HAL
open science

Non-observabilité des communications à faible latence

Nicolas Bernard

To cite this version:

Nicolas Bernard. Non-observabilité des communications à faible latence. Réseaux et télécommunications [cs.NI]. Université Joseph-Fourier - Grenoble I; université du Luxembourg, 2008. Français. NNT : . tel-00325234

HAL Id: tel-00325234

https://theses.hal.science/tel-00325234v1

Submitted on 26 Sep 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université du Luxembourg
Faculté des Sciences, de la Technologie
et de la Communication

Université Grenoble I – Joseph Fourier
École doctorale Mathématiques, Sciences et Technologies
de l'Information, Informatique

PhD-FSTC-4-2008

___/___/___/___/___

Thèse de doctorat en cotutelle internationale

Informatique

spécialité sécurité des systèmes et réseaux

Non-observabilité des communications à faible latence
ou
« Comment Alice peut-elle *chatter* avec Bob sans qu'Eve et Mallory le sachent ? »

présentée et soutenue publiquement par

Nicolas Bernard
né le 15 septembre 1981

le 23 septembre 2008

pour l'obtention des titres de

Docteur de l'Université du Luxembourg
Docteur de l'Université Grenoble I – Joseph Fourier

devant un jury composé de :

- Yves Denneulin, maître de conférences, Institut National Polytechnique de Grenoble ;
- Olivier Festor, directeur de recherches, INRIA Lorraine, rapporteur ;
- Franck Leprévost, professeur, Université du Luxembourg, directeur de thèse ;
- Bruno Martin, maître de conférences, Université de Nice Sophia Antipolis, rapporteur ;
- Jean-François Méhaut, professeur, Université Grenoble 1 - Joseph Fourier, directeur de thèse ;
- Serge Vaudenay, professeur, École Polytechnique Fédérale de Lausanne, président du jury.

Empreinte de la version (`git`)
`c1ffcbd9e8e9a58e184d252e11780993c96c1b93`
Document compilé (LaTeX) le 31 août 2008.

Thèse préparée au
Laboratoire d'Algorithmique, Cryptologie et Sécurité
(LACS/CSC, Université du Luxembourg, Luxembourg)
et au
Laboratoire d'Informatique de Grenoble
(LIG, anciennement ID-IMAG, Université Joseph Fourier, Grenoble, France)

Cette thèse est formatée avec LaTeX. Sauf exception, les figures ont été réalisées avec `xfig` et les graphiques à l'aide de `gnuplot`.
Pour une lisibilité optimale, ce document devrait être imprimé en couleurs.

Non-observabilité des communications à faible latence

Résumé

Cette thèse s'articule autour de deux parties, toutes deux liées à la protection de la vie privée dans les réseaux informatiques en général et sur l'Internet en particulier.

Dans la première partie, nous proposons un système permettant d'établir des communications interactives non-observables, c'est-à-dire telles qu'un observateur ne puisse pas déterminer vers quelle(s) destination(s) sont établies ces communications, ni même, en fait, être certain qu'il y a bien de vraies communications ! Ce système innove par le niveau de protection qu'il vise, puisque même un observateur très puissant ne devrait pas être à même de la contourner. Cette protection se base sur l'*Onion-Routing* et le complète avec des méthodes sophistiquées destinées à déjouer l'analyse de trafic.

Dans la seconde partie, nous nous intéressons plus particulièrement au protocole DNS. Bien qu'il soit possible de le protéger avec notre proposition de la partie précédente, cela dégrade ses performances (en termes de latence), ce qui à son tour a un impact sur celles des protocoles qui utilisent DNS. Dans cette partie, nous proposons une solution spécifique à DNS, qui fournit à la fois un bon niveau de protection et de meilleures performances.

Ces deux systèmes peuvent bien sûr se combiner, mais aussi être utilisés comme des briques séparées, avec d'autre mécanismes de protection de la vie privée.

Mots clefs : sécurité réseau, vie privée, non-observabilité, anonymat, DNS, *Onion-Routing*, analyse de trafic.

Unobservability of Low Latency Communications

Abstract

This thesis is built around two parts, both related to the protection of privacy in computer networks, and more precisely on the Internet.

In the first part, we propose a system allowing the establishment of unobservable interactive communications, unobservable meaning an observer can neither pinpoint the destination(s) of those communications, nor, in fact, know if there are real communications in the first place ! This system aims to provide an unequaled level of protection, as even a very powerful observer should not be able to bypass it. This protection is based on Onion-Routing and adds innovative methods against traffic analysis.

In the second part, we take a closer look at the DNS protocol. While it is possible to protect it with our proposition described in the first part, this degrades performance, specifically latency, which has an impact on those protocols using DNS. In this part, we propose a DNS specific solution, providing both a good level of protection and better performance.

These two systems can be combined of course, but they can also be used as separate bricks with other privacy enhancing mechanisms too.

Keywords : network security, privacy, unobservability, anonymity, Onion-Routing, DNS, traffic analysis.

Civilization is the progress toward a society of privacy. The savage's whole existence is public, ruled by the laws of his tribe. Civilization is the process of setting man free from men.

— Ayn Rand, *The Fountainhead* (1943)

Remerciements

Cotutelle oblige, il y a beaucoup (deux fois plus ? Est-ce linéaire ?) de personnes à remercier. Au risque de passer pour un ours, je ne m'étendrai pas ici en de longs remerciements individualisés. Après tout, cette thèse traite de la protection de la vie privée, et je ne vois pas de raison de rendre public ce qui ne le serait pas déjà (une alternative serait d'écrire des remerciements personnels en changeant les noms et les rôles !).
Que tous ceux qui ne sont pas mentionnés ici pour cette raison (qu'ils n'aillent pas croire que je les ai oubliés !) et qui auraient préféré y figurer plutôt que de voir préservée l'intimité — dont ils n'ont que faire — de leur vie privée, me pardonnent.

Je remercie donc tout d'abord vivement mes directeurs de thèse, officiels et officieux, qui ont su s'intéresser à mes travaux tout en me laissant très libre.
Je voudrais également remercier chaleureusement Olivier Festor et Bruno Martin, qui ont accepté d'être rapporteurs de cette thèse à une période où la plupart des gens prétendent à des vacances.
J'adresse aussi mes remerciements à Serge Vaudenay, qui a accepté de présider ce jury.

J'ai encore le plaisir de remercier tous ceux qui ont rendu les tests décrits au chapitre 7 possibles, en particulier Eddy Caron, Bruno Delcourt, Simon François et Oliver Voigt. Un grand merci également aux gens du SIU de Limpertsberg, qui ont dû supporter avec patience mes demandes parfois exotiques.

Je voudrais aussi remercier toutes les personnes amicales que j'ai côtoyées durant ces années de thèse :

- à Grenoble, cela comprend pratiquement tout le laboratoire ID (« pratiquement » non pas parce que j'y ai rencontré des gens antipathiques, mais parce qu'il doit y avoir quelques personnes que je n'ai jamais croisées !).
- à Luxembourg, il faudrait aussi remercier beaucoup de monde. Mes pensées vont naturellement vers les rares autres thésards présents à mon arrivée : Sébastien, qui a longtemps dû me subir dans son bureau et m'a entrainé dans des aventures chronophages telles que la publication d'un livre, Patrick et Geneviève et, bien sûr, *last but not least*, Nathalie.
  D'une manière plus générale, que tous ceux que j'ai à un moment ou un autre invités à boire un thé dans mon bureau soient remerciés.

J'ai aussi une pensée pour mes « co-thésards à distance » qui viennent de ou qui vont soutenir leurs thèses respectives : Laurent, Mathieu et [l'autre] Sébastien.

Finalement, un autre grand merci, à ma mère cette fois, pour son soutien, ses relectures et tout le reste.

Table des matières

Résumé iii
Abstract iv
Remerciements vi
Table des matières vii

0 Introduction : *dramatis personae* & contexte **xi**
Contexte xi
La *vie privée* xii
Ce sur quoi portent nos travaux xiii
Avertissement au lecteur xv

1 L'existant : forces et faiblesses **1**
1.1 Les protocoles de sécurité « traditionnels » et leurs limites 1
1.1.1 SSL et TLS 2
1.1.2 Les autres protocoles 9
1.1.3 NAT et proxies 10
1.2 Anonymat des communications à forte latence 11
1.2.1 Mix-nets et remailers 11
1.2.2 DC-nets 13
1.3 Anonymat des communications à faible latence 14
1.3.1 Onion-Routing 15
1.3.2 Crowds 19
1.3.3 Attaques 20

I *True Nyms* : vers des communications non-observables à faible latence **21**

2 Introduction et vue globale **23**
2.1 Un point de vocabulaire 23
2.2 Le problème à résoudre 24
2.3 La protection contre l'analyse de trafic 25
2.4 La théorie et la pratique 25

3 Les problèmes de l'établissement des communications **27**
3.1 L'établissement de connexion : Diffie-Hellman 27
3.1.1 Diffie-Hellman et le *Man-in-the-Middle* 28
3.1.2 Les fuites d'information du protocole de Diffie et Hellman 29
3.1.3 « Notre » protocole de Diffie-Hellman 29
3.2 L'établissement de communications 30
3.2.1 Un *pool de routes* pré-établies 30

3.2.2 Mécanismes anti-observation 31
3.3 La fin des connexions et des communications 34

4 Le lissage de trafic et ses implications 37
4.1 Pourquoi lisser le trafic ? 37
4.2 Problème : taille et période d'émission 39
4.3 Problème : perte de paquet leurre 40
4.4 Problème : débit 42
4.4.1 Routes multiples 42
4.5 Problème : discrétion 43

5 Protection contre le rejeu de paquet et ses variantes 45
5.1 Empêcher le rejeu de paquet 45
5.1.1 Les filtres de Bloom 46
5.1.2 Le problème des filtres de Bloom 47
5.1.3 Modification des filtres de Bloom pour éviter les dénis de service 48
5.2 Rendre le rejeu de paquet avec modification inutile 49
5.3 Implications sur l'architecture du système 50
5.3.1 Les cas possibles 50
5.3.2 Position de notre solution 51

6 Bref aperçu du protocole *True Nyms* et analyse de sécurité 53
6.1 Le protocole *True Nyms* 53
6.1.1 Une question de couche 54
6.1.2 Au niveau réseau 55
6.1.3 Paramètres 57
6.2 Sécurité 57
6.2.1 Attaque par corrélation globale 57

7 Implémentation et évaluation 59
7.1 Notre implémentation 59
7.1.1 Architecture générale du démon `nymsd` 60
7.1.2 Le processus `writer` 63
7.1.3 Le processus `listener` et l'utilisation d'UDP 63
7.1.4 Notes diverses sur l'implémentation 63
7.2 Évaluation des performances 64
7.2.1 Procédure de test 64
7.2.2 Établissement des routes 66
7.2.3 Performances des routes 68
7.2.4 Conclusion sur les tests 70

8 Usage(s), conclusion et problèmes ouverts 73
8.1 Usage(s) de *True Nyms* 73
8.2 Améliorations possibles et problèmes ouverts 75
8.2.1 Quelques idées pour améliorer les performances 75
8.2.2 Quelques problèmes (toujours) ouverts 77

II Vie privée dans le *Domain Name System* 79

9 Le fonctionnement du DNS 81
9.1 DNS : une base de données distribuée et hiérarchisée 81
9.1.1 Une architecture arborescente 82
9.1.2 Principe de la résolution d'un nom 82
9.2 La manière dont DNS est déployé 83
9.2.1 Un réseau de caches 83
9.2.2 La résolution de nom dans la pratique 83
9.3 Un aperçu du protocole 84
9.4 La sécurité dans le DNS 86
9.4.1 Authentification et intégrité : *Spoofing* et *Cache poisonning* 86
9.4.2 Dénis de service 88
9.4.3 Confidentialité 89
9.4.4 Autres abus 90

10 L'intimité dans le DNS 91
10.1 Quelques solutions naïves 92
10.1.1 Interroger directement la hiérarchie DNS 92
10.1.2 Un cache DNS chiffré 93
10.1.3 Un ensemble de caches DNS chiffrés 93
10.2 Vers une solution évoluée 94

11 Un réseau pair-à-pair de caches DNS inspiré de Crowds 95
11.1 Description 95
11.2 Que doit-on cacher ? 97
11.3 Remplissage du cache 98

12 Remplir le cache 101
12.1 Auto-rafraîchissement des données 101
12.2 Tricher sur la date de péremption 102
12.3 Servir des données périmées 102
12.4 Synchroniser réponses et rafraîchissements 104
12.5 A-t-on encore besoin du réseau P2P ? 104

13 Trois options envisageables 105
13.1 Un cache désynchronisé local 105
13.2 Un cache désynchronisé qui utilise TN pour le trafic DNS 106
13.3 Un réseau P2P de caches (désynchronisés) 107
13.4 Comparaison récapitulative 107

14 Implémentation et tests de notre proposition 109
14.1 L'implémentation 109
14.1.1 Le cache lui-même : `cachedatastruct2.c` 110
14.1.2 Analyse et création de paquets DNS : `dns.c` 110
14.1.3 La gestion du réseau P2P : `p2p.c` 111
14.1.4 L'interface entre les modules et la gestion du DNS : `pdns.c` 112
14.2 Tests 113

15 Conclusion sur la protection du DNS 115

16 Conclusion générale **117**

A Le protocole *True Nyms* : couches et format des paquets **121**
A.1 Couche transport 122
A.1.1 En-tête 122
A.1.2 Paquet non-acquiescé (≈ UDP) 122
A.1.3 Paquet acquiescé (≈ TCP) 123
A.2 Session données 124
A.3 Session de contrôle 125

B True Nyms : adaptation pour le « monde réel » **131**
B.1 Contexte et objectif 131
B.2 Anonymat version peer-to-peer (APP) 132
B.3 Anonymat version contrôlée (AC) 134

Bibliographie **137**

Electronic communication will be the fabric of tomorrow's society (...). By codifying the Governement's power to spy invisibly on these contacts, we take a giant step toward a world in which privacy belongs only to the wealthy, the powerful, and perhaps, the criminals.

— Whitfield Diffie

People are entitled to privacy, and some will kill for it.

— Larry Niven & Jerry Pournelle

0

Introduction : *dramatis personae* & contexte

Dramatis Personae

Alice	une utilisatrice d'Internet ;
Bob	son correspondant ;
Eve	une espionne ; elle est discrète et se contente d'écouter aux *i*portes ;
Mallory	un autre espion ; il ne se contente pas d'écouter et n'hésite pas à agir si cela lui permet d'obtenir de l'information.

Contexte

Une définition pleine d'humour que l'on trouve sur l'Internet nous explique ce qu'est un cryptographe en ces termes :

> « Alice et Bob discutent quotidiennement via l'Internet de la fabrication de bombes, de terrorisme, de fraude fiscale, d'infidélité conjugale et de pratiques sexuelles tordues. Ils conspirent pour commettre des crimes, partager des textes interdits et des informations censurées, ou encore pour renverser des gouvernements tyranniques dont les agents espionnent toutes leurs communications. Ils font tout cela dans le plus grand secret, avec des chiffres incassables.
>
> Néanmoins Alice et Bob ne se font même pas confiance.
>
> Un cryptographe est quelqu'un qui ne pense pas qu'Alice et Bob sont fous. »

Si l'on peut remettre en cause la pertinence de cette définition – cette thèse traite des problèmes de communication entre Alice et Bob sans être une thèse

de cryptographie – elle a néanmoins l'intérêt d'introduire à merveille notre sujet : *comment faire pour qu'Eve et Mallory, les agents qui surveillent Alice, ne puissent savoir qu'elle communique avec Bob ?*

Une communication entre Alice et Bob au travers de l'Internet peut être attaquée chez Alice, chez Bob, ou entre les deux, lors de son passage par le réseau. Si, écrit ainsi, cela paraît évident, cela semble pourtant n'être que peu compris : il suffit pour s'en convaincre de voir l'importance – bien comprise des commerçants – accordée par la plupart des internautes au chiffrement des connexions quand ils achètent en ligne, alors que leurs propres ordinateurs sont, bien souvent, des nids de *spywares* et autres logiciels malveillants. En outre, dans le cas où Bob est un commerçant, un pirate a davantage intérêt à attaquer son site pour récupérer toute la base de données contenant les informations bancaires d'un seul coup.

Cela étant précisé, nous allons maintenant nous empresser de l'oublier : nous supposerons en effet dans ce qui suit que les machines d'Alice et Bob sont sûres et nous ne nous intéresserons qu'aux problèmes liés au passage sur le réseau de leurs communications.

À ce point, le lecteur peut s'interroger : il existe déjà plusieurs protocoles cryptographiques destinés à protéger les communications sur un réseau, qui chiffrent, authentifient et garantissent l'intégrité des données. Pour ne citer que les plus connus, on peut mentionner SSL et TLS, IPsec, ou encore SSH. Ne sont-ils pas suffisants pour Alice et Bob ?

Il faut être conscient que la sécurité de ces protocoles repose sur un certain nombre d'hypothèses. Sans remettre celles-ci en cause pour l'usage auquel les protocoles susmentionnés sont destinés, elles ne sont pas forcément adaptées à tous les cas, loin s'en faut. En particulier, si ces protocoles permettent de payer par Internet sans révéler son numéro de carte bancaire à l'ensemble de la planète, ils n'offrent que peu de garanties en ce qui concerne la vie privée. Un observateur peut ainsi savoir à quels sites vous vous connectez et avec qui vous communiquez. Même dans le cas du commerce électronique, pourtant réputé sûr, ces protocoles ne sont généralement utilisés que pour protéger le paiement ; aussi, si l'attaquant ne peut obtenir les informations nécessaires pour payer à son tour en votre nom, peut-il en général savoir exactement ce que vous achetez. Dans certains cas enfin, comme nous le verrons en 1.1, ces protocoles échouent même à protéger réellement le contenu de la communication, une attaque par analyse de trafic pouvant révéler celui-ci.

Ces protocoles échouent donc pour *protéger la vie privée* des individus.

La *vie privée*

Le concept de vie privée (qui correspond approximativement à celui de « *privacy* » en anglais) est relativement récent. Le terme « vie privée » lui-même, semble en effet vieux de seulement deux siècles : c'est peut-être Benjamin Constant qui l'a créé en 1819. Il en faisait une des principales caractéristiques de la *liberté des modernes*, par opposition à la *liberté des anciens* [22]. En suivant la pensée de Constant, on peut donc considérer que le concept est antérieur,

implicite à la liberté telle que conçue par Locke au XVIIe siècle, puis par les philosophes des Lumières.

La vie privée est depuis explicitement devenue un droit moderne. Actuellement, la protection de la vie privée est garantie par la loi dans de nombreux pays : dans l'Union Européenne, c'est un droit assuré par la convention européenne des droits de l'homme ; en France, le Code civil indique que « chacun a droit au respect de sa vie privée » (article 9). Il n'en existe néanmoins pas à notre connaissance de définition légale formelle. Selon [69], la vie privée serait difficile à cerner précisement car elle ne se réduirait pas à *un* critère précis, mais à un ensemble de critères liés ; c'est également ce qui ressort de [38].

Comme pour la liberté, le « besoin » de vie privée est ressenti différemment selon les individus. Cette différence incite néanmoins certains (ceux qui n'éprouvent pas ou peu ce besoin ?) à remettre ce droit en cause. Daniel Solove, dans [69], développe la thèse que cela vient de la pluralité des concepts qui se trouvent derrière celui de *vie privée* et s'appuie dessus pour réfuter les arguments du type « je suis honnête donc je n'ai rien à cacher ».

Dans son essai *The Transparent Society* [17], David Brin développe l'idée paradoxale qu'il faut sacrifier la vie privée pour protéger la liberté des effets pervers des nouvelles technologies : la surveillance de tout le monde par tout le monde permettrait d'échapper à celle de tout le monde par quelques-uns...

Dans son sens actuel *vie privée* implique *intimité*, même si l'on devrait en fait parler de l'*intimité de la vie privée* pour être tout à fait correct. Dans un monde idéal, la vie privée n'aurait pas forcément besoin d'être « intime » à partir du moment où les libertés individuelles seraient protégées. Dans le monde réel cependant les différences entre les individus étant souvent mal acceptées, l'intimité de la vie privée est nécessaire ; une illustration de ce fait est donnée par la « lutte anti-terroriste » qui depuis les attaques du 11 septembre 2001 ressemble trop souvent, comme l'écrit Bruce Schneier [66], à ceci :

> *It's an attack on the unique, the unorthodox, the unexpected ; it's a war on different. If you act different, you might find yourself investigated, questioned, and even arrested – even if you did nothing wrong, and had no intention of doing anything wrong.*

Sans nous attarder plus sur le débat visant à savoir si la vie privée est nécessaire ou néfaste (le lecteur intéressé pourra par exemple se reporter à [50]), nous allons maintenant nous intéresser aux moyens techniques qui permettent de la maintenir dans un monde en réseau.

L'objet de nos travaux

Le contexte qui nous intéresse est représenté par la figure 1 : il s'agit par exemple du cas d'une personne dont la liaison Internet est observée au niveau de son fournisseur d'accès, mais aussi de celui d'un individu dans une entreprise qui est surveillé par le « service informatique ». C'est la façon dont procèdent probablement les agents du gouvernement tyrannique – Eve et Mallory – pour observer Alice ou Bob. Si les moyens dont dispose Eve sont considérables, elle se contente cependant d'observer de manière passive. Mallory, lui, est moins discret

et il n'hésitera pas à émettre des données si cela peut lui permettre d'obtenir des informations supplémentaires.

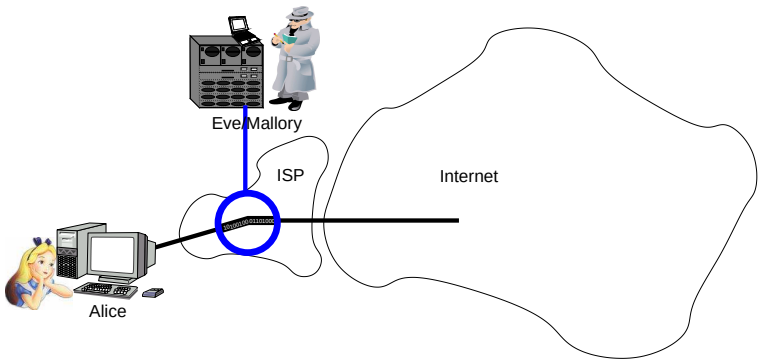


FIG. 1 – Contexte : l'utilisateur (Alice) est spécifiquement observé.

