer-to-peer live media streaming. In *Proceedings of INFOCOM. 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 2102–2111, 2005.

Y. Zhang, V. Paxson, and S. Shenker. The stationarity of internet path properties: routing, loss, and throughput. Technical report, In ACIRI Technical Report, 2000.

B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22:41–53, 2004.

L. Zhuang, F. Zhou, B. Y. Zhao, and A. Rowstron. Cashmere: resilient anonymous routing. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 301–314, Berkeley, CA, USA, 2005. USENIX Association.

S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *NOSSDAV '01: Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, pages 11–20, New York, NY, USA, 2001a. ACM Press.

S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *NOSSDAV '01: Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, pages 11–20, New York, NY, USA, 2001b. ACM.

Index