Dans tous les cas nous supposons que l'attaquant n'a pas accès à la machine de l'utilisateur, l'observation de celle-ci étant alors triviale. En revanche, nous supposons qu'Eve et Mallory disposent de ressources énormes, qui leur permettent d'observer (et, pour Mallory, d'émettre sur) les liens d'une partie importante de l'Internet, ainsi que de machines dispersées (ou non) sur le réseau.

Bien que tel que présenté ci-dessus notre contexte semble se focaliser sur le cas où un individu précis est la cible de l'attaquant, nous ne présumons aucunement que chacun des utilisateurs du réseau n'est pas soumis à une surveillance identique par le même attaquant. Nous supposons que celui-ci, s'il n'est pas limité aux attaques passives, tente cependant d'éviter d'être détecté.

Après une présentation rapide des protocoles existants et de leurs problèmes respectifs, nous nous intéresserons dans une première partie au cas général de la protection des communications à faible latence (*i.e.*, « interactives »), et présenterons diverses idées pour améliorer la résistance du classique mécanisme d'*onion-routing* à l'analyse de trafic ainsi que pour améliorer ses performances.

Bien que notre méthode soit aussi générique que possible, nous verrons qu'il reste, avec certains protocoles, des problèmes de performances. Nous nous intéresserons en particulier au cas du système de noms de domaines (DNS) et à sa protection. En effet, comme nous le verrons, ce protocole quasi indispensable :

- soit provoque une fuite d'information pouvant potentiellement réduire à néant d'autres efforts pour masquer l'activité de l'utilisateur ;
- soit pénalise les performances de l'accès à l'Internet de manière conséquente (par exemple s'il est utilisé avec notre solution générique).

Bien que la première partie fournisse les motivations qui ont conduit à la seconde, elles restent néanmoins indépendantes.

Si les travaux présentés dans ce document restent dans le cadre général de la sécurité informatique, ils présentent cependant une certaine variété : ainsi le début (chapitre 1) consiste en une analyse critique des protocoles existants et en la mise en oeuvre d'*attaques* sur ces protocoles ; ensuite, la première partie, tout en s'appuyant sur des idées pré-existantes, fait *tabula rasa* et l'on y bâtit un protocole nouveau en y intégrant un certain nombre de propositions nouvelles.

Dans la seconde partie par contre, nous nous livrons à l'exercice périlleux qui consiste à modifier l'existant pour améliorer sa sécurité.

Avertissement au lecteur

Finalement, avant de laisser le lecteur découvrir le contenu des chapitres qui suivent, nous aimerions lui présenter tout à la fois un avertissement et des excuses. En effet, comme il le constatera, nous nous sommes en grande partie dispensé des traditionnels chapitres introduisant les notions nécessaires, pour plonger directement dans l'état de l'art. À notre décharge, il faut expliquer l'une des causes de cet oubli volontaire : cette thèse est à cheval sur plusieurs disciplines, tenant à la fois de la cryptologie, de la sécurité et des réseaux informatiques, avec des incursions dans l'algorithmique, l'architecture logicielle et la programmation.

Il nous semble qu'il serait illusoire de vouloir expliquer en quelques pages les notions parfois très complexes de ces domaines à un lecteur qui ne les connaîtrait pas déjà.

Une introduction à la cryptographie à la fois suffisamment poussée et accessible aurait déjà une taille équivalente à l'ensemble de ce document, et plus encore si nous devions parler du cas particulier de la cryptographie basée sur les courbes elliptiques, ce qu'un de nos directeurs de thèse eût alors sans doute suggéré !

De même, une description des protocoles réseaux que nous utilisons ou dont nous nous inspirons serait forcément longue. À titre d'exemple, le protocole que nous présentons dans la première partie, *True Nyms*, reprend des idées de TCP et une bonne connaissance de celui-ci serait requise pour comprendre certains détails. Malheureusement, une description détaillée de TCP serait forcément longue : celle donnée dans [70] fait, par exemple, plus de 130 pages !

Nous pourrions multiplier ainsi les exemples. Nous aurions, certes, pu faire quelques chapitres pour survoler très rapidement les différents concepts utilisés, mais il nous semble qu'ils n'auraient pas apporté grand chose au lecteur et risqué de l'ennuyer. Pour cette raison, nous avons préféré plonger le lecteur directement dans le vif du sujet dès les pages suivantes, avec parfois un maigre encadré pour rappeler une notion, qui, s'il ne lui permettra pas à lui seul de comprendre finement les choses, devrait — nous l'espérons — lui permettre de saisir de quoi il retourne.

Il y a bien sûr une exception qui confirme la règle : dans la seconde partie, nous revenons durant un chapitre entier sur le fonctionnement de DNS (chapitre 9) : cela nous semble nécessaire car les détails de ce protocole sont à la fois indispensables à la compréhension de la suite de cette partie et souvent méconnus même des spécialistes en réseaux.

Enfin, le lecteur le verra, nous sommes par nature plus attiré par les concepts que par les détails. Cela tient à notre avis à la façon dont la recherche est parfois présentée : « 1% d'inspiration et 99% de transpiration ». Dans cette thèse, nous décrivons avant tout le « 1% d'inspiration ». La transpiration est laissée de côté dans ce document, même si quelques gouttes de sueur se sont glissées, par

exemple dans les chapitres 7 et 14 : l'implémentation de nos idées a probablement occupé 99% du temps de la thèse.

L'avantage pour le lecteur est que cela se traduit notamment par des chapitres dont sont absents la plupart des *gory details* : nous considérons que l'important est qu'ils soient dans le code source. Notre texte est du coup concis, mais il reste dense (certains diraient « une forte entropie ») ; il peut demander un effort de réflexion au lecteur pour qu'il absorbe toutes les implications : une idée peut en cacher une autre. Pour faciliter ces moments de réflexion, nous avons privilégié l'utilisation de courts chapitres, sur des thèmes précis : le lecteur peut donc reprendre son souffle assez fréquemment.

Durant cette thèse, nous avons également travaillé sur des points dont le lecteur ne trouvera pas mention dans ce manuscrit. En effet, dans un souci d'homogénéïté et de concision, nous avons délibérement laissé de côté certains de nos travaux. Notons simplement que, au-delà du sujet très spécialisé des pages qui suivent, nous avons également contribué – à divers degrés – à diverses publications dans un contexte plus vaste, allant du relativement proche (protection de la vie privée dans le cadre du commerce électronique [26, 27]), à plus lointain (courbes elliptiques [12], écriture d'un livre de C [73]), avec des étapes intermédiaires (sécurité informatique « générale » [28, 25, 10, 30]).

[The Internet is] a world where the line between being concerned about privacy and becoming clinically paranoid is fairly thin.

Michal Zalewski, *in* [82]

1

L'existant : forces et faiblesses

La cryptographie est souvent présentée comme la solution aux problèmes de vie privée des communications, et en particulier sur l'Internet. Il est vrai que si Alice peut chiffrer un message en s'assurant qu'il ne sera lisible que par Bob et réciproquement, cela leur permet d'avoir des discussions privées.

Pourtant, cela n'est pas toujours suffisant. Même si l'on suppose que la cryptographie est sûre et ne peut être cassée par l'attaquant, celui-ci peut souvent obtenir des informations précieuses par *analyse de trafic*. Ce type d'analyse est pratiqué au moins depuis que les communications par voie électronique existent et était déjà répandu durant la première guerre mondiale [44]. Cryptanalyse et analyse de trafic sont parfois regroupées dans le terme *sigint* (pour *signal intelligence*).

Si les militaires ont bien compris les risques de l'analyse de trafic – une émission radio indique la position de l'émetteur, que son contenu soit chiffré ou non – les protocoles cryptographiques traditionnels, employés actuellement sur l'Internet, ne permettent pas d'éviter ce problème. Et les protocoles existants pour protéger la vie privée restent souvent vulnérables face à un attaquant puissant.

1.1 Les protocoles de sécurité « traditionnels » et leurs limites

Ce que nous qualifierons dans cette section de protocole cryptographique « traditionnel » est un protocole qui permet à Alice et Bob de s'authentifier mutuellement, de chiffrer leurs échanges et de garantir l'intégrité des données tranférées ; SSL ou IPsec pour les communications interactives ou encore OpenPGP et S/MIME pour les emails sont des exemples de tels protocoles.

Ces protocoles sont une application directe des méthodes cryptographiques modernes :

- la cryptographie asymétrique est employée pour chiffrer une clef de session symétrique aléatoire et pour signer [une empreinte des] les données. Elle n'est généralement pas employée pour chiffrer les données elles-mêmes en raison du temps de calcul que cela impliquerait ;
- la cryptographie symétrique, plus rapide, est utilisée pour chiffrer les données elles-mêmes ;
- une fonction de hachage cryptographique est utilisée pour générer une empreinte des données, empreinte qui sera signée en lieu et place de celles-ci pour les authentifier.

Cryptographie asymétrique

Chaque participant dispose d'une clef publique, qu'il diffuse, et d'une clef privée correspondant à la clef publique, qu'il maintient secrète.
- Des données chiffrées avec la clef publique ne sont lisibles que par une personne qui détient la clef privée correspondante.
- Inversement des données « chiffrées » avec la clef secrète sont déchiffrables par toute personne possédant la clef publique correspondante. Cette personne est alors assurée que ces données ont bien été chiffrées par la clef secrète en question et donc par le détenteur de celle-ci : c'est le principe de la signature électronique.

Ces protocoles ne chiffrent néanmoins que le contenu de la communication ; les participants sont généralement identifiables facilement et, parfois, connaître cette information permet de déterminer le contenu, pourtant chiffré, de la communication.

Dans la suite de cette section, nous décrivons en détail ce qui se passe dans le cas où l'on utilise SSL/TLS pour protéger la navigation sur le Web, puis nous reviendrons brièvement sur d'autres protocoles cryptographiques courants, avant de terminer sur deux technologies souvent présentées comme permettant de protéger la vie privée, à savoir la traduction d'adresses (NAT) et les proxies.

1.1.1 SSL et TLS

SSL (Secure Socket Layer) et son évolution TLS (Transport Layer Security) [63] sont les protocoles habituellement utilisés pour sécuriser le trafic Web entre le client et le serveur. Ils se présentent sous forme de couches cryptographiques s'intercalant entre TCP et le niveau applicatif dans le modèle IP.

SSL et TLS sont également utilisés pour la protection de divers flux TCP, mais nous allons nous intéresser dans la suite de cette section à leur usage pour la protection des connexions vers des serveurs Web, ce qui se fait en général avec `https`. Il s'agit simplement d'une connexion `http` normale au-dessus d'un tunnel SSL ou TLS établi vers le port 443 du serveur Web.

Une attaque de Mallory

Il existe une attaque que Mallory peut mettre en œuvre assez simplement. Elle est bien connue et nous n'entrerons donc pas dans les détails ; elle est

d'ailleurs pratiquée de manière institutionnelle dans certaines entreprises, officiellement pour *sécuriser le trafic chiffré*. Il s'agit tout simplement de l'attaque de l'intercepteur, ou *Man-in-the-Middle* (MitM). L'idée ici est que Mallory place un proxy transparent entre Alice et Bob : il se fait passer pour Bob vis-à-vis d'Alice, et pour Alice vis-à-vis de Bob, en générant dynamiquement des certificats quand ils sont demandés grâce à sa propre *autorité de certification* (CA).

SSL et TLS sont conçus pour résister à cette attaque, grâce à l'authentification fournie par une infrastructure à clefs publiques (c'est-à-dire basée sur des CA auxquelles « l'utilisateur » fait confiance). Malheureusement, dans la pratique, cela requiert encore un utilisateur paranoïaque : l'attaque a donc de fortes chances de fonctionner. Deux cas de figure sont courants :

- soit Mallory peut ajouter le certificat racine de sa CA dans l'ordinateur d'Alice (ce qui est généralement le cas dans une entreprise, l'utilisateur et l'administrateur étant deux personnes distinctes). Ce certificat sera visible dans la liste des autorités de certification de confiance[1], mais aucun message ne s'affichera lorsqu'Alice ira sur le site de Bob, le proxy ayant automatiquement généré un certificat lui permettant de se faire passer pour ce site avec l'autorité de certification de Mallory ;
- soit Mallory ne peut rien changer sur l'ordinateur d'Alice. Le certificat généré lorsqu'Alice ira sur le site de Bob ne sera alors pas "signé par une autorité de certification à laquelle on fait confiance" et l'ordinateur affichera généralement un avertissement. La plupart du temps, Alice cliquera sur « continuer »[2] sans même lire le message, ce genre de boîte de dialogue étant relativement fréquent et n'indiquant de plus pas forcément une attaque. Si ce message apparaît pour chaque site chiffré, il faut pourtant se poser des questions !

L'attaque d'Eve

Eve peut également mettre en œuvre une attaque, beaucoup plus subtile, nécessitant plus de moyens pour être déployée à grande échelle et n'offrant pas d'aussi bons résultats, mais infiniment plus difficile à détecter.

Pour voir de quoi il s'agit, observons de plus près maintenant ce qui se passe exactement quand Alice surfe sur le Web, du point de vue de l'agent du gouvernement tyrannique qui la surveille, Eve, d'abord dans le cas où la connexion n'est pas chiffrée, puis dans celui où elle l'est. Nous verrons que si le contenu n'est plus lisible directement, l'« apparence » de la connexion ne change que peu.

---

[1] Questions :
- Savez-vous comment accéder à la liste des autorités de certification de votre navigateur ?
- Quand avez-vous regardé pour la dernière fois cette liste ? Combien d'autorités y avait-il ?
- Pour chaque autorité de cette liste, comment avez-vous décidé si *vous* deviez réellement lui faire confiance et s'il ne s'agissait pas d'une façade de la NSA ou de votre Mallory personnel ?
- (Question subsidiaire) Vous êtes-vous assuré que *toutes* les autorités auxquelles votre navigateur fait confiance étaient bien affichées dans la liste ? Comment ?

[2] Il s'agit de plus de l'option par défaut sur de nombreux navigateurs, qui y ajoutent parfois la possibilité de cocher une case « ne plus me poser la question » !

Navigation Web « normale ». Lorsque l'on tape l'adresse d'un site Web (non-sécurisé) dans son navigateur, il y a plusieurs phases vis-à-vis du réseau :

- l'adresse est résolue via le protocole DNS, ce qui permet d'obtenir l'adresse IP du serveur ;
- une connexion TCP/IP est établie avec cette IP, la page demandée est récupérée et la connexion terminée ;
- (éventuellement) l'étape précédente est répétée pour chaque élément de la page (images, autres frames, etc. — cf. figure 1.1).

Navigation Web « sécurisée » et attaque d'Eve. Lorsque l'on navigue sur le Web avec un protocole « sécurisé », ces étapes restent essentiellement les mêmes, la différence étant en général que le contenu des connexions avec l'IP du serveur n'est pas lisible.

Comme la couche cryptographique fournie par SSL / TLS se trouve au-dessus de TCP, il est possible de connaître :

- vers quelle adresse et quel port la connexion est établie : comme l'adresse du certificat du serveur doit correspondre à celle présente dans la barre de navigation du navigateur et que ce certificat est vérifié lors de l'établissement de la connexion SSL, il est rare d'avoir des *virtual hosts* comme avec les versions non sûres de HTTP, et le couple adresse / port identifie donc le serveur[3] ;
- de quel port elle provient : cela nous permet généralement de distinguer deux connexions différentes vers le même serveur sans même avoir à analyser le contenu SSL / TLS lui-même.

Sachant cela, Eve peut maintenant essayer de suivre plus en détail la navigation d'Alice sur ce site.

La figure 1.2 illustre les données de type "application data" reçues par le client lors du chargement de la page `https ://www.sstic.org`. Il est intéressant de comparer cette figure avec la figure 1.1.

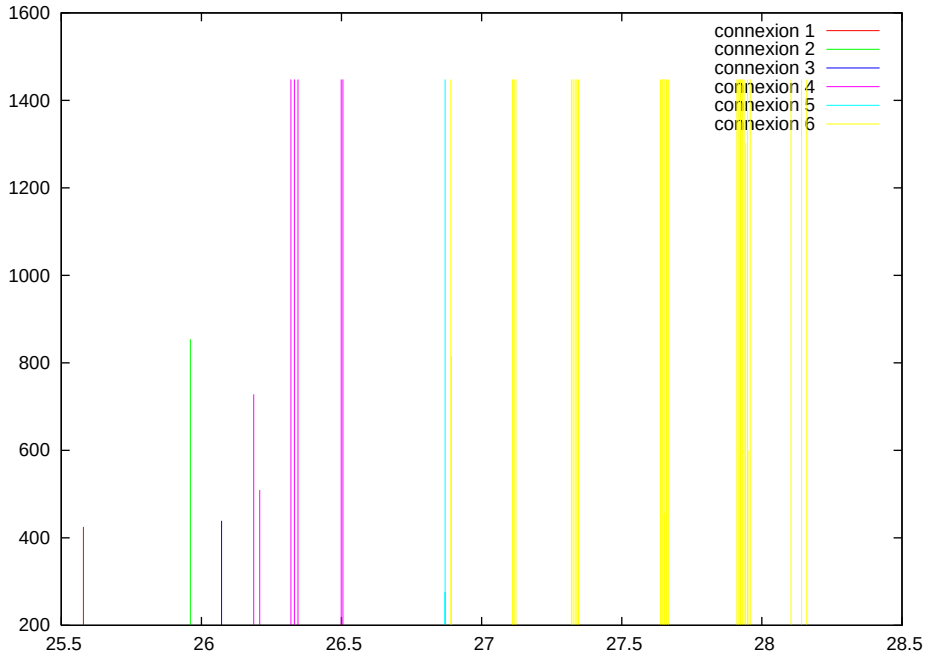
On constate en particulier que l'on observe le même nombre de connexions TCP que dans le cas où l'on n'utilisait pas de chiffrement. On constate également que chaque connexion TCP reçoit une quantité de données de type "application data" légèrement supérieure (quelques dizaines d'octets de plus) à la quantité de données dans les *payloads* TCP correspondantes.

Supposons qu'Eve voie qu'Alice fait une requête DNS pour connaître l'adresse de `www.sstic.org`, suivie de connexions en `https` vers ce site. Elle peut alors faire une copie (de la partie publique) du site pour mesurer la taille de chaque page. Elle a ainsi une liste de références pages-tailles (Table 1.1). Pour bien faire, elle devrait également faire un graphique indiquant comment les pages sont liées entre elles et avec les fichiers (images, feuilles de style, applets, etc.).

Pendant qu'Alice surfe sur ce site, Eve note (comme illustré par le tableau 1.2) pour chaque connexion TCP un triplet

$$\{\ \textit{timestamp début, timestamp fin, taille des données reçues}\ \}$$

[3] Pour avoir des *virtual hosts* dans ce contexte, il faut que le certificat comporte les noms de *tous* les hôtes en question : comme il faut changer le certificat à chaque ajout ou retrait d'un hôte, cela devient difficile à gérer, sans même parler des frais de certification. Pour l'attaque dont il est question, si jamais le certificat comportait un grand nombre d'hôtes, Eve pourrait facilement se rabattre sur l'observation du message DNS demandant l'adresse du serveur.



- La première connexion demande `http://www.sstic.org` (nos exemples prennent les données du site tel qu'il était en janvier 2006) ; le serveur répond que la page a été déplacée (de manière permanente, code 301) et se trouve à l'adresse `http://www.sstic.org/SSTIC06/` ;
- la deuxième connexion demande donc `http://www.sstic.org/SSTIC06/` ; le serveur répond que la page a été déplacée (de manière temporaire, code 302) et se trouve à l'adresse `http://www.sstic.org/SSTIC06/jsp/accueil.jsp` ;
- la troisième connexion demande donc `http://www.sstic.org/SSTIC06/jsp/accueil.jsp` ; le serveur répond à nouveau par une redirection permanente, vers `http ://www.sstic.org/SSTIC06/info.do` ;
- la quatrième connexion (qui demande `http://www.sstic.org/SSTIC06/info.do`) est la bonne, le serveur renvoie la page Web ;
- les connexions cinq et six chargent les images de cette page, `http://www.sstic.org/SSTIC06/static/images/logo_sstic_tranp.png` et `http://www.sstic.org/SSTIC06/static/images/bandeau_SSTIC_2006.jpg`.

FIG. 1.1 – Un accès vers la page `http ://www.sstic.org` avec le navigateur w3m. La figure représente les données reçues par le client du serveur en fonction du temps (*payload* TCP). On note que dans une même connexion les paquets arrivent par groupes (taille de la fenêtre TCP).

Afin d'y voir un peu plus clair, Eve peut faire un graphique correspondant à ces données (cf. Figure 1.3, page 8) : on voit que les connexions se font par rafales. Il est alors évident que chaque rafale correspond normalement au chargement d'une page. Le problème d'Eve est donc d'identifier chacune de ces pages.

Les trois premières connexions se succèdent et ne provoquent le chargement que de peu de données. Eve peut donc penser qu'il s'agit de messages d'erreur. Leur succession rapide indique qu'il n'y a pas d'intervention humaine, donc il

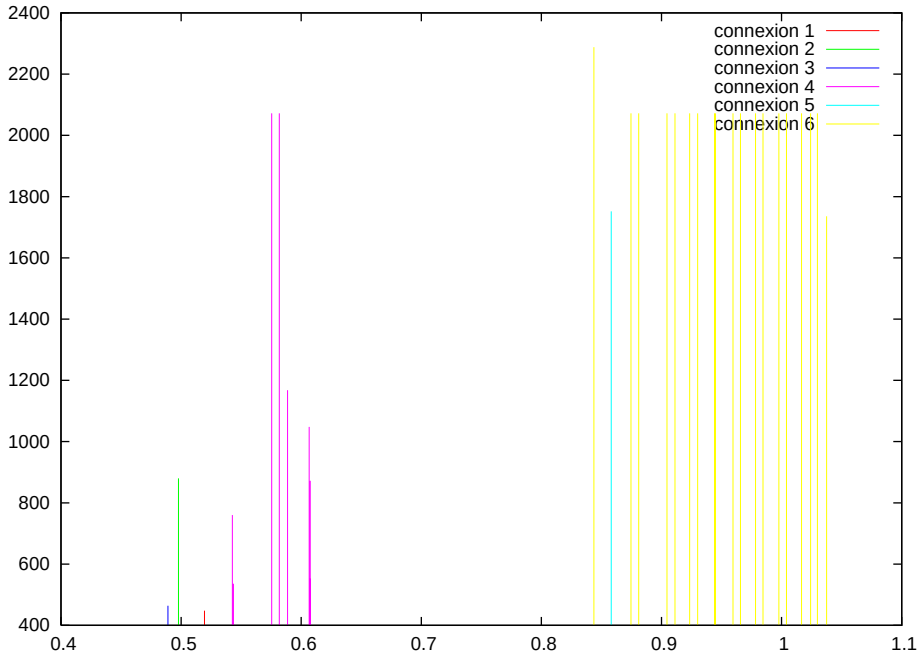


FIG. 1.2 – Un accès vers la page `https ://www.sstic.org` avec le navigateur w3m. La figure représente les données reçues par le client du serveur en fonction du temps (données SSL de type "application data"). Les différentes connexions correspondent à celles indiquées sur la figure 1.1.

s'agit probablement de redirections. La connexion 4 doit, elle, correspondre au chargement d'une page. Comme les connexions 5 et 6 suivent, on peut supposer qu'il s'agit d'éléments de cette page. 5 et 6 sont alors identifiables aux images `static/images/logo_sstic_transp.png` et `static/images/bandeau_SSTIC_2006.jpg` (les tailles correspondent, et ces images figurent sur toutes les pages du site). La page de la connexion 4 reste plus mystérieuse, car plusieurs fichiers pourraient correspondre. Néanmoins, les trois redirections qui précèdent laissent à penser qu'il s'agit de la page d'accueil, *info.do*.

La connexion 7 correspond au chargement d'une nouvelle page. On constate qu'aucune autre information n'est chargée, il s'agit donc d'une page qui, si elle contient des images, utilise celles de la page précédente, qui se trouvent dans le cache. La page dont la taille semble le mieux correspondre est *soumission_aide.do*.

La connexion 8 est, elle aussi, isolée. Les mêmes constatations que précédemment s'appliquent donc. La page dont la taille semble être la plus proche est *appel.do*, mais on ne peut exclure qu'il s'agisse de *comite.do*.

La connexion 9, par contre, provoque le chargement d'un certain nombre de fichiers. Du point de vue de la taille, la page *sponsors.do* correspond, et cette correspondance est confirmée par le fait qu'elle contient des images dont les tailles correspondent aux connexions 10 à 16.

Enfin la connexion 17. D'après la quantité de données transférées, on peut cette fois supposer qu'Alice consulte la page *inscription_aide.do*.

fichier	taille
FAQ.do	4868
appel.do	15469
comite.do	15016
connexion.do	39155
contact.do	8546
formulairePasswordOublie.do	38184
import.do	39031
info.do	8327
infos_pratiques.do	9576
inscription.do	49089
inscription_aide.do	6734
presse.do	6472
programme.do	22114
soumission_aide.do	7023
sponsors.do	10419
static/css/style.css	4396
static/js/util.js	194
static/images/arche.png	28408
static/images/bandeau_SSTIC_2006.jpg	38578
static/images/cea.png	1779
static/images/eads.png	1075
static/images/esat.gif	10592
static/images/ftrd.gif	2258
static/images/logo_sstic_transp.png	1512
static/images/misc.png	2721
static/images/plan_ESAT.jpg	93523
static/images/supelec.gif	4031

TAB. 1.1 – Les fichiers dans `http ://www.sstic.org/SSTIC06/` et leur taille. NB : la plupart des fichiers sont dynamiques et leur taille exacte peut varier selon le client. Il est donc important qu'Eve connaisse la configuration d'Alice pour récupérer les pages.

Portée de cette attaque et notes. On voit qu'avec des moyens très ordinaires (un PC avec tcpdump[4], ssldump [62] et/ou wireshark[5] convient très bien), un observateur peut suivre à la trace un utilisateur lorsque celui-ci navigue sur un site Web en utilisant `https`. Ce protocole n'est donc pas suffisant pour garantir la vie privée des utilisateurs.

Cependant, si une telle analyse de trafic est simple, réalisable par pratiquement n'importe qui, elle ne permet pas de tout découvrir. En effet, elle se base sur le fait que l'observateur peut à son tour accéder aux pages du site pour mesurer leur taille. Si par contre Alice accède à une section privée où Eve ne peut la suivre, alors cette dernière ne pourra savoir ce qu'elle y fait. De même, les informations qu'Alice envoie au site (un éventuel numéro de carte bancaire, un login et un mot de passe, etc) ne sont pas à la portée d'Eve.

[4] http ://www.tcpdump.org
[5] http ://www.wireshark.org

Numéro	début	fin	données reçûes
1	0.8972	1.9518	448
2	2.1943	3.2346	880
3	3.4399	4.4096	464
4	4.6016	5.8224	9080
5	6.0427	7.3877	1752
6	6.0437	8.3833	39248
7	131.9430	143.2731	7224
8	173.3688	174.5648	16048
9	256.5647	258.1510	10952
10	258.4016	260.1926	2984
11	258.4025	260.6357	10952
12	258.4033	260.1306	2016
13	258.4035	261.0380	28960
14	260.4091	261.6079	2520
15	260.4549	261.6785	4296
16	260.8955	261.8787	1312
17	281.6748	282.8524	6816

TAB. 1.2 – Les connexions enregistrées par Eve pendant qu'Alice navigue sur le site `https ://www.sstic.org`.

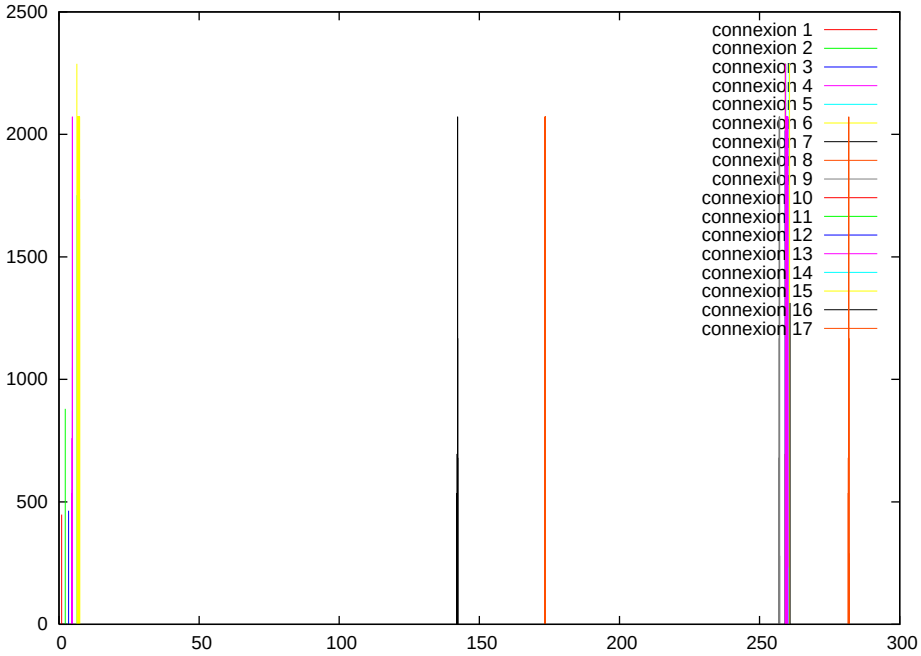


FIG. 1.3 – Les paquets avec des données (type SSL "application data") correspondant aux connexions de la table 1.2. On note le groupement des connexions par rafales.

Il faut noter que le navigateur utilisé dans ces exemples, `w3m`, est un navigateur qui rend l'attaque particulièrement simple car il n'est pas très sophistiqué. Des mécanismes utilisés dans d'autres navigateurs et destinés à accélérer la navigation (cache, préchargement de pages, certaines fonctionnalités de HTTP 1.1 comme le *keep-alive* et le *pipelining*) peuvent compliquer (ou, dans certains cas, faciliter) la tâche de l'attaquant. Nous sommes néanmoins persuadés que cette attaque reste non seulement réalisable, mais est de surcroît automatisable. Il se peut qu'elle nécessite des modèles (peut-être probabilistes) précis des caractéristiques de chaque navigateur, à paramétrer selon la configuration de l'utilisateur observé. Nous n'approfondirons pas plus : il y aurait sans doute matière à écrire une thèse entière sur le sujet.

La nécessité de garder un « contexte » (paramètres et état du cache du navigateur de l'utilisateur observé) peut gêner le déploiement d'un tel système de surveillance à grande échelle, mais non l'observation de quelques personnes seulement.

Upgrade to TLS

Pour terminer cette section sur `https`, TLS et SSL, notons qu'il existe également une option « upgrade to TLS » qui permet à un client de commencer une session en `http` et de la passer en `https` au bout d'un certain temps. Cela ne change bien sûr rien en ce qui concerne la sécurité.

1.1.2 Les autres protocoles

Nous ne nous attarderons pas sur les autres protocoles de « sécurisation » ; les problèmes et les attaques sont en gros les mêmes que pour SSL/TLS.

Dans le cas d'IPsec[6], la tâche de l'attaquant est compliquée par le fait qu'il ne connaît pas a priori le type de trafic qui va passer sur la connexion, le numéro de port étant dissimulé (alors qu'une connexion chiffrée avec TLS/SSL sur le port 443 sera pratiquement toujours utilisée pour du trafic HTTP). Il peut toutefois obtenir des indices, par exemple en observant le trafic DNS du client : la convention voulant que les machines aient des noms en rapport avec leur fonction (`www` pour un serveur Web, `ftp` pour un serveur ftp, `smtp` pour un serveur de mails, etc.) lui donnera de précieux indices. Il peut également faire un graphe du trafic au cours du temps : celui-ci a souvent une « forme » spécifique d'un type de trafic. Il existe des outils qui analysent ainsi l'aspect des flux chiffrés, comme `flOp` [83].

Le problème est similaire pour les échanges de mails. Les systèmes de chiffrement, comme OpenPGP ou S/MIME, dissimulent le contenu du message mais pas les en-têtes. Ceux-ci comportent des informations comme l'adresse de l'expéditeur, celle du destinataire et l'objet du message. La principale différence réside dans le fait que dans le cas des emails, il existe déjà des solutions satisfaisantes, comme nous le verrons dans la section 1.2.

[6] Nous faisons bien sûr ici allusion à IPsec utilisé en mode hôte-à-hôte, en non en mode tunnel / VPN.

1.1.3 NAT et proxies

Bien qu'il ne s'agisse pas à l'origine de technologies spécifiques à la sécurité, on rencontre régulièrement des NATs et des proxies. L'un et l'autre ont pour effet de masquer l'adresse du client au serveur. Il est donc intéressant de savoir s'ils fournissent un réel anonymat.

NAT

Les mécanismes de traduction d'adresses (abrégé en NAT, pour *Network Address Translation*) ont été à l'origine créés pour faire face à la pénurie d'adresses IPv4. Dans la pratique, cela signifie que plusieurs machines disposent d'adresses privées (*i.e.*, non accessibles directement depuis l'Internet) et qu'un routeur spécial (appelé couramment « routeur NAT »), qui possède lui une adresse publique, modifie les paquets que ces machines envoient de manière à ce que le destinataire ait l'impression qu'il en est à l'origine. Il effectue ensuite l'action inverse quand il reçoit la réponse.

Le résultat concret est que les connexions émanant de toutes les machines d'un réseau derrière un NAT semblent provenir de la même adresse ; un observateur ne peut donc plus trier les connexions en fonction de l'adresse source.

Pourtant, même en supposant que le routeur NAT lui-même soit sûr, il ne faut généralement pas compter sur ce type de mécanisme pour améliorer la protection de la vie privée. En effet, même si l'adresse d'origine est identique, il reste en général possible de distinguer des machines, souvent de manière entièrement passive pour l'attaquant, en se basant par exemple sur de légères différences dans les piles IP qui ne sont pas masquées par le processus de traduction d'adresses. Le lecteur désireux d'en savoir plus sur ces méthodes pourra consulter [82], tandis que celui plus intéressé par une démonstration pratique s'intéressera par exemple à `p0f` [81].

Proxies

Les proxies[7] sont pour leur part généralement situés au niveau applicatif. Ils ont des rôles divers, les plus courants étant ceux de cache (par exemple pour accélérer la navigation sur le Web) et de *firewall applicatif*. Ils sont en général propres à un protocole particulier ; quand ils transmettent une requête, la partie « réseau » (c'est-à-dire ce qui n'est pas au niveau applicatif) au moins est réécrite.

La possibilité pour l'attaquant de distinguer les clients dépendra donc en grande partie de ce que fait le proxy exactement, ce qui est très variable. Il est à noter qu'il faut généralement choisir entre l'utilisation d'un proxy et celle d'un protocole de sécurisation comme `https`.

Il est possible qu'Alice surfe sur le Web de manière anonyme (k-anonymat parmi les utilisateurs de ce proxy) en utilisant le protocole http de manière non sécurisée, puis, qu'un site passant en `https` (par exemple pour transmettre un numéro de carte bancaire) son anonymat antérieur soit perdu, l'observateur pouvant faire le lien entre les connexions non-sécurisées du proxy vers le site

[7] On désigne parfois en français un proxy par le vocable de *serveur mandataire*. Nous n'utiliserons pas ce terme pour ne pas alourdir le texte.

de Bob, et la connexion d'un utilisateur U vers le même site de Bob via une connexion sécurisée (cette dernière ne passant pas par le proxy).

Bien entendu, dans le cas du NAT comme dans celui du proxy, la machine qui joue ce rôle voit passer toutes les communications : il faut donc qu'elle soit inaccessible aux espions. Il y a alors un problème de confiance : Alice fera-t-elle confiance au proxy si elle ne le contrôle pas ? Et si elle le contrôle, pourquoi un autre utilisateur (Alfred par exemple) lui ferait-il confiance ? Et bien sûr, il ne faut pas qu'Eve puisse observer les communications entre le proxy ou le NAT et Alice : il n'y a aucune protection à cet endroit.

1.2 Anonymat des communications à forte latence

Avant de revenir aux problèmes de vie privée pour les communications interactives, il est intéressant de faire une parenthèse sur les communications non-interactives, dont l'email est un exemple.

1.2.1 Mix-nets et remailers

Les *remailers anonymes* sont des systèmes permettant d'envoyer des emails ou des messages Usenet de manière non seulement anonyme vis-à-vis de leurs destinataires, mais également, pour les plus récents, de manière non-observable.

Type 0

L'un des premiers remailers (type 0, 1993) était un système finlandais, `penet`, opéré par Julf Helsingius, qui permettait tout simplement à ses utilisateurs d'envoyer des mails en indiquant comme expéditeur une adresse sur cette machine. Le destinataire pouvait répondre à cette adresse et les messages étaient alors transmis à l'expéditeur original. Cela impliquait néanmoins la conservation d'une table de correspondance dans cette machine entre les *nymes*[8] des utilisateurs et leur adresse email réelle. L'histoire finit comme on peut le supposer : en 1995 une entité (en l'occurrence l'Église de Scientologie) n'a pas apprécié certaines des informations à son sujet divulguées par l'un des utilisateurs de ce remailer et a attaqué son opérateur en justice ; la *police* a obligé celui-ci à révéler l'adresse des personnes incriminées.

Type 1

Cela montre, s'il le fallait, la nécessité que l'anonymat ne dépende pas d'un unique point. Cependant, parallèlement, certaines des idées présentées par Chaum dès 1981 [19] avaient été mises en pratique, aboutissant aux remailers dits *cypherpunks*, ou type 1.

Un tel protocole est basé sur le reroutage des messages : l'expéditeur envoie un message au premier de ces routeurs (remailer) ; celui-ci déchiffre le message (qui était chiffré avec sa clef publique) et découvre un autre message ainsi qu'une adresse. Il renvoie alors le message à cette adresse, qui est généralement celle d'un autre remailer, qui fera de même. Finalement, après un certain nombre de

[8] Un *nyme* est, d'une certaine façon, la « partie technique » d'un pseudonyme, qui permet de s'assurer que deux messages viennent d'une même personne, sans révêler pour autant l'identité de celle-ci. Il s'agit généralement d'une paire de clefs (privée et publique).

passages par différents remailers, l'adresse déchiffrée sera celle du vrai destinataire du message (Figure 1.4).

$$
\begin{array}{rcl}
\text{Alice} \rightarrow R_1 & : & \left\{ R_2, \left\{ R_3, \left\{ \ldots, \left\{ Bob, \{M\}_{pk_B} \right\}_{pk_{R_n}} \ldots \right\}_{pk_{R_\ldots}} \right\}_{pk_{R_2}} \right\}_{pk_{R_1}} \\
R_1 \rightarrow R_2 & : & \left\{ R_3, \left\{ \ldots, \left\{ Bob, \{M\}_{pk_B} \right\}_{pk_{R_n}} \ldots \right\}_{pk_{R_\ldots}} \right\}_{pk_{R_2}} \\
R_2 \rightarrow R_3 & : & \left\{ \ldots, \left\{ Bob, \{M\}_{pk_B} \right\}_{pk_{R_n}} \ldots \right\}_{pk_{R_\ldots}} \\
\vdots & & \\
R_{n-1} \rightarrow R_n & : & \left\{ Bob, \{M\}_{pk_B} \right\}_{pk_{R_n}} \\
R_n \rightarrow Bob & : & \{M\}_{pk_B}
\end{array}
$$

FIG. 1.4 – Alice envoie un message M à Bob à travers une chaîne de remailers.

Il y a néanmoins des attaques possibles : si un tel système donne un bon anonymat (il faut pouvoir remonter toute la chaîne pour parvenir à l'expéditeur d'un message), un adversaire puissant, qui serait capable de surveiller les liens réseaux des remailers, pourrait casser le système simplement en observant les messages entrer puis ressortir quelques instants plus tard de chaque remailer. Bien que le message ait changé (une couche de chiffrement a été enlevée) l'observateur est encore capable de le reconnaître (par sa taille, par le moment auquel il ressort si le trafic est faible, etc.). Ceci n'est que la plus simple des attaques possibles, et il en existe un certain nombre d'autres dans lesquelles l'attaquant peut être passif comme ici, mais aussi actif (insertion de messages, etc.). Le lecteur intéressé pourra se reporter à [23].

Type 2

Pour pallier ces attaques, le développement du protocole *mixmaster* a commencé en 1996. Les changements les plus notables sont l'utilisation d'un temps d'attente au niveau de chaque remailer et le fait qu'un padding soit utilisé pour que tous les messages aient une taille similaire.

Chaque routeur ne renvoie pas immédiatement les messages qu'il reçoit : il attend d'en avoir reçu un certain nombre pour les envoyer alors dans un ordre différent et aléatoire. Comme tous les messages ont la même taille (un padding est utilisé) et que le message entrant ne ressemble pas au message sortant (une "couche" de chiffrement a été enlevée), un observateur ne peut donc pas déterminer la correspondance entre les messages entrants et les messages sortants. Si on ajoute le fait qu'un message passe généralement par une chaîne de remailers, on voit que l'observateur n'a aucun moyen de déterminer où il aboutit. En fait, il ne peut même pas déterminer *quand* un utilisateur envoie réellement des messages car la possibilité d'envoyer de faux messages (des messages "vides" qui se perdent dans le réseau de remailers) existe.

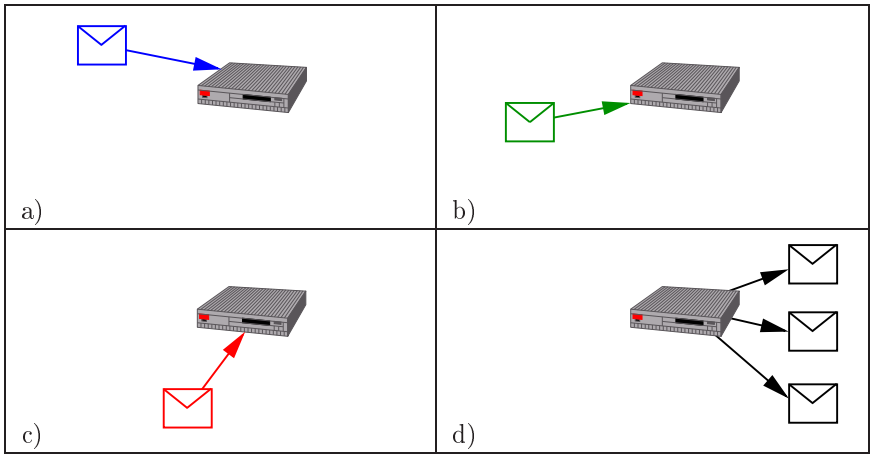


FIG. 1.5 – Un nœud mixmaster reçoit un certain nombre de messages (a,b,c) puis les renvoie (d) simultanément (du moins dans un ordre indépendant de l'ordre d'arrivée) vers le nœud suivant après avoir enlevé une "couche" de chiffrement.

L'un des points essentiels de ce système est le temps d'attente sur chaque remailer, où l'on attend la présence d'un nombre suffisant d'autres messages. Cela fait qu'un message envoyé en utilisant mixmaster peut facilement mettre quelques dizaines d'heures avant d'atteindre sa destination. De telles attentes sont bien sûr impossibles pour des communications interactives...

Une nouvelle version de *mixmaster* est actuellement en cours de standardisation par l'Internet Engineering Task Force (IETF) [55].

Type 3

Le défaut des types 1 et 2 par rapport au remailer de Helsingius (et qui explique le succès de celui-ci alors même qu'un réseau de remailers de type 1 était déjà déployé) est qu'ils ne permettent pas au destinataire de répondre. Le projet *mixminion*, actuellement en développement actif, devrait pallier ce problème avec ce qui sera le type 3 de remailer.

Il faut cependant noter qu'il existe un moyen d'avoir des échanges anonymes dans les deux sens (*i.e.*, avec une personne dont on ne connaît pas l'adresse mail) avec les types 1 et 2 de remailers : il suffit de poster les messages anonymes dans un groupe usenet convenu, par exemple `alt.anonymous.messages`, avec un sujet indiquant le destinataire. Avec l'utilisation de remailers de type 2, un tel schéma est inobservable si le destinataire prend soin de télécharger systématiquement tous les messages du forum en question et pas uniquement ceux qui lui sont destinés.

1.2.2 DC-nets

David Chaum est également à l'origine des DC-nets [18], et le plus simple est probablement de reprendre [une traduction de] ses propres mots pour introduire le sujet :

> « Trois cryptographes dînent, assis dans leur restaurant trois étoiles

favori. Leur serveur les informe que des arrangements ont été pris avec le maître d'hôtel pour que la note soit réglée, de manière anonyme. Ce peut être l'un des cryptographes qui paie, ou ce peut être la NSA (l'agence de sécurité nationale américaine). Les trois cryptographes respectent leur droit mutuel de payer anonymement, mais ils aimeraient savoir si c'est la NSA qui paie. Ils résolvent leur incertitude honnêtement en utilisant le protocole suivant.

Chaque cryptographe lance une pièce non biaisée derrière son menu, entre le cryptographe à sa droite et lui-même, de manière à ce qu'ils soient les seuls à voir le résultat. Chaque cryptographe indique ensuite à haute voix si les deux pièces qu'il peut voir — celle qu'il a lancée et celle lancée par son voisin de gauche — sont tombées sur la même face ou non. Si l'un des cryptographe a payé, il indique l'inverse de ce qu'il voit[9]. Un nombre impair de différences dans ce qui est annoncé indique que l'un des cryptographes paie ; un nombre pair indique que c'est la NSA qui paie (en supposant que le dîner ne soit payé qu'une seule fois). Pourtant, si l'un des cryptographes paie, aucun des deux autres n'apprend de quel cryptographe il s'agit. »

Le lecteur aura probablement deviné à présent que *DC* signifie *dining cryptographers*. Ce protocole peut se généraliser à un nombre supérieur à trois participants, et l'information transmise peut être un bit quelconque (dans l'exemple « j'ai payé »). Si l'on imagine que l'on lance plus d'une pièce à chaque fois, il est possible de transmettre plus d'un bit par *tour de table*.

L'intérêt des DC-nets est qu'ils sont plus sûrs que les mix-nets. Ils posent malheureusement de nombreux problèmes pratiques (passage à l'échelle, déni de service, fiabilité, latence, etc.), notamment du fait de leur fonctionnement en anneau, qui font qu'ils ne sont pas utilisés en pratique à notre connaissance.

1.3 Anonymat des communications à faible latence

En ce qui concerne les communications interactives, les choses sont moins simples :

- la communication doit être bidirectionnelle, au moins pour des questions de contrôle de trafic ;
- la communication doit être établie entre les deux parties qui communiquent, par opposition par exemple au cas des remailers où une connexion est faite vers le premier de la chaîne, le message est envoyé, la connexion fermée, puis le remailer itère le processus vers le suivant de la chaîne.

La solution la plus simple peut être vue comme l'équivalent du remailer de type 0 (cf. 1.2.1) : il s'agit d'un proxy auquel le client peut se connecter par une liaison chiffrée (en général avec SSL / TLS). C'est le principe utilisé par la plupart des offres commerciales d'anonymat disponibles de nos jours, tels qu'Anonymizer.com [10]. Les problèmes des proxies et de SSL / TLS que nous

[9] Mais s'il n'a pas payé, il indique ce qu'il voit réellement (NDT).

[10] Cf. `http://www.anonymizer.com`. Le proxy ne se contente toutefois pas de rediriger les connexions mais sert aussi de filtre pour déjouer des attaques (non liées à l'anonymat) contre

avons évoqués précédemment (respectivement en 1.1.3 et 1.1.1) s'appliquent pleinement.

Il existe toutefois des solutions plus évoluées.

1.3.1 Onion-Routing

Pour ce qui est de l'anonymat, un système similaire à celui des *mix-net* des remailers, appelé *Onion-Routing*[11] [60], est généralement utilisé, à ceci près que l'on ne transmet pas par leur intermédiaire *un* message, mais un certain nombre de paquets qui sont chiffrés individuellement. Il s'agit en fait de tunnels réseaux chiffrés : l'émetteur établit un tunnel vers une première machine, puis, en passant par ce tunnel en crée un autre vers une seconde machine, et ainsi de suite jusqu'au destinataire (cf. figure 1.6). On pourrait à la limite imaginer un système d'Onion-Routing basique qui fonctionnerait en utilisant des tunnels `ssh` imbriqués. Notons d'ailleurs que si le terme « Onion-Routing » s'est imposé, l'idée avait déjà été décrite en 1995 sous le nom de *Pipe-net* [29], ce que l'on pourrait traduire par « réseau de tunnels ».

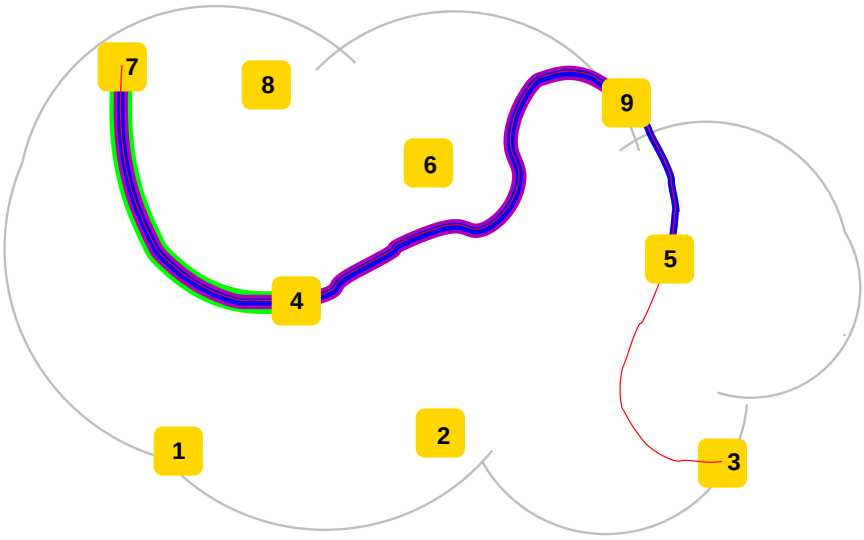


FIG. 1.6 – L'*Onion-Routing* : Alice (nœud 7) établit une série de tunnels imbriqués jusqu'à Bob (nœud 3).

Un tel système offre ce que nous appellerons de l'*anonymat à sens unique* : le destinataire de la communication ne peut déterminer qui l'appelle, mais l'appelant doit connaître l'adresse réseau du destinataire, ce qui revient généralement à connaître son identité. Un tel système est vulnérable à des attaques telles que celles évoquées précédemment face à un adversaire puissant, qui peut alors déterminer qui appelle qui : ainsi, sur la figure 1.6, on « voit » immédiatement qu'il y a une communication entre le nœud 7 et le nœud 3. Bien entendu, dans

l'utilisateur par des sites Web malintentionnés.

[11] L'analogie avec des oignons étant que chaque paquet transmis par un tel système est chiffré de multiples fois et que chaque routeur "pèle" une couche de chiffrement...

la réalité, il n'y a pas une unique communication sur le réseau, et la tâche de l'observateur est un peu moins aisée, mais l'idée est là.

L'Onion-Routing dégrade les performances par rapport à une connexion classique, en effet :

- le temps d'établissement d'une communication est long s'il faut établir tous les tunnels pour celle-ci ;
- le débit entre Alice et Bob est limité par le débit plus faible entre deux nœuds de la route qui les relie (ou par le moins puissant des nœuds, si les performances sont limitées par les machines et non par le réseau) ;
- la latence (définie ici comme le temps que met un paquet pour aller d'Alice à Bob et retour) correspond à la somme des latences entre les nœuds.

Pour améliorer le « mélange » entre les communications de plusieurs utilisateurs et gêner un observateur, on utilise parfois une *cascade* : il s'agit d'un tunnel qui utilise l'Onion-Routing et que peuvent utiliser plusieurs clients (cf. figure 1.7).

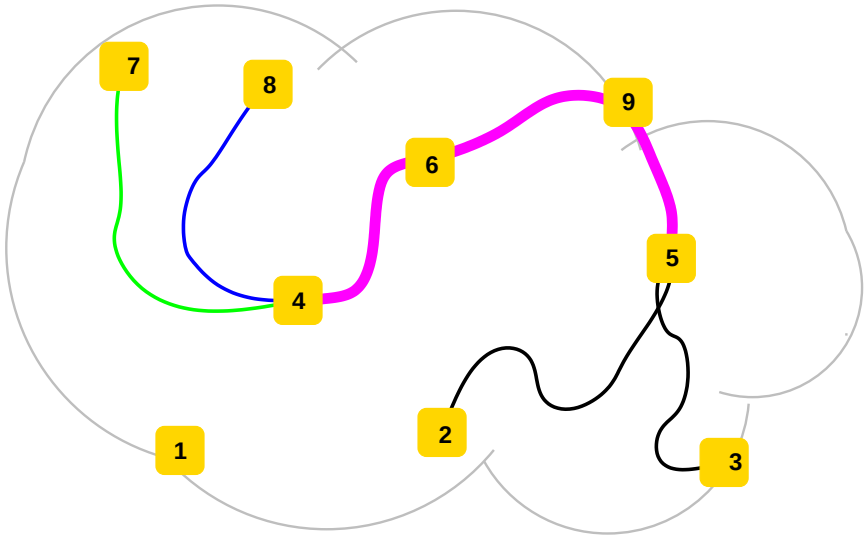


FIG. 1.7 – Les clients (nœuds 7 et 8) utilisent une cascade entre les nœuds 4 et 5. Les nœuds destinataires (2 et 3) ne savent pas qui se connecte à eux et les nœuds 5, 6 et 9 qui font partie de la cascade non plus. Un observateur a plus de mal à lier les connexions sortant du nœud 5 avec celles qui entrent au nœud 4.

Il existe plusieurs systèmes basés sur l'Onion-Routing décrits dans la littérature, reposant sur différentes architectures de réseaux sous-jacentes : ainsi, par exemple, *Tarzan* se situe au niveau IP et utilise une architecture pair-à-pair tandis que *Tor* utilise une architecture plus classique de nœuds "fiables" et se trouve au niveau de TCP.

Un observateur situé dans la position décrite par la figure 1, ne peut savoir directement où vont les connexions qui émanent d'Alice. Par contre, s'il l'apprend d'une autre source (par exemple parce que l'Onion-Routing protège les connexions TCP mais pas le DNS), il peut tenter de suivre de manière plus fine chaque connexion, à la manière de ce que nous décrivons en 1.1.1 avec

SSL / TLS.

Tarzan

Bien que plus ancien (2002) que la plupart des implémentations de l'Onion-Routing, Tarzan [33] est l'un des modèles interactifs utilisant le plus de protections contre l'analyse de trafic. Il s'agit d'un Onion-Routing pair-à-pair au niveau IP, avec des *mimiques* invariantes entre des paires de nœuds destinées à contrer l'analyse de trafic.

Le réseau de Tarzan est pair-à-pair, chaque nœud dispose d'un ensemble de mimiques qui le lient avec des voisins : les mimiques sont un invariant de trafic entre deux nœuds et restent les mêmes pour un observateur extérieur qu'il y ait du trafic réel ou non sur ce lien. Ces liens entre deux nœuds sont chiffrés.

Les communications sont établies de nœud en nœud, les routes suivant les mimiques qui existent. Les paquets des communications sont *paddés* de nœuds en nœuds pour être conformes à ce que requiert la mimique du lien.

À notre connaissance, Tarzan n'a jamais dépassé le stade du prototype expérimental.

Les principaux problèmes de Tarzan sont les suivants :
- les protections fonctionnent mal contre des attaquants internes au réseau. En effet, chaque nœud ajoute et enlève du padding pour se conformer aux mimiques d'entrée et de sortie : il peut donc connaître la taille des données ;
- il y a des problèmes de performances dus aux mimiques : comme les liens d'un réseau, la mimique qui offre le plus petit débit (que ce soit parce que son débit est réellement plus faible ou parce qu'elle est saturée par d'autres communications) est le facteur limitant ;
- le système passe mal à l'échelle car chaque nœud doit connaître l'intégralité de la topologie du réseau.

Tor

Le but principal de Tor [32] est d'offrir de l'anonymat pour des connexions TCP. Présenté comme de l'Onion-Routing de « seconde génération », Tor est avant tout une implémentation de l'Onion-Routing qui résout les problèmes pratiques, sans améliorations significatives en termes de sécurité. La principale nouveauté est sans doute la possibilité de protéger non seulement les clients mais aussi les serveurs, par l'introduction de points de rendez-vous (voir ci-dessous).

L'architecture du réseau est constituée de deux niveaux : les routeurs constituent un graphe complet qui forme un réseau central, auquel peuvent se connecter des clients. Le logiciel client offre en particulier une interface SOCKS, ce qui permet à un grand nombre d'applications de l'utiliser via un paramétrage en *proxy*, donc sans modifications.

Lorsqu'un client se connecte au réseau, il reçoit une liste de routeurs du réseau central. Cette liste est déterminée en utilisant un algorithme de vote, pour éviter qu'un routeur n'indique que des complices.

Tor n'utilise cependant aucune protection contre l'analyse de trafic autre que des paquets de taille identique.

Des services cachés et des communications anonymes dans les deux sens sont possibles grâces à des *points de rendez-vous*. Un point de rendez-vous est un nœud vers lequel à la fois Alice et Bob établissent une connexion anonyme et qui les met en relation.

La principale difficulté est pour Alice et Bob de se mettre d'accord de manière anonyme sur un point de rendez-vous. Dans Tor, les points de rendez-vous sont établis d'une manière élégante : Alice établit une communication avec un nœud qu'elle choisit puis envoie les paramètres pour établir la connexion avec Bob en les diffusant sur une table de hachage distribuée (DHT) qui les route vers Bob.

Ce système ne fournit néanmoins pas la non-observabilité et il nous semble qu'un attaquant peut découvrir le nyme d'un utilisateur particulier ; en effet, étant donné que les utilisateurs ne font pas partie du réseau de reroutage, l'absence de faux trafic peut permettre d'envoyer une demande d'établissement de communication à chaque utilisateur et de voir quand celui que l'on surveille reçoit des paquets. Un observateur global peut également suivre le cheminement de ces demandes dans la DHT.

Points forts et faiblesses. Tor compte un certain nombre de points forts, mais aussi des faiblesses[12].

Dans les points forts de Tor, on compte :

- réellement déployé (environ 800 nœuds centraux et « plusieurs centaines de milliers » d'utilisateurs début 2008 [21]) ;
- utilisation d'une table de hachage distribuée pour établir les points de rendez-vous : c'est une solution élégante (dans un contexte d'anonymat) ;
- utilisation d'un algorithme de vote pour éviter qu'un nœud n'indique que des complices.

Dans ses points faibles :

- problèmes de passage à l'échelle : d'une part les nœuds centraux établissent un graphe complet, d'autre part le nombre des nœuds centraux — et donc les ressources disponibles pour l'acheminement des communications — évolue d'une manière indépendante du nombre de clients.
- il y a une distinction entre les clients et les nœuds qui servent pour l'Onion-Routing ;
- ne fonctionne pas pour les protocoles utilisant UDP ;
- pas de protection contre une analyse de trafic évoluée (autre que sur les adresses sources et destination) ;
- destiné à l'anonymat (face au destinataire de la connexion), pas à la non-observabilité (face à un observateur extérieur à la connexion) ;

Influence sur nos travaux. Tor a eu une influence notable sur le début de nos travaux, des brouillons de [32] étant disponibles fin 2003, c'est-à-dire quelques mois à peine après le début de notre propre recherche, alors dans le cadre de notre master. Notre positionnement initial était assez proche de Tor et

[12] Notez que ce que nous appelons ici « points faibles » en sont selon *notre perspective*, par rapport à *nos buts* (non-observabilité), mais pas forcément par rapport aux objectifs (anonymat) des auteurs de Tor ; ceux-ci justifient d'ailleurs leurs choix dans la FAQ de Tor [21].

la publication de celui-ci nous a poussé à nous recentrer sur notre thème actuel, c'est-à-dire à tenter de fournir une protection plus forte, contre un observateur plus puissant, sans sacrifier pour autant les performances. Diverses contraintes imposées par ces nouveaux critères font que nous avons abouti à un résultat plus éloigné de Tor que dans notre modèle initial (et qui se rapproche, par certains côtés, plus de Tarzan), comme nous allons le voir dans la première partie.

I2P

I2P, pour *Invisible Internet Project* [39], est un système pair-à-pair qui fonctionne au-dessus d'UDP. Son développement a commencé selon un modèle *Open Source* à la même époque que celui de Tor ou de notre propre *True Nyms*, décrit dans la partie 1.

I2P utilise le *Garlic Routing*, une extension de l'Onion-Routing qui permet de chiffrer simultanément plusieurs messages, afin d'offrir une meilleure protection contre l'analyse de trafic.

I2P est encore très expérimental et en développement constant : il est donc difficile de connaître son fonctionnement précis à un moment donné sans examiner le code source. Nous noterons cependant qu'il semble utiliser des routes différentes pour l'émission et la réception, et que, comme notre propre proposition, il utilise de multiples routes pour une même communication et des paquets leurres.

Ce système offre une API spécifique et non un proxy SOCKS générique, ce qui explique probablement en partie sa diffusion inférieure à celle de Tor. Nous ne l'avons d'ailleurs nous-même découvert que tardivement.

1.3.2 Crowds

Crowds [61] est un système conçu pour anonymiser l'accès au Web. Ce système est constitué d'un ensemble de nœuds. Quand un nœud veut faire une requête, il envoie celle-ci à un autre nœud. Celui-ci lance alors un dé, et :

- avec une probabilité p, il transmet la requête à un autre nœud du réseau ;
- avec une probabilité $1 - p$, il interroge le serveur Web.

Les réponses sont transmises suivant le chemin inverse des requêtes. Un nœud ne sait pas si une requête provient du nœud qui semble l'avoir envoyée, ou si celui-ci ne faisait que la transmettre.

Comparaison avec l'Onion-Routing

Le but de Crowds est de fournir l'anonymat vis-à-vis du serveur. Comme dans l'Onion-Routing, un nœud ne peut savoir de manière simple si un message qu'il reçoit a été envoyé par le nœud précédent ou non. Par contre, il connaît cette fois le contenu de la requête.

Les échanges entre les nœuds du réseau ne sont pas chiffrés et aucune protection contre une analyse de trafic par corrélation des entrées et sorties d'un nœud n'est offerte : un observateur global n'a aucun mal à suivre les requêtes, et un observateur local peut toujours espionner Alice.

Une attaque statistique basée sur le fait que la plupart des requêtes Web sont propres à un utilisateur et sur le fait qu'un même utilisateur fera la même requête régulièrement permet à un [ensemble de] nœud[s] du réseau de déterminer l'origine d'une requête : c'est l'*attaque des prédécesseurs*.

Crowds offre donc a priori moins de protection que l'Onion-Routing.

1.3.3 Attaques

Nous ne décrivons pas ici les différentes attaques connues contre les systèmes d'anonymat à faible latence dont le nôtre devra se prémunir. En effet, les attaques à considérer dépendent la plupart du temps de petits détails de conception du système et / ou leur description nécessite d'expliquer des points techniques qui doivent être présentés dans leur contexte. Nous préférons donc :

- soit parler d'une certaine attaque quand, par la suite, nous décrirons une mesure destinée à s'en protéger ;
- soit les expliquer dans les chapitres 6 et 13 où nous analyserons la sécurité de nos propositions.

Première partie

True Nyms : vers des communications non-observables à faible latence

Computer technology is on the verge of providing the ability for individuals and groups to communicate and interact with each other in a totally anonymous manner. Two persons may exchange messages, conduct business, and negotiate electronic contracts without ever knowing the True Name, or legal identity, of the other. Interactions over networks will be untraceable, via extensive re-routing of encrypted packets and tamper-proof boxes which implement cryptographic protocols with nearly perfect assurance against any tampering.

Timothy C. May, *The Crypto Anarchist Manifesto*, 1988

2 Introduction et vue globale

Les travaux présentés dans cette partie sont la suite directe de ce que nous avions commencé dans notre mémoire de master [8] et la plupart des idées ici présentées sont le développement de celles du chapitre deux de [8].

Ils consistent principalement à renforcer l'Onion-Routing pour empêcher l'analyse de trafic, ce qui implique de résoudre un certain nombre de problèmes (voir la section 2.2, ci-dessous).

Nous avons donc différentes propositions en ce sens, qui sont décrites dans les chapitres qui suivent. Ces propositions sont ensuite utilisées par un protocole, *True Nyms*. Nous avons implémenté ce protocole, puis évalué l'implémentation. Dans la suite de cette partie, nous nous référons souvent de la même manière au protocole et à l'implémentation ; cette dernière sera toutefois notée `truenyms` lorsqu'une unique notation peut prêter à confusion ou que nous voulons distinguer explicitement les deux.

2.1 Un point de vocabulaire

Avant d'entrer plus avant dans notre sujet, nous aimerions préciser un point de vocabulaire :

- par *connexion*, nous désignons la liaison entre Alice et un nœud d'une route. La connexion$_1$ avec le premier nœud de la route s'effectue directement entre Alice (nœud N_0) et le nœud N_1, tandis que les connexions suivantes sont créées récursivement, chacune au travers de la précédente : la connexion$_i$ (entre Alice et le nœud N_i) passe par la connexion$_{i-1}$;
- par *route* vers un nœud N_n, nous désignons une sorte de « méta-connexion » constituée de la connexion$_n$ entre Alice et ce nœud et de toutes celles au travers desquelles la connexion$_n$ est établie. La route vers un nœud N_n est donc l'ensemble de connexions $\{\text{connexion}_1, \ldots, \text{connexion}_n\}$. Nous appelons *longueur* d'une route le nombre de connexions qui en font partie ;

– par *communication*, nous désignons la transmission de bout en bout, c'est-à-dire entre Alice et Bob. Une communication peut être constituée d'une ou plusieurs routes, de longueurs identiques ou non.

La figure 2.1 illustre ces différents termes.

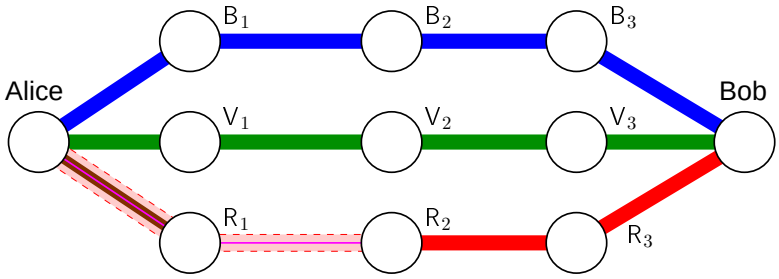


FIG. 2.1 – Cette figure représente une *communication* entre Alice et Bob, ici constituée de trois *routes* (en bleu, vert et rouge). La vue éclatée du début de la route rouge laisse apparaître ses deux premières *connexions*, entre Alice et le nœud R_1 et Alice et le nœud R_2. Voir aussi la figure 1.6, où les différentes connexions qui composent une route sont visibles.

2.2 Le problème à résoudre

Notre but est d'obtenir un système permettant à Alice de communiquer avec Bob sans que ni Eve ni Mallory puissent s'en rendre compte. Ce système doit être suffisamment générique pour permettre des communications avec la plupart des protocoles actuels.

Le système doit de plus être utilisable par Alice pour contacter Bob sans que Bob soit obligé de l'utiliser. Dans ce cas, on accepte qu'il offre un peu moins de garanties, la partie entre la sortie du système et Bob étant alors forcément visible par un observateur.

Nous avions dès le départ quelques idées sur la manière de parvenir à cela :

– Onion-Routing, pour que la sécurité soit distribuée et que la compromission d'un point de passage de la communication ne trahisse pas l'intégralité de la communication ;
– protection de l'Onion-Routing contre l'analyse de trafic, pour éviter qu'un observateur puissant (voire global) puisse découvrir une route par analyse de trafic. Nous revenons sur ce point dans la section suivante ;
– utilisation d'une architecture pair-à-pair (P2P) : d'une part pour qu'un utilisateur n'ait pas un rôle particulier par rapport à un autre type d'entité. Ainsi, des communications passent par la machine de l'utilisateur même s'il ne communique pas lui-même. D'autre part, une architecture P2P permet d'éviter d'avoir *un* point faible sur le réseau dont la chute provoque la paralysie de l'ensemble du système (que cette chute soit causée par une attaque informatique de Mallory ou par une contrainte légale par exemple) ;
– passage à l'échelle : l'anonymat dépend en grande partie de la taille du réseau ; pour avoir un grand réseau, il faut que le système passe à l'échelle.

Nous voulions de plus résoudre autant que possible les problèmes de performances de l'Onion-Routing, ou du moins avoir des performances comparables du point de vue de l'utilisateur du système.

Comme c'est souvent le cas, la solution d'un problème pose elle-même d'autres problèmes, qu'il faut alors résoudre. La figure 2.2 donne un aperçu des relations problèmes-solutions que nous avons obtenues en cherchant à résoudre les problèmes d'analyse de trafic posés par l'Onion-Routing.

En revanche, une de nos hypothèses de travail était celle de la « gratuité » de la bande passante : son gaspillage apparent n'est pas un problème si cela permet une meilleure protection.

2.3 La protection contre l'analyse de trafic

Comme on le voit sur la figure 2.2, il y a principalement trois problèmes à résoudre pour protéger l'Onion-Routing contre l'analyse de trafic :

1. le fait qu'Eve (au niveau d'un nœud) peut lier les connexions entrantes et sortantes par leurs temps d'établissement et de fin ;
2. le fait qu'Eve peut lier les connexions entrantes et sortantes par le nombre et la taille des paquets qui transitent ;
3. le fait que Mallory peut rejouer des paquets sur une connexion entrante pour observer sur quelle connexion sortante des paquets sont alors dupliqués.

Cette liste n'est pas exhaustive (ainsi, par exemple, nous n'y indiquons pas qu'un nœud de la route peut essayer de connaître son emplacement dans celle-ci en envoyant un message de contrôle quelconque à Alice et en mesurant le temps qu'elle met à répondre. Ce temps dépendra de la longueur de la route entre Alice et le nœud si aucune précaution n'est prise), mais il s'agit des principaux problèmes à résoudre. Les solutions que nous proposons à ces problèmes ne sont pas, pour la majorité d'entre elles, particulièrement exotiques ; néanmoins, la plupart des auteurs qui déclarent dans la littérature que pour résoudre l'un de ces problèmes « il suffit de ... » n'ont probablement jamais essayé les solutions qu'ils préconisent. En effet, on s'aperçoit bien vite que ces solutions ne sont pas directement utilisables, car elles posent à leur tour des problèmes. Dans les chapitres qui suivent, nous revenons donc sur ces problèmes, essayons de les résoudre, puis de résoudre les problèmes qui se posent alors, et itérons ainsi jusqu'à obtenir une solution satisfaisante, où tous les problèmes majeurs sont résolus.

2.4 La théorie et la pratique

Nous nous sommes efforcé de séparer nos propositions de *True Nyms* même. Les aspects propres au protocole et / ou à son implémentation sont donc pour la plupart regroupés dans les chapitres 6 et 7.

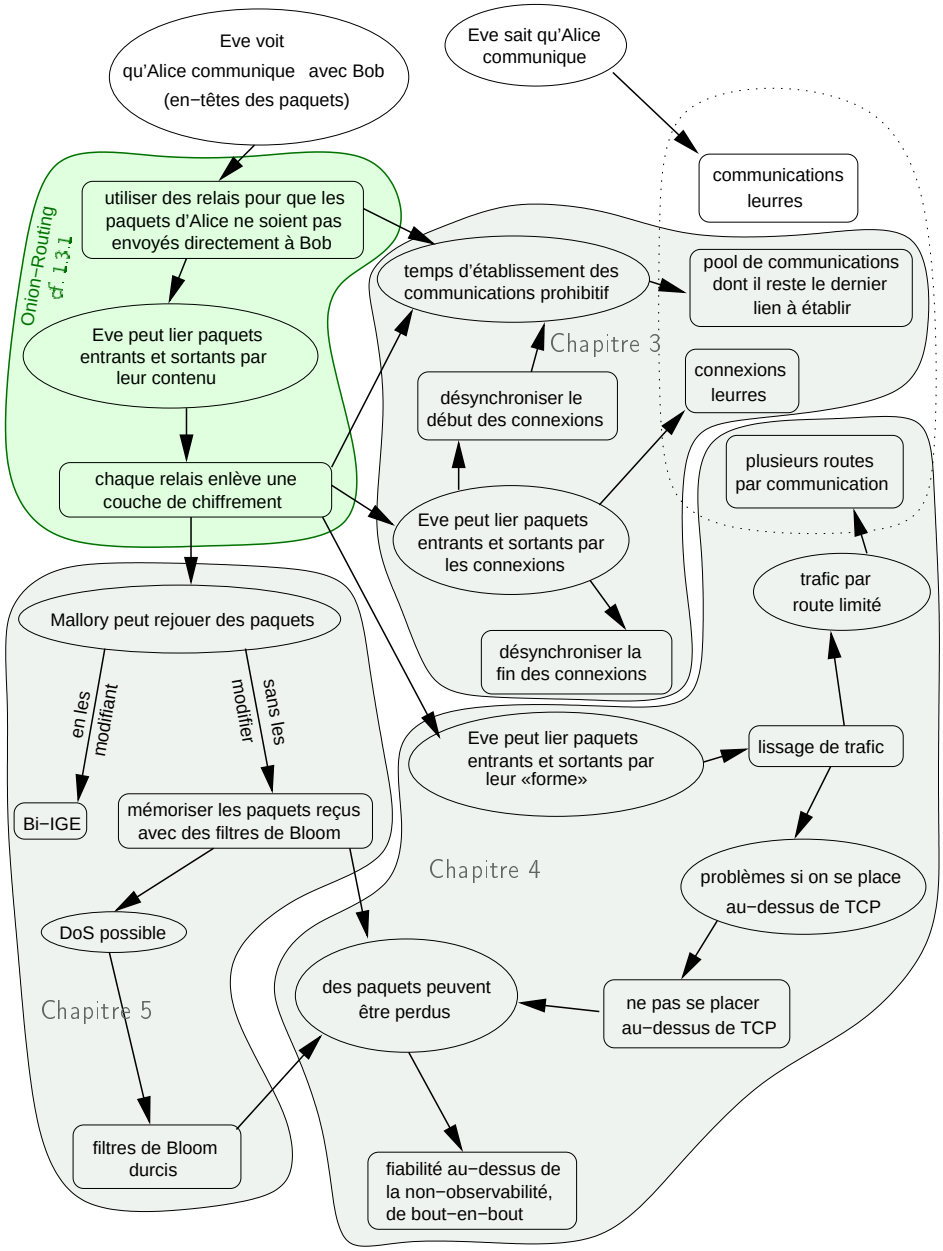


FIG. 2.2 – Relations entre problèmes et solutions (version simplifiée, à partir des problèmes d'analyse de trafic que pose l'Onion-Routing). Les ellipses représentent les problèmes, les boîtes les solutions. L'ensemble vert représente l'Onion-Routing tel qu'il existait déjà, et les ensembles gris les principales parties de notre proposition. L'ensemble en pointillé dénote des idées conceptuellement différentes, mais qui peuvent être regroupées en une mise en œuvre commune.

3

Les problèmes de l'établissement des communications

L'établissement d'une communication pose divers problèmes, tant du point de vue de la sécurité que pour l'« utilisabilité » du système. En effet, si l'on suit une approche naïve et que l'on n'établit les divers tunnels constituant une route en oignon qu'au moment où l'utilisateur en a besoin, on est alors confronté aux faits suivants :

- d'une part la connexion est très lente : il faut se connecter successivement aux différents nœuds de la route, en négociant à chaque fois une clef de session. Cela nécessite donc plusieurs aller-retours sur le réseau, ainsi que la mise en oeuvre de cryptographie à clef publique, cette dernière étant connue pour sa lenteur ;
- d'autre part, si au niveau d'un nœud de la route une nouvelle connexion entrante est créée puis que, peu de temps après, une connexion sortante est établie, une entité qui observe le nœud peut raisonnablement conclure qu'il y a une relation entre les deux.

Il y a de plus des risques à l'échelle des connexions ; nos hypothèses de sécurité sont plus strictes et / ou inhabituelles par rapport à celles sous-jacentes à la plupart des protocoles d'établissement de clef de session existants, et ceux-ci ne conviennent pas. Dans ce chapitre, nous allons donc commencer par traiter de ce dernier problème, avant de revenir sur les deux précédents.

3.1 L'établissement de connexion : Diffie-Hellman

Le protocole de Diffie-Hellman (DH) est bien connu. Basé sur le problème du logarithme discret, il fut le premier protocole à clef publique *publié* ; ses principales étapes sont rappelées dans l'encadré qui suit.

Le protocole de Diffie-Hellman (1976)

Ce protocole permet à deux parties (en général Alice et Bob ; Alice et un nœud de la route dans notre cas) de se mettre d'accord sur un « nombre secret » commun. Ce nombre pourra par exemple servir de clef de chiffrement symétrique. Les échanges se font de manière publique, mais Eve ne peut pas déterminer pour autant le nombre final. C'est le premier protocole de cryptographie à clef publique publié [31].

Soient p un nombre premier et $\alpha \in \mathbf{F}_p^*$ un générateur de $\mathbf{F}_p^*$.

Étape 1 :
le client choisit un secret $x, 1 \leq x \leq p-2$ et envoie au serveur $\alpha^x \bmod p$;
Étape 2 :
le serveur choisit un secret $y, 1 \leq y \leq p-2$ et envoie au client $\alpha^y \bmod p$;
Étape 3 :
les deux parties calculent la clef $K = (\alpha^y)^x \bmod p = (\alpha^x)^y \bmod p$.

Il n'y a donc que deux messages envoyés sur le réseau (étapes 1 et 2).

Il est de notoriété publique que ce protocole est vulnérable à des attaques de type *Man-in-the-Middle* (MitM). Il existe des parades à ce défaut, mais qui, comme nous allons le voir (3.1.1), ne conviennent pas forcément à notre cas. Il y a aussi des problèmes techniques qui, s'ils sont aisés à résoudre, doivent l'être d'une manière inhabituelle du fait de notre contexte particulier (3.1.2).

3.1.1 Diffie-Hellman et le *Man-in-the-Middle*

L'un des problèmes du protocole de Diffie-Hellman est le risque de *Man-in-the-Middle*. Si dans de nombreux cas cette attaque, bien que théoriquement possible, est difficile à mettre en oeuvre, ce n'est pas vrai ici. Au contraire, si rien n'est fait pour s'en prémunir, un nœud N_i par lequel passe la route serait dans une position idéale pour déployer ce genre d'attaque entre Alice et les nœuds en aval (*i.e.,* les nœuds N_j, avec $j > i$), y compris Bob. Si ce nœud ne peut pas forcément pour autant identifier Alice, il aurait par contre accès au contenu de la communication.

Il nous faut donc authentifier, vis-à-vis d'Alice, chaque nœud de la route lors de l'établissement de la clef de session. Même si l'on suppose (comme dans notre cas) que chaque nœud dispose d'une paire de clefs RSA et qu'Alice dispose des clefs publiques, il faut encore éviter les attaques par rejeu. S'il existe divers protocoles pour cela (STS par exemple), ils ont l'inconvénient de faire de l'authentification mutuelle, ce qui ne nous convient pas : nous ne voulons pas que le nœud auquel on se connecte connaisse l'identité d'Alice ! Nous n'avons pas trouvé trace dans la littérature d'un protocole qui réponde à la fois à ces contraintes et à celles du paragraphe 3.1.2 : c'est l'objet du paragraphe 3.1.3.

3.1.2 Les fuites d'information du protocole de Diffie et Hellman

Lorsque le protocole de Diffie-Hellman est décrit d'une « manière cryptographique », comme dans l'encadré ci-dessus, un certain nombre de paramètres sont « partagés » par les utilisateurs, d'une manière non précisée.

Dans la pratique, bien que ces paramètres puissent être fixés (par exemple dans la définition d'un protocole), ce n'est généralement pas le cas, pour des raisons d'évolutivité du système (en effet, pour changer les paramètres il faudrait alors faire toutes les mises à jour simultanément). Les paramètres sont donc en général fixés par les parties avant le début du protocole lui-même.

La solution la plus élégante consiste à ce que ce soit le client qui envoie les paramètres au serveur, car il peut alors envoyer en même temps le premier message du protocole, à savoir $\alpha^x \bmod p$. Dans notre cas néanmoins, puisque c'est toujours Alice qui joue le rôle du client, si l'on suppose que chaque nœud (chaque utilisateur) utilise des paramètres différents, cela pose le problème suivant : l'espace dans lequel ces paramètres peuvent être choisis est très vaste, et par conséquent, si chaque nœud choisit ses paramètres aléatoirement, ceux-ci peuvent alors être utilisés comme un moyen d'identifier ce nœud. Cela impliquerait donc que :

- chaque nœud de la route (y compris Bob) dispose d'un moyen d'identifier le nœud origine (Alice), et cela rendrait donc l'anonymat inexistant ;
- comme la connexion n'est pas chiffrée entre le nœud N_i et le nœud N_{i+1}, un observateur peut voir les paramètres de Diffie-Hellman d'Alice être émis sur cette connexion et donc connaître également l'origine de la route. En observant le nœud N_{i+1}, il peut à nouveau observer ces mêmes paramètres être émis à destination du nœud N_{n+2}, et ainsi de suite suivre la route jusqu'à son terme. Il n'y aurait alors pas davantage de non-observabilité.

Pour ces raisons, cette solution n'est bien sûr pas viable dans notre contexte. Une possibilité serait pour le nœud qui crée la route de générer aléatoirement des paramètres à chaque utilisation du protocole de Diffie-Hellman. Le coût de cette opération rend cette idée impraticable sur les machines actuelles, mais elle pourrait être intéressante d'ici quelques années[1]. Une variante un peu plus rapide consiste à générer des paramètres DSA et à les convertir en paramètres DH.

Même cette dernière variante requiert beaucoup de calculs de la part d'Alice, aussi nous sommes-nous tourné vers une solution où c'est le nœud jouant le rôle de serveur qui choisit les paramètres du protocole de Diffie-Hellman et les transmet au client. L'inconvénient est qu'il faut alors un message supplémentaire du client vers le serveur, comme on le voit dans le paragraphe suivant (3.1.3).

3.1.3 « Notre » protocole de Diffie-Hellman

Nous avons finalement abouti au protocole suivant (les notations sont les mêmes que dans l'encadré sur Diffie-Hellman, H est une fonction de hachage cryptographique, $|$ dénote la concaténation et $\{\ldots\}_{K_s^{\mathrm{N}}}$ un chiffrement avec la clef secrète (K_s) (c'est-à-dire une signature) du nœud N) :

[1] Il faudrait toutefois savoir quel est le facteur limitant : est-ce la génération de nombres aléatoires ? Dans ce cas un générateur *hardware* pourrait peut-être aider. Si c'est « l'élimination » des mauvais candidats, ce n'est qu'une question de puissance du processeur.

$$\begin{array}{l}
\text{Client} \longrightarrow \text{Serveur} : \textit{nonce} \\
\text{Client} \longleftarrow \text{Serveur} : \underbrace{p \mid \alpha}_{\substack{\text{paramètres} \\ \text{de DH}}} \; |\alpha^y \bmod p| \underbrace{\{H\,(p|\alpha|\alpha^y \bmod p|\textit{nonce})\}_{K_s^{\text{serveur}}}}_{\text{signature avec le } \textit{nonce}} \\
\text{Client} \longrightarrow \text{Serveur} : \alpha^x \bmod p
\end{array}$$

La présence du *nonce*[2] empèche le rejeu du message envoyé par le serveur par un attaquant. Notez qu'un attaquant pourrait toujours détourner la connexion avec le serveur à son profit en envoyant un $\alpha^{x'} \bmod p$ au serveur à la dernière étape, mais cela ne lui serait d'aucune utilité, puisque c'est a priori Alice qu'il veut tromper.

NB : on suppose qu'Alice connaît les clefs publiques des nœuds (ou de certains d'entre eux : cette connaissance n'est nécessaire que pour éviter une attaque de type MitM)). Dans *True Nyms*, elles sont obtenues par une autre partie du système pair-à-pair.

3.2 L'établissement de communications

L'un des problèmes de l'Onion-Routing et de ses dérivés est que si l'on établit les tunnels au moment où l'on désire créer une communication, alors :

- d'une part l'établissement de la communication est lent (il faut créer tous les tunnels, avec négociation d'une clef via des mécanismes de cryptographie à clef publique pour chacune) ;
- d'autre part, cela implique que les tunnels sont créés pour une même communication les uns après les autres sans pause (sauf à rendre l'établissement de communication extrêmement lent).

3.2.1 Un *pool de routes* pré-établies

Pour contourner le problème de la vitesse d'établissement des communications, l'idée que nous avons retenue consiste à créer des routes aléatoires à l'avance, dans un *pool de routes* ; ainsi, lorsqu'Alice souhaite établir une communication, il n'y a plus qu'à prendre une de ces routes et à établir le lien entre le dernier nœud et Bob.

Cela permet donc d'établir une communication beaucoup plus rapidement (cf. 7.2.2) ; de plus, il est possible lors de la création de la route d'utiliser des temps d'attente aléatoire, ce qui permet de rendre l'observation plus difficile pour l'utilisateur (cf. 3.2.2). L'inconvénient est qu'il n'est plus possible pour l'utilisateur de spécifier à chaque fois la taille de la route.

L'observateur ne peut déterminer si une route est utilisée ou non à un instant donné. Ces routes servent donc aussi de communications leurres.

Il faut bien sûr déterminer combien de routes il est nécessaire de garder dans le pool. Si le nombre choisi est trop grand, on risque de saturer la bande passante et / ou la machine (coût des opérations cryptographiques) inutilement ; s'il est trop faible, l'utilisateur risque de ne pas avoir de routes disponibles au moment où il en a besoin.

[2] En cryptographie, le terme *nonce*, pour *number used once*, désigne un nombre [aléatoire] utilisé une seule fois.

Ce paramètre n'est pas aisé à déterminer dans l'absolu, car il dépend également du type d'utilisation qui est fait des routes : une connexion `ssh` utilise une seule route et a une durée de vie longue. Inversement, la navigation sur le Web utilise beaucoup de communications très brêves et risque d'« user » les routes plus vite qu'il n'est possible de les créer. Dans ce dernier cas, l'utilisation d'un client (ou d'un proxy entre le vrai client et notre système) « intelligent », par exemple qui utilise une même route en ne changeant que la fin de celle-ci, à la manière du *keepalive* de `http`, est à envisager pour réellement bénéficier du pool de routes.

3.2.2 Mécanismes anti-observation

L'un des problèmes avec l'Onion-Routing face à un observateur global, c'est que celui-ci, s'il peut observer tous les nœuds d'une route, même s'il ne les contrôle pas, peut très bien suivre une communication lors de son établissement. En effet, s'il voit une connexion arriver sur un nœud, puis un instant plus tard, une nouvelle connexion en sortir, il se doutera qu'il existe un lien entre celles-ci. S'il peut itérer le processus, il peut ainsi suivre l'établissement de la communication d'Alice à Bob.

Nous décrivons dans cette section deux contre-mesures destinées à rendre la tâche d'Eve beaucoup plus complexe.

Un arbre de connexions leurres

Une première contre-mesure est la suivante : lors de la construction d'une route, quand on demande à un nœud d'établir une connexion vers un autre nœud, au lieu d'établir simplement celle-ci, il crée simultanément d'autres connexions vers d'autres nœuds.

L'observateur verra ainsi *une* connexion entrante, mais c connexions sortantes et ne saura pas laquelle correspond à la première. Il est bien sûr possible, et même recommandé, que le nœud combine ces créations de connexions avec la création de routes pour son propre *pool* de routes pré-établies (cf. section précédente).

Bien sûr, le nœud qui crée les connexions leurres sait quelle est la « bonne » connexion sortante, mais si l'on a plusieurs nœuds sur une route, c'est entre autres pour diminuer la probabilité que tous les nœuds d'une route soient compromis.

Le principal problème de cette contre-mesure est qu'il ne faut pas que ces fausses connexions saturent le réseau. En effet, si pour chaque connexion sortante, un nœud crée c *routes* leurres dont chaque connexion provoquera la création d'autres routes par les nœuds par lesquelles elles passent, il y a une rétroaction positive dans le réseau : chaque route va en créer à son tour c à l'étape suivante (on est déjà à c^2), qui vont provoquer la création de c autres routes chacune (c^3), etc. Cela va provoquer un effondrement du réseau à cause d'une quantité exponentielle de route créées (c^n après n étapes, puisqu'il s'agit d'un arbre c-aire de profondeur n) ! La figure 3.2 illustre cela, avec pourtant c et n petits (tous deux égaux à trois).

Une solution est de ne pas créer *toujours* c connexions sortantes leurres pour chaque connexion, mais un nombre qui varie dynamiquement selon la charge du

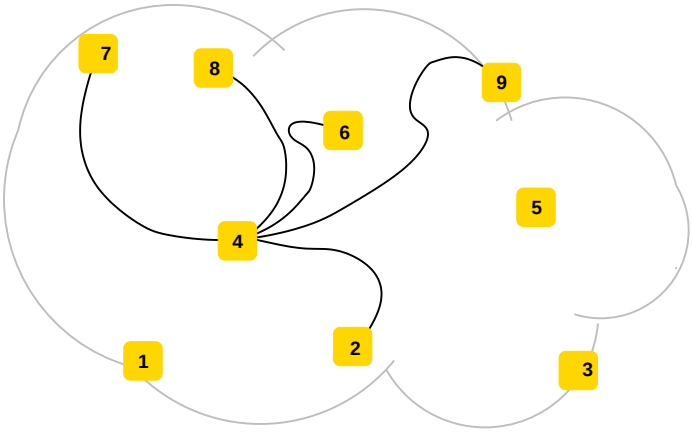


FIG. 3.1 – Lorsqu'on lui demande d'établir une connexion avec le nœud N_{i+1}, le nœud N_i crée c autres connexions, vers d'autres nœuds. L'attaquant ne sait donc pas laquelle de ces connexions est la bonne et doit tenter d'observer tous les nœuds destinataires d'une de ces connexions.

réseau perçue par le nœud. Le problème est alors de s'assurer que Mallory ne peut pas faire croire à chaque nœud que le réseau est très chargé pour éviter la création de ces connexions.

En supposant que Mallory ne puisse « charger » tous les nœuds simultanément, une solution pourrait être pour chaque nœud de considérer non pas la charge à l'instant où il doit créer la connexion sortante, mais celle à l'instant où a été établie la connexion entrante. En effet, Mallory ne peut deviner vers quel nœud sera établie la prochaine connexion ; quand celle-ci est établie, il n'a pas le temps de « charger » le nœud destinataire.

Cette contre-mesure est efficace contre un observateur puissant mais non global. Par exemple, si l'on suppose que les nœuds de l'arbre sont répartis dans de nombreux pays, un gouvernement qui désire observer Alice devrait obtenir l'aval de ceux des autres pays. Alors que cette attaque légale (en anglais *subpoena attack*) serait possible avec une route normale (linéaire), dont la taille (en nombre de nœuds) est de l'ordre de n, elle ne serait, bien que théoriquement possible, guère envisageable avec un arbre (c^n autorisations à obtenir !).

Cette attaque reste néanmoins à la portée d'un adversaire réellement global, puisque celui-ci peut observer l'intégralité des nœuds simultanément, donc l'arbre dans sa totalité, et en particulier voir les données en clair sortir de l'arbre[3]. La méthode décrite dans le paragraphe suivant est alors utile.

[3] Si Bob fait partie du réseau, les données en clair ne circulent pas sur l'Internet : l'observateur, même global, ne sait pas qui est Bob (k-anonymat, où k est le nombre de nœuds dans l'arbre). En fait, l'attaquant ne sait même pas si Bob fait partie du réseau ou s'il s'agit d'une communication leurre ! Si des routes multiples (cf. 4.4.1) sont utilisées, il peut tenter d'établir l'arbre pour chaque route. Les nœuds qui figurent dans toutes les routes sont suspects ; le problème pour l'attaquant est qu'Alice peut établir plusieurs communications simultanément, sans même parler des routes leurres : toutes les routes ne font donc pas partie de la communication avec Bob !

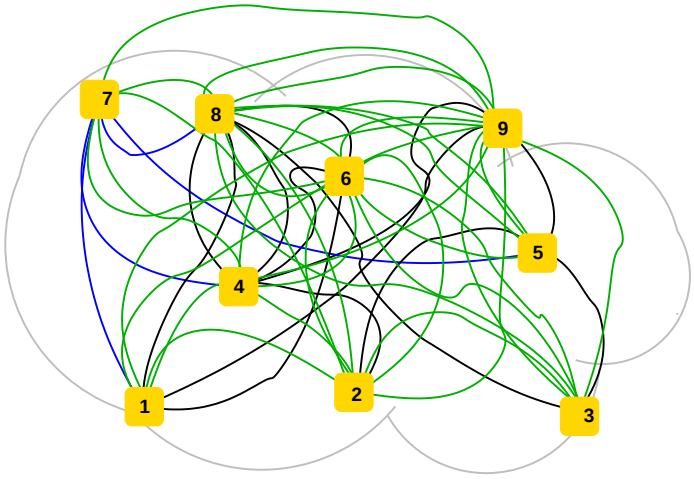


FIG. 3.2 – La création naïve de c connexions leurres par connexion entrante produit une réaction en chaîne qui conduit à la surcharge rapide du réseau (ici $c = 3$, au bout de trois étapes). Seul le nœud 7 crée des routes au départ.

Délais d'attente

Si l'on établit une route au moment où l'utilisateur en a besoin, on ne peut pas se permettre d'attendre au niveau de chaque nœud. Dans notre cas cependant, les communications sont, sauf pour leur dernière partie, pré-établies (cf. section 3.2). Nous pouvons donc nous permettre d'attendre à chaque nœud avant d'établir la connexion suivante sans pénaliser le temps d'établissement ressenti.

Si l'on attend ainsi un certain temps (une valeur aléatoire comprise entre certaines bornes), avant d'établir notre connexion sortante, il y a alors des chances qu'une autre connexion entrante arrive. L'attaquant ne pourra alors plus décider si la connexion sortante correspond à l'une ou l'autre des connexions entrantes.

On peut encore compliquer sa tâche en combinant cela avec la méthode exposée au paragraphe précédent, mais en établissant nos connexions leurres non pas simultanément, mais avec pour chacune un temps d'attente aléatoire indépendant.

Si l'on suppose que deux connexions entrantes arrivent à peu près en même temps sur un nœud N, la différence est la suivante :

- dans le premier cas, Eve voit à l'instant t_1 (respectivement t_2) un groupe de c_1 (resp. c_2) connexions s'établir. Elle sait qu'une seule connexion parmi chaque groupe est bonne, ce qui lui fait $\mathrm{C}^1_{c_1} + \mathrm{C}^1_{c_2} = c_1 + c_2$ couples {connexion entrante, connexion sortante} à envisager[4) ;
- dans le second cas, Eve a vu deux connexions arriver (aux instants t_1 et t_2) puis voit $c_1 + c_2$ connexions s'établir, parmi lesquelles elle sait que seules deux sont bonnes. Elle a cette fois $\mathrm{C}^2_{c_1+c_2}$ couples {connexion entrante, connexion sortante} à envisager.

Si par exemple, sur le nœud N, pour chaque connexion, on établit trois connexions leurres (*i.e.*, $c_1 = c_2 = 3 + 1 = 4$), on passe de $2 \times C^1_4 = 8$ à $C^2_8 = 28$ possibilités. Cela diminue donc la probabilité de pouvoir lier une connexion

[4] Avec $\mathrm{C}^k_n = \frac{n!}{k!(n-k)!}$. Notons le cas particulier suivant : $\mathrm{C}^1_n = n$.

sortante (leurre ou non) avec une connexion entrante.

De plus, même si l'attaquant réussit à suivre une communication jusqu'à son terme et observe alors que des données en clair sont réellement transmises, il n'est pas certain qu'elles proviennent de la personne qu'il observe, puisqu'elles peuvent provenir d'une des autres connexions qu'il n'a pu distinguer sur un nœud.

Comme d'habitude, *le Diable est dans les détails* : en effet, la vraie connexion avec le nœud N_{i+1} est établie par Alice, alors que les connexions leurres le sont par le nœud N_i. Or, avant la négociation d'une clef de session, les données entre le nœud N_i et les nœuds « suivants » (c'est-à-dire les nœuds des connexions leurres et le nœud N_{i+1}) ne sont pas chiffrées (le lissage de trafic est donc inutile) et Eve peut les observer. Si elle observe un motif semblable à celui de la figure 3.3, elle peut aisément déterminer quelle est la vraie connexion.
Pour éviter cela, il y a plusieurs possibilités envisageables :

- chiffrer les communications de nœud en nœud *en plus* de l'Onion-Routing : le lissage de trafic (voir chapitre suivant) serait alors utilisable pour masquer à Eve les délais introduits par la latence. En revanche, si Mallory contrôle le nœud N_{i+1} il pourra toujours savoir que la connexion est « réelle » ;
- introduire des délais et des attentes : si le nœud N_i attend que la réponse d'Alice à destination du nœud N_{i+1} arrive avant de continuer, Eve ne peut plus distinguer les communications leurres des vraies. C'est cette solution que nous avons retenue dans *True Nyms*.

3.3 La fin des connexions et des communications

Avant de terminer ce chapitre, nous voudrions en profiter pour parler sommairement du problème inverse : en effet, dans l'Onion-Routing naïf, Eve peut observer l'établissement successif des connexions qui forment une route ; elle peut aussi observer leurs fins et en déduire a posteriori les relations qu'elle cherche à établir si, naïvement, l'on clôt toutes les connexions d'une route simultanément.

La fin d'une route peut être due à deux causes :

- la communication dont elle fait partie est terminée ;
- un problème technique est survenu sur cette route (par exemple un des nœuds disparaît).

Dans le premier cas, à l'instant où la route n'est plus utile, seuls Alice et/ou Bob le savent. Dans le second, si c'est le nœud N_i qui coupe la route (en ne fonctionnant plus), cette interruption sera détectée par les nœuds N_{i-1} et N_{i+1}. Le nœud N_{i-1} communique avec Alice et l'informe de la situation ; on se ramène donc au premier cas, le nœud N_{i-1} remplaçant Bob. Le nœud N_{i+1}, par contre, ne peut communiquer avec Bob, car il ne connaît pas les clefs des connexions qui passent par lui. Bob peut néanmoins découvrir la situation de deux manières :

- si la route fait partie d'une communication qui comporte d'autres routes qui fonctionnent encore, Alice peut l'informer par l'une de ces autres routes ;
- il ne recevra plus de messages d'Alice sur cette route.

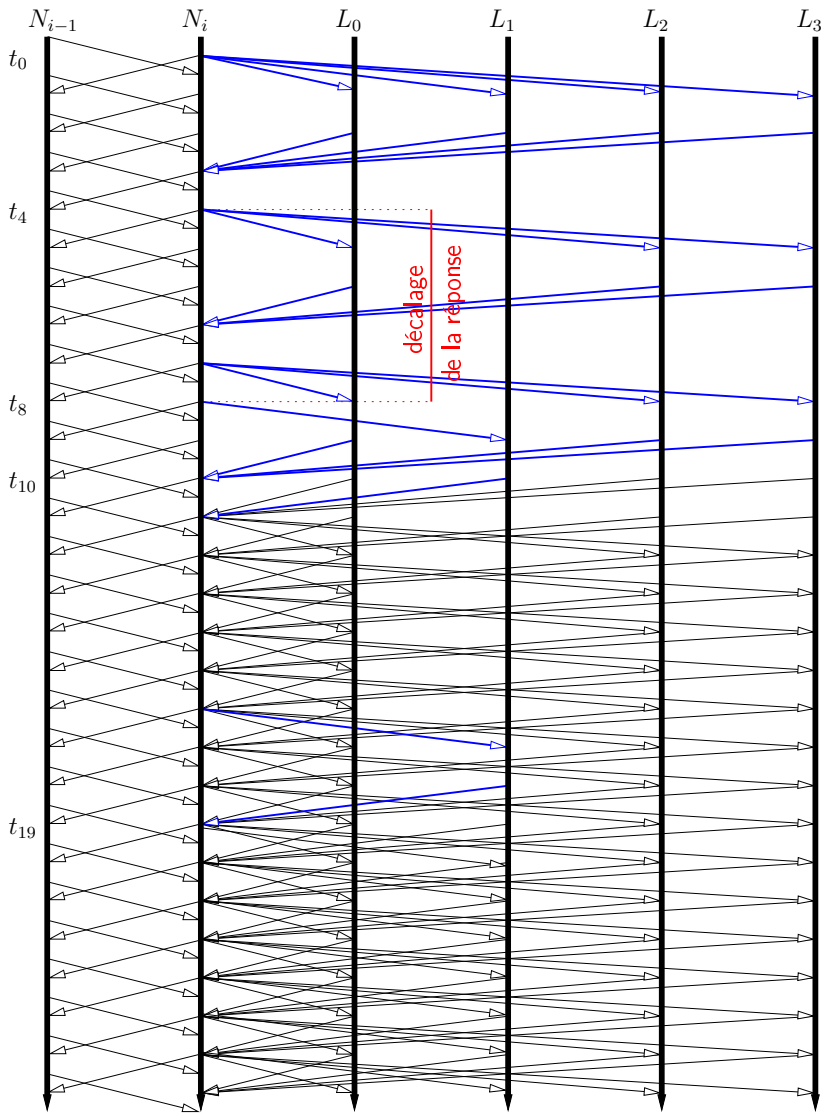


FIG. 3.3 – À t_0, des demandes d'établissement de connexion sortent du nœud N_i vers les nœuds L_x. Les paquets non chiffrés (établissement de connexion) sont représentés en bleu, les paquets chiffrés en noir. Eve sait que seule une de ces connexions correspond à la connexion entrante depuis le nœud N_{i-1}. Ici, elle peut observer que les « réponses du nœud N_i » à destination du nœud L_1 sont beaucoup plus lentes que celles à destination des autres nœuds L_x : par exemple, le second message de l'établissement d'une connexion est envoyé vers les nœuds L_0, L_2, L_3 à t_4 et à t_8 vers le nœud L_1, et les connexions sont réellement établies (début du chiffrement et du lissage de trafic) respectivement à t_{10} et t_{19}. Elle en déduit que L_1 est le nœud N_{i+1}, cette lenteur apparente étant due au fait que c'est en réalité Alice qui répond et à la latence de la communication entre le nœud N_i et celle-ci. Note : pour simplifier, les différentes connexions L_i ont été représentées comme établies de manière simultanée.

Dans tous les cas, il y a des nœuds qui savent que la route n'est plus utilisée. Ces nœuds peuvent décider d'interrompre la connexion qui les lie avec un autre nœud de la route, par exemple au bout d'un temps aléatoire ou encore parce qu'ils ont besoin de bande passante pour établir une nouvelle route.

Le cas d'Alice est un peu particulier car elle dispose encore de connexions avec les nœuds de la route (au moins jusqu'au nœud N_{i-1}) ; elle peut donc demander à un des nœuds de cette route d'interrompre celle-ci à son niveau.

Dans tous les cas, il y a encore un autre facteur qui vient s'ajouter : en effet, à cause des mécanismes de prévention des attaques par rejeu que nous décrirons au chapitre 5, chaque connexion a une durée de vie limitée (même si elle peut être prolongée en changeant de clef de session). Ainsi même si Alice et Bob ne font rien pour détruire explicitement une route, celle-ci sera progressivement détruite, par « pourrissement ».

Ajoutons pour finir sur ce point qu'il serait judicieux que le protocole fasse en sorte que la raison pour laquelle une connexion est interrompue ne soit pas discernable par Eve.

4

Le lissage de trafic et ses implications

L'idée de fournir un « lissage du trafic » (*traffic shaping*) est simple : il suffit de définir le protocole de manière à ce qu'un nœud envoie sur chaque connexion une quantité constante de données par unité de temps, c'est-à-dire la taille des paquets et la périodicité des envois.

4.1 Pourquoi lisser le trafic ?

L'Onion-Routing seul ne suffit pas à empêcher un observateur de faire la correspondance entre les connexions entrantes et sortantes d'un routeur. En effet, si le *contenu* lui-même change et ne peut être utilisé pour établir cette correspondance, une analyse du trafic particulièrement simple, temporelle et / ou basée sur la taille des paquets permet de retrouver celle-ci (cf. figure 4.1).

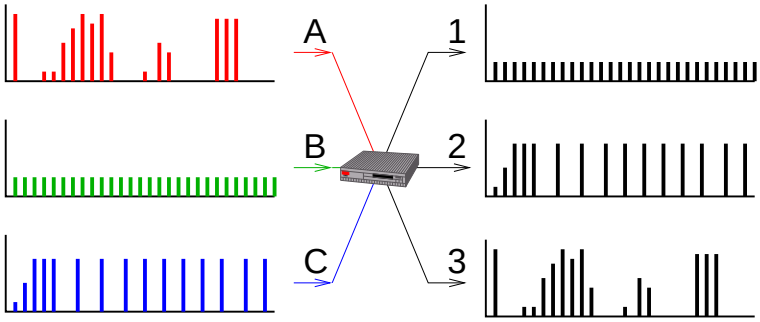


FIG. 4.1 – Sans lissage de trafic, un observateur peut facilement établir la correspondance entre les connexions entrantes (à gauche) et sortantes (à droite) d'un routeur, ici A-3, B-1, C-2.

Le lissage de trafic consiste donc à ce que tous les nœuds envoient des paquets d'une taille fixée commune, avec une même période. Si les données à envoyer

ne tiennent pas dans un paquet, on en retarde une partie qui partira avec le suivant. Inversement, s'il n'y a pas assez de données pour remplir un paquet, on complète (« *padding* ») celui-ci, de manière à ce qu'il ait la même taille que les autres, avant de le chiffrer. Si à un instant précis il n'y a aucune donnée à envoyer, le paquet entier sera constitué de padding ; on parlera alors de paquet leurre. Le trafic de chaque connexion semble alors similaire (cf. figure 4.2) et l'observateur ne peut plus se baser sur une analyse de ce type.

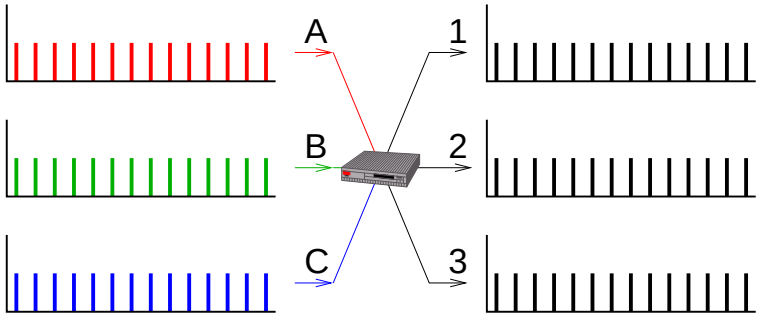


FIG. 4.2 – Avec lissage de trafic, la correspondance entre les connexions entrantes (à gauche) et sortantes (à droite) d'un routeur est plus difficile à déterminer.

Il n'est pas suffisant de lisser le trafic au niveau des émetteurs. En effet, les aléas du réseau font que deux paquets qui se suivent n'auront pas exactement le même temps de transit. S'ils sont tous deux transmis immédiatement par le routeur (même si le temps de transmission peut, lui aussi, varier légèrement), on risque fort de retrouver une corrélation significative entre les temps d'entrée et les temps de sortie. Pire, un paquet peut se perdre. Si le routeur n'introduit pas de paquet pour compenser, on se retrouve dans le cas de la figure 4.3, et l'observateur peut établir la correspondance entre une connexion entrante et une connexion sortante.

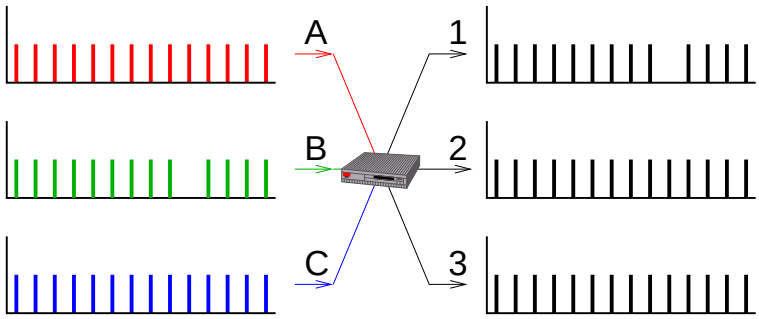


FIG. 4.3 – Si les routeurs ne veillent pas à rétablir le lissage en cas de problèmes, l'observateur obtient des informations. Ici, par exemple, la correspondance entre la connexion entrante B et la connexion sortante 1.

Il faut donc qu'un routeur puisse insérer des paquets leurres pour remplacer d'éventuels paquets perdus. Ces paquets leurres doivent être indistinguables des paquets réels pour les routeurs suivants. Comme le routeur qui les émet ne doit

bien sûr pas connaître les clefs de chiffrement établies entre Alice et les routeurs en aval, il ne peut insérer d'en-tête spécifique.

Cela implique donc qu'un paquet ne doit pas contenir d'en-tête d'Alice destiné à chaque routeur.

Ce n'est pas un problème dans la pratique. Une fois la communication établie, chaque routeur peut utiliser les informations fournies par les niveaux inférieurs de la pile IP pour connaître la connexion à laquelle appartient un paquet.

4.2 Problème : taille et période d'émission

Une des questions problématiques est, par contre, celle du choix des paramètres. En effet, le lissage consiste donc, nous l'avons dit, à choisir une taille de paquet et une période d'émission. Cependant, les besoins des différentes applications ne sont pas les mêmes, loin de là. Voici par exemple deux cas :

1. une application de VoIP, envoie généralement de petits paquets, avec une fréquence élevée. La quantité de données à envoyer reste stable dans le temps : il s'agit du son capté par le micro, avec une fréquence d'échantillonnage fixée (il y a bien sûr en général des algorithmes de compression spécifiques qui influent sur la quantité de données à transmettre) ;
2. les navigateurs Web, de leur côté, ont des quantités de données à transmettre très irrégulières et leur trafic montre des pics de données pendant quelques instants, puis une absence de données pendant un temps assez important, puis un nouveau pic, etc. (voir par exemple les figures de la section 1.1.1).

Il faut donc concilier diverses catégories d'applications autant que possible, sachant qu'il y a toujours une partie « en-tête » de taille fixe. L'emploi de petits paquets impliquera donc un ratio

$$\frac{\text{taille des données utiles}}{\text{taille du paquet}}$$

plus petit ; c'est pourquoi les applications qui transfèrent beaucoup de données préfèrent envoyer des paquets aussi gros que possible.

Notons que ce type de dilemme s'est déjà posé : par exemple lors de la conception d'ATM, qui devait être destiné à transporter à la fois de la voix et des données. Les spécialistes de réseaux informatiques voulaient une taille de paquet[1] d'au moins 128 octets afin que le rapport précédent ne soit pas trop mauvais. À l'inverse, les spécialistes de téléphonie voulaient envoyer une grande quantité de petit paquets et ne voulaient pas de paquets de plus 32 octets.

Finalement, les paquets ATM ont une taille de 53 octets (dont 5 d'en-tête).

Notons toutefois que le cas d'ATM est assez éloigné du nôtre et qu'une telle taille n'est a priori pas appropriée :

- le nombre de paquets par seconde d'une connexion n'est pas fixé ;
- la taille de l'en-tête est très inférieure : 5 octets, contre 20 pour IPv4 (40 pour IPv6 !) auxquels s'ajoutent soit 8 octets pour UDP soit 20 avec TCP.

[1] En fait, dans le cadre d'ATM, on parle de *cellule* et non de paquet.

4.3 Problème : perte de paquet leurre

L'un des problèmes de l'utilisation du lissage de trafic, s'il est implémenté naïvement au-dessus de TCP, est qu'en cas de perte d'un paquet leurre par le réseau, le trafic réel est retardé. En effet, si chaque nœud fonctionne au niveau applicatif et que deux nœuds qui se suivent dans la chaîne communiquent à l'aide d'une connexion TCP/IP, lorsqu'un paquet se perd, le protocole TCP retarde les données suivantes pour le retransmettre (cf. figure 4.4).

Pendant ce temps, le nœud en aval qui attend des données émet de nouveaux paquets leurres pour que le flux sur sa connexion sortante ne reflète pas les problèmes sur le flux entrant. Par contre, quand le paquet manquant est finalement arrivé, généralement avec les paquets suivants, il a cette fois trop de données à transmettre. En supposant que la transmission d'une quantité supérieure de données en sortie soit exclue, il y a alors deux possibilités :

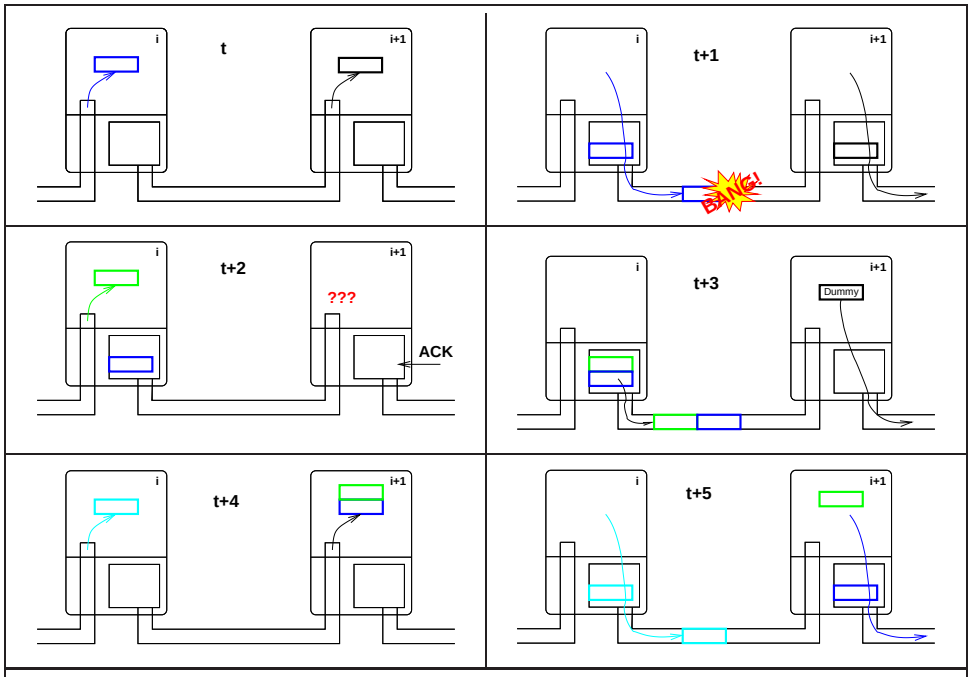
- il jette une partie des données. Comme il ne peut distinguer le trafic leurre des paquets qui contiennent réellement des données, il ne peut faire de choix. Il n'y a alors plus de garantie sur la transmission des données de bout en bout et des pertes sont possibles ;
- il place les données à émettre dans une file. S'il y a de nouveau des pertes en amont, il peut en profiter pour diminuer la taille de la file au lieu de créer de nouveaux paquets leurres, cependant cette amélioration est dans le meilleur des cas temporaire, puisque ces pertes provoqueront finalement de nouveaux afflux de données en grande quantité, et que la file retrouvera une taille au moins aussi grande. Aussi, en faisant abstraction de ce qu'il faut considérer comme des « accidents temporaires », une telle file ne peut-elle que grandir au cours du temps. Le problème d'une file longue est que cela augmente la latence de la communication de bout en bout. Par ailleurs, la modification de la taille de la file provoque une modification de la latence (gigue), ce qui est également un problème. Enfin, la mémoire d'un ordinateur étant limitée, la file ne peut grandir de manière infinie : que faire si elle dépasse une certaine taille ? Les choix possibles sont, là aussi, soit de perdre une partie des données, soit d'arrêter temporairement le lissage du trafic.

Ces propositions n'étant ni l'une ni l'autre satisfaisantes, nous préférons une approche alternative : les connexions de nœud à nœud doivent se faire en utilisant un protocole non fiable, et la fiabilité est implémentée de bout en bout, *au-dessus* du mécanisme de non-observabilité. Cela offre en particulier les avantages suivants :

- les pertes de paquets leurres ne pénalisent plus les données ;
- l'utilisation d'un mécanisme de contrôle de flux au-dessus du mécanisme de non-observabilité devient naturel (il agit sur les données et non sur le trafic transmis sur le réseau) ;
- il est moins coûteux d'encapsuler certains[2] protocoles fonctionnant normalement au-dessus d'UDP dans notre système de non-observabilité.

Cependant, la création et l'implémentation d'un tel protocole soulèvent des difficultés. Celles-ci seront abordées au chapitre 6.

[2] Les protocoles « multimédia » (VoIP, vidéo, etc.) qui transmettent de grandes quantités de données par UDP entre deux machines en bénéficient ; ce n'est pas le cas des protocoles comme DNS qui transmettent peu de données mais vers de nombreuses machines différentes.



– À l'instant t (en haut à gauche), les nœuds N_i et N_{i+1} reçoivent chacun un paquet. Ceux-ci ont été émis respectivement par les nœuds N_{i-1} et N_i.
– À l'instant suivant (en haut à droite), ils passent le paquet déchiffré à TCP pour envoi au nœud suivant. Le paquet envoyé par le nœud N_i est perdu et n'arrive donc pas au nœud N_{i+1}.
– À l'instant $t+2$, le nœud N_i reçoit un nouveau paquet, tandis que le nœud N_{i+1} s'étonne de ne pas en recevoir. À l'insu du programme, qui se trouve au niveau applicatif, la couche TCP du nœud N_{i+1} reçoit un ACK du nœud N_{i+2} qui indique que le paquet a bien été reçu. Le nœud N_i ne reçoit pas de tel paquet et garde donc le paquet en mémoire dans les tampons de TCP.
– À l'instant $t+3$, les nœuds envoient à nouveau le paquet reçu à l'instant précédent. Le nœud N_{i+1} qui n'en a pas reçu envoie un paquet leurre, indistinguable d'un paquet chiffré pour un observateur. L'implémentation de TCP du nœud N_i renvoie le paquet précédent ET le paquet que vient de lui passer l'implémentation.
– À l'instant $t+4$, sur les deux nœuds l'application lit les données. Sur le nœud N_{i+1} elle reçoit deux fois plus de données que la normale.
– À l'instant suivant $(t+5)$, les deux nœuds envoient leurs données. Le nœud N_{i+1} n'envoie que la moitié des données qu'il possède à cause du lissage de trafic : sa file de données à envoyer s'est donc allongée.

Nota bene : les détails exacts peuvent varier en fonction de l'enchaînement temporel des arrivées de paquets et des temporisations des différentes implémentations de TCP ; il existe néanmoins bel et bien une file qui s'allonge, que ce soit sur le nœud N_i ou sur le nœud N_{i+1}.

FIG. 4.4 – La difficulté du lissage de trafic au-dessus de TCP.

4.4 Problème : débit

Puisque le lissage de trafic fixe la taille et la fréquence d'émission des paquets, il fixe une borne maximale sur le débit de la communication. Il ne semble pas y avoir de solution simple à ce problème.

Si l'on fixe les paramètres du lissage de trafic à des valeurs permettant un grand débit, on augmente le gaspillage de ressources réseau par toutes les communications qui n'en ont pas besoin (c'est-à-dire la majorité). Cela limite donc la quantité de communications possibles pour chaque nœud, et rend même impossible l'utilisation de ce système par des nœuds qui n'auraient pas les ressources nécessaires pour une telle connexion. Cette solution ne nous semble donc pas viable ; même si elle l'était, elle ne serait pas pérenne, du fait de l'évolution des débits des réseaux.

Une autre possibilité serait d'avoir une gamme de paramètres. Cette solution permettrait par exemple d'avoir des paramètres pour les communications qui ont besoin de « plein de petits paquets » (VoIP, etc.) et d'autres pour celles qui ont besoin plutôt de gros paquets (transfert de fichiers, etc.). L'inconvénient de cette solution est qu'elle fournit des informations à l'observateur sur le type de trafic qui passe par le réseau (nonobstant les connexions et communications leurres). Bien que nous ne l'ayons pas retenue pour cette raison dans notre système, nous nous gardons bien de rejeter définitivement cette solution qui peut être intéressante dans certaines conditions.

Nous nous sommes donc tourné vers une autre solution, qui est présentée plus en détails dans la section suivante : l'utilisation de routes multiples vers une même destination.

4.4.1 Routes multiples

L'idée des « routes multiples » est conceptuellement simple : elle consiste à dire « puisque qu'une route est limitée à un débit d à cause du lissage de trafic, si l'on relie Alice à Bob par k routes, ils pourront échanger des données avec un débit $k \times d$ ».

La proposition a d'autre part l'avantage de fournir une meilleure robustesse : le réseau pair-à-pair sous-jacent n'est pas constitué de machines a priori fiables. Si l'on utilise une seule route, la disparition d'une machine sur celle-ci provoque la rupture de la communication. En utilisant plusieurs routes différentes, on évite ce problème, la communication pouvant se poursuivre sur la ou les route(s) restante(s) pendant qu'une nouvelle route est établie pour remplacer celle qui a été perdue.

La latence pourrait également, dans une moindre mesure, profiter de routes multiples : si l'on envoie des paquets dont le contenu est identique (mais pas le chiffrement bien entendu), le récepteur peut utiliser ceux qui arrivent le plus rapidement, tout en jetant les copies qui arrivent ensuite par les autres routes.

Il est légitime de se demander si l'usage de plusieurs routes simultanément ne nuit pas à la non-observabilité, en permettant à Eve ou Mallory de faire des corrélations qu'ils n'auraient pas été en mesure de faire avec une route unique. Nous revenons sur ce point au chapitre 6.

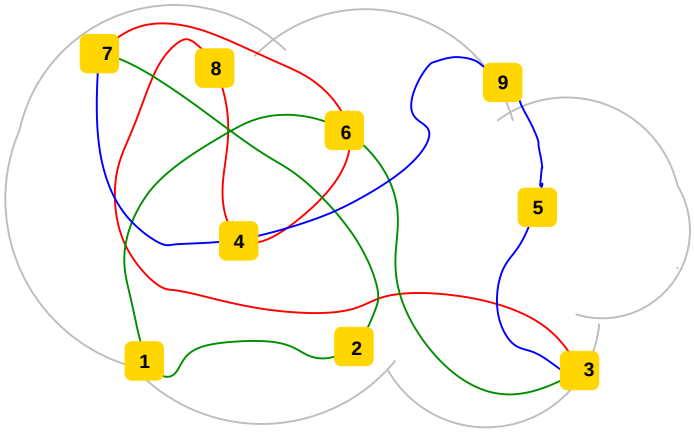


FIG. 4.5 – La communication entre Alice (nœud 7) et Bob (nœud 3) utilise trois routes ({7, 2, 1, 6, 3}, {7, 4, 9, 5, 3} et {7, 6, 4, 8, 3}). Si le lissage de trafic sur les routes limite le débit à, par exemple, 10 kio/s chacune, le trafic sur la communication pourra atteindre 30 kio/s.

4.5 Problème : discrétion

Pour conclure ce chapitre, mentionnons un dernier problème inhérent au lissage de trafic : il est peu discret. En effet, nos connexions lissées ont un profil typique.

Pour notre usage, ce n'est pas gênant : notre but est qu'Eve et Mallory ne puissent pas savoir ce que fait Alice avec Internet, mais nous considérons comme acceptable qu'ils sachent qu'elle utilise un système de non-observabilité. Toutefois, cela rend le système plus difficilement utilisable dans un contexte « autoritaire », par exemple pour accéder à des sites censurés : en effet, comme l'usage du système est détectable, il peut être interdit facilement.

On peut se demander s'il serait possible de modifier le lissage de trafic pour qu'il n'ait pas un tel profil. Cela nous semble difficile : il serait certes possible d'ajouter de nœud en nœud du padding supplémentaire à certains paquets, pour qu'ils n'aient pas tous la même taille, mais la régularité de l'émission resterait visible, de même que l'existence d'une taille minimale fixe pour ces paquets (figure 4.6).

La taille minimale en question semble difficile à dissimuler : en effet, un nœud n'a pas connaissance du contenu d'un paquet. Il ne peut donc pas en extraire les données pour les réorganiser. Tronquer un paquet « en aveugle » revient à le perdre, puisqu'il provoquera une erreur d'authentification (MAC) ou de déchiffrement (bi-IGE, voir la section 5.2 dans le chapitre suivant) à l'arrivée.

Une piste éventuelle serait de dissimuler le trafic par des méthodes stéganographiques dans un flux compatible. Il reste à trouver un tel flux dont la régularité et l'usage fréquent ne soient pas suspects (de la vidéo-conférence, peut-être ?). Pour l'heure la question reste ouverte.

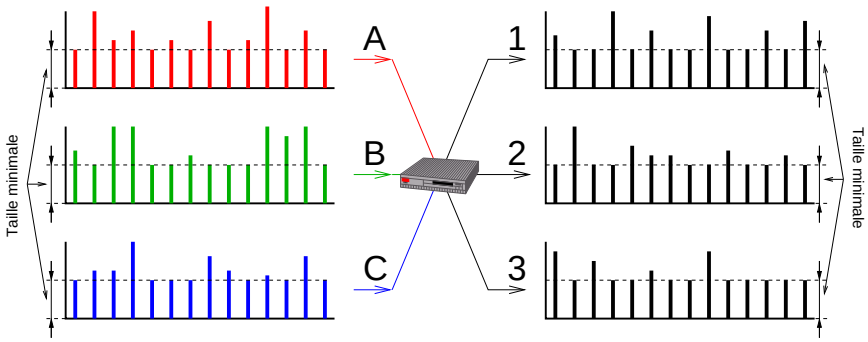


FIG. 4.6 – Même si le nœud N_{i-1} ajoute du padding aléatoire qui est enlevé et remplacé par le N_i, une taille minimale et une fréquence fixe dues au lissage de trafic restent visibles.

5

Protection contre le rejeu de paquet et ses variantes

L'un des moyens dont dispose Mallory pour établir la correspondance entre les connexions sur un nœud consiste à enregistrer un paquet entrant P_e ainsi que ceux qui sortent (P_{s_i}) dans les instants qui suivent. Une couche de chiffrement ayant été enlevée et les paquets étant de taille identique, il ne peut bien sûr pas en déduire cette correspondance immédiatement. Il peut cependant réintroduire le paquet entrant P_e sur la même connexion et regarder sur quelle connexion sort un paquet semblable à l'un des P_{s_i} vus lors de la première phase d'obvervation.

Une variante de cette attaque consiste à se baser sur une des propriétés des modes de chiffrement usuels : si le paquet entrant est modifié, seule une partie du paquet sortant le sera, tandis que la partie non modifiée permettra de l'identifier.

Nous évitons cette attaque d'une part en nous assurant que l'on n'a pas déjà reçu un paquet identique sur la même connexion (cf. 5.1), et d'autre part en utilisant un mode de chiffrement qui nous assure que si l'on reçoit deux paquets partiellement semblables ils seront, après déchiffrement, aussi différents que s'il n'y avait pas eu de ressemblance initiale (cf. 5.2) : ainsi Mallory ne pourra-t'il pas lier les paquets émis la première fois avec ceux de la seconde.

5.1 Empêcher le rejeu de paquet

Pour nous assurer que l'on ne reçoit pas le même paquet, il nous faut garder une trace des paquets reçus. Pour cela, nous avons besoin d'un moyen qui permette de déterminer si un paquet a déjà été vu ou non qui soit à la fois efficace en termes de taille (à cause du nombre de paquets) et en termes de vitesse (il nous faut la réponse très rapidement à cause de la succession des paquets, de la latence et du risque de déni de service).

Bien que ce soit plus un problème d'implémentation qu'un problème fonda-

mental lié à l'anonymat et à la non-observabilité, il nous semble intéressant de décrire ici la méthode que nous avons utilisée. En effet, elle nous paraît élégante et nous a conduit à une amélioration (en termes de sécurité) d'une structure de données connue, les *filtres de Bloom*[1].

Les filtres de Bloom [14] sont des structures de données probabilistes qui permettent de se "souvenir" de quelque chose sans le stocker (comme ils ne stockent pas les données, ils ne permettent pas d'accéder à celles-ci comme des tables de hachage, mais on peut décider si l'on a déjà vu une certaine chose ou non), d'une manière très efficace du point de vue de la mémoire utilisée. Plus formellement, ils permettent de tester si un élément fait partie ou non d'un ensemble. Si l'on est certain de la réponse lorsque le test est négatif, il y a une certaine probabilité d'erreur lorsque la réponse est positive.

Les filtres de Bloom sont une représentation intéressante d'un ensemble en raison de leur efficacité en termes d'espace-mémoire occupé. Ils permettent de plus de faire facilement des opérations telles que l'union et l'intersection d'ensembles. Leur principal défaut est qu'il n'est pas possible de retirer un élément d'un ensemble une fois celui-ci ajouté, ce qui est rédhibitoire dans la plupart des cas, mais pas dans le nôtre. C'est peut-être pour cela qu'ils restent peu connus et peu utilisés (Knuth n'y consacre que trois brefs paragraphes [48, pages 572-573]) ; c'est d'ailleurs pour cela que nous nous permettons de rappeller ci-dessous, en 5.1.1, leur fonctionnement et leurs principales propriétés statistiques.

Il existe diverses évolutions des filtres de Bloom (modifications pour pouvoir supprimer un élément, pour les transformer en tableaux associatifs, etc.), que nous ne considérons pas ici car elles ne sont pas utiles dans notre cas.

5.1.1 Les filtres de Bloom

Un filtre de Bloom est un tableau T de 2^n bits positionnés à 0 quand il est vide, associé à k fonctions de hachage H_i donnant des empreintes sur n bits de leur entrée[2]. Chaque empreinte peut être considérée comme un entier codé sur n bits, et donc comme un *index* du tableau T.

Pour ajouter un élément e au filtre, on hache cet élément avec chacune des k fonctions de hachage et on positionne le bit à l'emplacement $H_i(e)$ du tableau à 1 s'il ne l'est pas déjà.

$$\forall i \in [0, k-1]\,, \quad T\left[H_i(e)\right] \leftarrow 1$$

Pour interroger le filtre, on hache l'élément que l'on cherche (e) avec chacune des k fonctions de hachage et on fait un *et* logique de chacun des emplacements $H_i(e)$. Si le résultat est 1, on déclare que l'élément est dans l'ensemble, s'il est nul on déclare qu'il n'y est pas.

$$\bigwedge_{i=0}^{k-1} T\left[H_i(e)\right]$$

Il n'est pas possible d'enlever un élément d'un filtre.

[1] La solution préconisée n'exclut pas qu'il y ait d'autres approches possibles.

[2] Il s'agit ici de fonctions de hachage au sens algorithmique et non cryptographique du terme.

Si on fait l'hypothèse que les fonctions de hachage renvoient des valeurs sur l'ensemble $[0, 2^n - 1]$ de manière uniforme, la probabilité qu'une fonction de hachage positionne à 1 un certain bit est de $1/2^n$, donc la probabilité qu'elle ne positionne pas à 1 un certain bit est

$$1 - \frac{1}{2^n}$$

Les fonctions de hachage étant indépendantes, la probabilité qu'aucune de ces k fonctions ne le fasse est

$$\left(1 - \frac{1}{2^n}\right)^k$$

Si on insère m éléments, la probabilité que ce bit soit toujours nul est de

$$\left(1 - \frac{1}{2^n}\right)^{km}$$

et donc la probabilité qu'il soit à 1 est

$$1 - \left(1 - \frac{1}{2^n}\right)^{km}$$

Par conséquent, si on teste la présence d'un élément qui n'a pas été inséré dans le filtre, la probabilité d'un faux positif (tous les bits de ses hachés à 1, avec la probabilité ci-dessus pour chacun) est

$$\left(1 - \left(1 - \frac{1}{2^n}\right)^{km}\right)^k \approx \left(1 - e^{\frac{-km}{2^n}}\right)^k$$

(où n est la taille des empreintes des k fonctions de hachage, 2^n est la taille du tableau de bits, et m est le nombre d'éléments ajoutés jusqu'au moment où l'on fait le test.)

5.1.2 Le problème des filtres de Bloom

Le problème avec les probabilités précédentes est qu'elles supposent que les éléments que l'on ajoute sont quelconques, et donc que les fonctions de hachage renvoient des résultats distribués uniformément.

Un attaquant qui pourrait injecter des données peut par contre s'arranger pour remplir plus rapidement le filtre avec des 1, de manière à ce que la probabilité de collision sur les messages en entrée soit plus grande.

Supposons que l'attaquant puisse injecter des messages et que la fonction de hachage soit suffisamment simple pour qu'il puisse l'inverser totalement. Il peut alors générer des messages tels qu'il choisit la valeur renvoyée par chaque fonction de hachage et peut donc injecter, à chaque message qu'il envoie, k « 1 » dans le filtre.

Regardons ce qui se passe alors sur un exemple : soit un filtre F de 2^{16} bits, avec 16 fonctions de hachage. Calculons la probabilité de faux positifs après l'insertion de 4096 messages :

$$\left(1-\left(1-\frac{1}{2^{16}}\right)^{16\times 4096}\right)^{16} \approx 6,4987\times 10^{-4}$$

Imaginons maintenant que les 4096 messages aient tous été créés par l'attaquant. Il a donc pu ajouter $4096 \times 16 = 65536 = 2^{16}$ « 1 » dans le filtre. La probabilité d'avoir un faux positif est alors de 1 ! On est loin du comportement statistique !

Si l'on ajoute les paquets que l'on reçoit au filtre de Bloom et que l'attaquant peut insérer ses propres paquets, il peut alors faire un déni de service.

5.1.3 Modification des filtres de Bloom pour éviter les dénis de service

Nous proposons deux solutions pour éviter ce type d'attaque, chacune ayant des avantages et des inconvénients.

Solution 1

L'utilisation de fonctions de hachage cryptographiques permet d'empêcher l'attaquant de générer des paquets ajoutant à chaque fois le nombre maximal de 1 dans le filtre.

Avantages : deux exécutions avec les mêmes données donnent le même résultat.

Inconvénients : comme deux exécutions avec les mêmes données donnent le même résultat, l'attaquant sait à l'avance ce que chacun des messages qu'il envoie va provoquer. Même si la quantité de données qu'il peut envoyer est limitée (ce qui est le cas dans notre proposition), il peut générer de nombreux messages hors-ligne afin de choisir ceux qui rempliront le plus le filtre. De plus, ces messages produiront le même effet sur deux instances du filtre.

Solution 2

L'idée est simple : lors de la création d'un filtre de Bloom, on génère un *nonce* aléatoire d'une taille assez grande pour ne pas être devinable par force brute. Lors des opérations de hachage, on concatène ce *nonce* avec les données. Si les fonctions de hachage sont cryptographiquement sûres, le résultat sera très éloigné de ce qu'il aurait été sans cette concaténation. Un attaquant ne pourra pas le prédire[3].

L'attaquant pourra toujours insérer des données dans le filtre de Bloom pour augmenter la probabilité de faux positifs, mais il devra en insérer beaucoup plus car il ne pourra pas combattre le comportement statistique du filtre.

[3] Dans ce cas, on n'a pas réellement besoin de toutes les propriétés d'une fonction de hachage cryptographiquement sûre. Par exemple, le fait qu'elle soit à sens unique n'est probablement pas critique ici.

Avantages :

- la sécurité est supérieure à celle de la solution précédente : en effet, l'attaquant étant incapable de savoir quel effet va produire un message sur l'état interne du filtre, il ne peut pré-calculer hors-ligne les messages les plus efficaces ;
- Même dans le cas où l'attaquant trouve par hasard une série de messages efficaces sur une certaine instance d'un filtre, ces messages n'ont pas d'effet particulier sur une autre instance.

Inconvénients : le *nonce* étant généré à chaque exécution, le filtre n'aura pas les mêmes états à chaque exécution sur des données identiques. Si le comportement statistique du filtre est le même à chaque fois, il peut y avoir des variations d'une exécution à l'autre.

Dans notre cas, à part éventuellement pour la reproductibilité de tests lors de la phase de débugage, cet inconvénient n'est pas gênant. Nous avons donc choisi cette deuxième solution, qui rend les attaques hors-ligne inefficaces. Notons de plus que le nombre de paquets que l'attaquant peut introduire est limité par le lissage de trafic.

5.2 Rendre le rejeu de paquet avec modification inutile

Si l'on empêche la circulation des paquets identiques à un paquet qui est déjà passé, Mallory peut changer son attaque en envoyant des paquets modifiés. Le problème vient de ce qu'avec les *modes* habituellement utilisés avec les algorithmes de chiffrement par blocs (voir par exemple le chapitre 9 de [65]), une différence dans un bloc (par rapport à un « original ») donné n'affecte pas les blocs déjà traités, et souvent pas ou peu les blocs suivants (voir l'exemple du mode CBC sur la figure 5.1). La non-propagation des erreurs, qui est en général considérée comme une qualité, est ici un défaut.

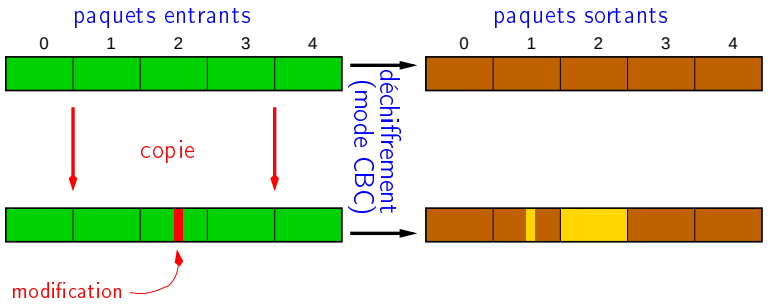


Fig. 5.1 – Avec un mode de chiffrement classique (ici, CBC), Mallory peut copier un paquet et faire une petite modification. Les blocs qui suivent (respectivement précèdent) le bloc où la modification a eu lieu après déchiffrement (resp. chiffrement) seront identiques à ceux du paquet original déchiffré (resp. chiffré).

Il semble difficile de parer cette attaque en examinant chaque paquet pour

voir s'il s'agit d'une variation d'un paquet déjà reçu.

Il est par contre relativement facile de s'assurer que les deux messages sortants – l'original et celui issu de la copie modifiée – soient totalement différents, de telle sorte que Mallory ne pourra plus faire le lien entre eux.

Il doit être possible d'y parvenir en utilisant des transformations tout ou rien (All Or Nothing Transform – AONT), néanmoins ce n'est pas nécessaire : il existe un mode de chiffrement exotique qui convient, bi-IGE.

Le mode bi-IGE, consiste à appliquer deux fois, bi-directionnellement, un chiffrement en mode IGE au message, IGE étant l'*Infinite Garble Extension*, un mode proposé en 1977 par Carl Campbell. La particularité d'IGE est qu'il propage les erreurs ; un bit modifié changera totalement le bloc où se trouve la modification, mais aussi tous ceux traités après. Dans bi-IGE, on fait d'abord une passe dans un sens, ce qui modifie les bits situés *après* une éventuelle modification, puis on surchiffre dans l'autre sens, ce qui règle le problème des bits situés *avant* la modification.

Notons qu'IGE a été « généralisé » par Knudsen en 2000 [47] dans un mode appelé ABC, pour *Accumulated Block Chaining*, qui garantit l'intégrité du message, ce qui évite le recours à un MAC. L'usage de ce mode pourrait être étudié pour une future version de notre système (il faudrait bien sûr toujours faire un « bi-ABC »), les problèmes soulevés dans [42, 7] ne semblent pas gênants pour notre usage.

5.3 Implications sur l'architecture du système

Une protection contre le rejeu de paquets a diverses implications sur l'architecture du système de non-observabilité, en particulier sur la nécessité de gérer la perte de paquets.

5.3.1 Les cas possibles

Il existe plusieurs possibilités selon que l'on autorise Alice à envoyer ou non des paquets dupliqués (sachant que de toute façon, ceux-ci ne doivent pas aller au-delà du nœud suivant de la route). En effet :

1. soit Alice s'assure qu'elle n'envoie aucun paquet identique à un paquet déjà envoyé. C'est une opération relativement lourde, puisqu'elle doit vérifier non seulement les paquets qu'elle envoie *effectivement*, mais aussi ce qu'ils seront après chaque relai, c'est-à-dire entre chaque couche de chiffrement. Si un paquet est perdu pour une cause « naturelle » (engorgement réseau, etc.) entre les nœuds i et $i+1$, il doit bien sûr être retransmis, il y a là encore deux choix :
   (a) il est retransmis par le nœud i, avant la perte : cela suppose une liaison fiable entre ce nœud et le nœud $i+1$, ce que nous avons rejeté dans le chapitre précédent pour incompatibilité avec le lissage de trafic.
   (b) il est retransmis par Alice. Le paquet doit être différent, donc son chiffrement aussi. Cela implique simplement, du point de vue cryptographique, de changer le vecteur d'initialisation. Par contre, cela implique également que la couche de chiffrement doit être en dessous de celle qui assure la fiabilité.

2. soit Alice envoie les paquets sans se préoccuper de la présence d'éventuels « doublons ». Ceux-ci seront alors rejetés par la protection contre l'analyse de trafic du premier nœud (à juste titre puisqu'autrement Eve aurait pu directement faire la même observation que veut faire Mallory quand il introduit volontairement des doublons). Il faut donc qu'une couche située à un niveau supérieur se charge de retransmettre si nécessaire les données contenues dans ces paquets.

5.3.2 Position de notre solution

Notre solution basée sur des filtres de Bloom durcis (ou, a priori, toute solution probabiliste) est difficilement compatible avec les solutions 1.a et 1.b du paragraphe 5.3.1, puisque celles-ci supposent qu'Alice peut contrôler les données de la même façon que les nœuds. Or, avec nos filtres durcis selon la solution 2 exposée en 5.1.3, les filtres d'Alice et celui créé par les nœuds ne le seront pas avec le même *nonce*. Un faux positif sur l'un ne le sera pas sur l'autre et donc la vérification par Alice n'est pas possible. On peut bien sûr imaginer que l'un des premiers paquets émis par Alice contienne une copie du *nonce*, mais cela pose d'autres problèmes : avant l'établissement d'une session chiffrée, Mallory peut prendre connaissance du *nonce*, qui devient alors inutile ; après, cela implique que la protection n'est pas en place immédiatement. Étant donné l'existence de solutions alternatives (notre proposition marche très bien avec le cas 2 du paragraphe 5.3.1), il ne nous semble pas nécessaire de poursuivre d'avantage l'étude des « solutions » du cas 1.

Par ailleurs, dans presque tous les cas (sauf le 1.a), un paquet perdu, que ce soit parce qu'il a été rejeté par un de ces mécanismes de protection ou à cause d'un problème de réseau, doit être renvoyé si nécessaire *chiffré différemment*. En effet, s'il était chiffré de la même manière, il serait rejeté par lesdits mécanismes. Cela implique donc que, dans un modèle en couches, le chiffrement doit être *sous* la fiabilité.

Cela renforce donc l'intérêt de notre idée, exposée au chapitre précédent, d'avoir un protocole où la fiabilité est assurée au-dessus de la couche de non-observabilité.

6

Bref aperçu du protocole *True Nyms* et analyse de sécurité

6.1 Le protocole *True Nyms*

True Nyms est un protocole qui utilise les différents moyens décrits dans les chapitres précédents pour fournir une couche de non-observabilité générique.

Pourquoi avoir un *protocole* et pas simplement une *implémentation* des idées en questions ? Principalement parce que les idées que nous avons décrites peuvent être implémentées de multiples façons ; pour donner un exemple simple, il y a de nombreux paramètres, par exemple ceux relatifs au lissage de trafic : la taille des paquets et la fréquence d'émission. Ce genre de choix a une influence sur les performances et sur la sécurité.

D'autre part, ce que nous avons décrit précédemment sont nos idées pour résoudre le problème de la non-observabilité, c'est loin d'être un tout cohérent : on peut les voir comme un nouveau type de briques blindées. Le but du protocole *True Nyms* est alors de faire les *plans* qui expliqueront comment construire, avec non seulement toutes ces briques mais aussi d'autres éléments indispensables (avez-vous déjà vu une maison constituée *uniquement* de briques ?), un bâtiment solide, fonctionnel et capable de résister aux attaques. Finalement, l'implémentation peut être vue comme le bâtiment lui-même, réalisé d'après le plan.

Notre but dans ce chapitre n'est pas toutefois de décrire de manière exhaustive ce protocole — ce document est une thèse, pas une RFC, et de plus il est encore expérimental et donc destiné à changer — mais simplement d'en donner un aperçu au lecteur, aussi ne décrirons-nous pas le format de chaque paquet bit par bit. C'est ensuite la sécurité de ce protocole que nous évaluerons dans la section 6.2.

Il ne faut bien sûr pas penser que nous avons écrit le protocole et qu'une fois

cette étape faite nous nous sommes attelés à son implémentation. Les deux ont été développés en parallèle, avec une rétroaction permanente de l'un sur l'autre, de manière à aboutir à un protocole sûr, qui puisse également être implémenté de manière sûre ! C'est d'ailleurs pourquoi ce chapitre fera souvent référence à l'implémentation et à son développement.

Il faut également noter que sur un certain nombre de points, le protocole n'a aucun pouvoir de contrainte sur les implémentations et ne peut donc que se contenter de faire des suggestions. En effet, alors que par exemple sur le format des paquets, le non-respect du protocole empêchera l'interopérabilité, ce n'est aucunement le cas pour ce qui est, disons, de la création des routes leurres.

Pour la création de ce protocole, nous nous sommes efforcé de ne pas réinventer la roue, mais, au contraire, d'utiliser des méthodes *tried and true* de diverses origines, n'inventant de nouvelles choses que pour protéger la non-observabilité.

La première question à résoudre était celle de l'emplacement du protocole dans la pile réseau. C'est le thème de la section qui suit.

6.1.1 Une question de couche

Nous avions conclu la section 4.3 sur la nécessité de placer la couche de fiabilité au-dessus de celle de non-observabilité.

Cela implique donc que, dans le modèle IP, il n'est pas possible de se placer au-dessus de TCP pour bénéficier de la fiabilité.

Conceptuellement, nous devons donc nous placer au niveau 3,5-4 du modèle TCP/IP (cf. figure 6.1, à gauche), car notre protocole peut :

- se placer entre la couche IP et la couche transport (d'où le « 3,5 », par analogie avec MTLS qui est souvent qualifié de « protocole de niveau 2,5 ») ;
- aussi fournir le transport (d'où le 4).

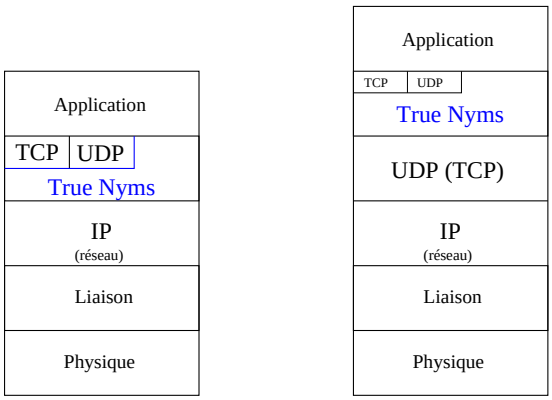


FIG. 6.1 – Emplacement conceptuel (à gauche) de True Nyms dans la pile IP. Pour des raisons pratiques (*e.g.*, compatibilité avec les firewalls actuels et franchissement de ceux-ci), il s'agit en réalité d'une surcouche au-dessus d'UDP ou de TCP (à droite).

Dans la pratique, pour des raisons de compatibilité avec l'existant, il vaut mieux ne pas se placer directement au-dessus d'IP, mais au-dessus d'UDP : en

effet les équipements réseau peuvent accepter plus facilement un nouveau protocole sur UDP (il s'agit d'indiquer les ports) qu'au-dessus d'IP. L'inconvénient principal de cette approche est que l'en-tête UDP crée un *overhead* supplémentaire. Le protocole *True Nyms* pourrait être facilement modifié pour être utilisé directement au-dessus d'IP (il faudrait ajouter une *étiquette* entre chaque paire d'hôtes successifs d'une route afin d'identifier les flux).

Nous nous sommes également demandé s'il valait mieux recréer un protocole fournissant des services similaires à ceux de TCP au-dessus de la couche de non-observabilité, ou simplement fournir un service qui encapsule un protocole de niveau supérieur et l'utiliser pour encapsuler TCP quand une liaison fiable est nécessaire. La question se pose du fait que l'on n'a pas besoin de toutes les fonctionnalités de TCP (par exemple, la création d'une session est prise en compte par la couche non-observabilité elle-même, la gestion du débit n'est pas possible à cause du lissage de trafic, etc.).

Comme nous avons décidé de réaliser notre implémentation de test en espace utilisateur (cf. 7.1), dans un cas comme dans l'autre, le traitement se fait dans l'*espace utilisateur* du système d'exploitation.

Nous avons étudié l'implémentation de TCP du noyau d'OpenBSD, qui est réputée pour sa qualité, publiée avec une licence permissive et bien documentée puisqu'elle est encore assez proche de celle décrite dans [80].

Nous sommes arrivé à la conclusion que ce serait une meilleure chose de ne garder de TCP que les idées directrices, et de réimplémenter un nouveau protocole de fiabilisation du trafic, qui pourrait se substituer à TCP (cf. figure 6.1, à droite).

Pour être générique, notre protocole se devait d'être également à même de « simuler UDP », c'est-à-dire permettre la transmission non fiable d'information.

Nous avons donc décidé de créer un tel protocole intégrant à la fois non-observabilité et fiabilisation *optionnelle* des données ; la section suivante décrit un peu plus ce protocole. Bien sûr, ce choix d'un protocole spécialisé n'empêche pas d'encapsuler un protocole standard par-dessus celui-ci.

6.1.2 Au niveau réseau

Notre protocole est nouveau et conçu pour fournir de nouveaux services, toutefois une personne familière avec les protocoles réseaux existants retrouvera vite ses repères : notre protocole s'inspire ou utilise l'existant aussi souvent que possible.

En ce qui concerne le format des paquets, notons simplement qu'ils commencent par une valeur aléatoire et se terminent par un MAC ou un haché cryptographique selon que la session a été établie ou non. La valeur aléatoire change si le paquet est retransmis (et donc le MAC/haché également), ce qui assure que si un paquet est retransmis le résultat du chiffrement sera différent et donc qu'un observateur ne pourra pas faire le lien entre les différentes émissions.

Les paquets contiennent des informations de type numéro de paquet / numéro d'aquiescement à la manière de TCP afin de gérer les pertes.

Comme pour TCP, tant qu'un paquet n'a pas été acquiescé (réception d'un ACK), il faut le garder en mémoire pour pouvoir le retransmettre le cas échéant. La latence a priori plus grande qu'avec TCP fait que les acquiescements (ACK) mettront plus de temps à nous parvenir ; pour pouvoir continuer à émettre, il

faut donc des tampons d'émissions plus grands que ceux de TCP. Dans la pratique, cela n'est pas un problème étant donné la capacité mémoire des machines actuelles.

Contrairement à ce qui se fait pour les protocoles normaux comme TCP, dans notre contexte, **il vaut mieux qu'Alice ne réagisse pas immédiatement à un paquet** si celui-ci implique une réponse de sa part (y compris un simple ACK) : en effet, un nœud d'une route pourrait mesurer la latence entre un paquet qu'il envoie et l'aquiescement de celui-ci pour en déduire une approximation du nombre de nœuds qui le séparent d'Alice.

Une possibilité pour empêcher cela serait que, pour chaque connexion, Alice attende un petit délai avant de répondre, ce délai étant choisi aléatoirement pour chaque connexion mais de manière non uniforme : plus le nœud de la connexion en question est loin d'Alice sur la route, plus le délai devrait être bref.

Établissement d'une connexion

L'établissement de connexion dans True Nyms commence par un mécanisme en trois étapes inspiré de celui de TCP (de type SYN, SYN/ACK, ACK), à ceci près que les mécanismes de type *syncookie* sont intégrés dès le départ et non pas ajoutés après coup comme dans le cas de TCP. Contrairement à ce qui se passe dans TCP où la taille de ceux-ci a été limitée par l'existant, ce n'est pas le cas ici et True Nyms utilise un champ plus grand (160 bits dans la version actuelle) et sans rapport avec un numéro de session comme dans TCP.

Une fois cette étape franchie, les deux parties se mettent d'accord sur une clef, au moyen de l'algorithme de Diffie-Hellman tel que décrit en 3.1.3.

Les messages subséquents sont chiffrés avec la clef et l'algorithme négociés, en mode IGE (notre implémentation ne supporte que l'AES pour le moment).

Une fois la connexion établie

Une fois la connexion établie, plusieurs possibilités sont offertes, les principales étant :

- diverses options relatives à la gestion du réseau pair-à-pair (récupérer une liste de nœuds, etc.) ;
- la connexion à un autre nœud (c'est-à-dire simplement le relais des paquets d'Alice vers une adresse) ou à une machine n'étant pas un nœud *True Nyms* (création d'une connexion TCP ou UDP vers cette machine et conversion/transmission des données) ;
- la coupure de la route au niveau de ce nœud (si une connexion sortante a déjà été établie), c'est-à-dire l'arrêt du relais des données ;
- la possibilité de fusionner la connexion en cours avec une autre (routes multiples – à condition que l'on n'ait pas établi de connexion sortante) ;
- la possibilité de changer la clef de session (permet de prolonger la durée de vie d'une session).

Un nœud doit bien sûr garder en mémoire l'état dans lequel se trouve chaque connexion, de manière à ne proposer que les possibilités autorisées par l'état actuel.

UDP contre IP

Nous l'avons dit, le protocole actuel utilise UDP et non IP directement. Si l'on voulait utiliser directement IP, il suffirait que chaque nœud ajoute en en-tête du paquet une sorte d'étiquette à la MPLS, que le nœud suivant enlèverait et remplacerait par la sienne. Cela permettrait à une paire de nœuds de distinguer des connexions s'il y en avait plus d'une entre eux (avec UDP ce sont les numéros des ports qui jouent ce rôle).

6.1.3 Paramètres

Notre système comporte divers paramêtres. Si certains sont du ressort de l'implémentation, d'autres doivent être fixés au niveau du protocole pour des raisons d'interopérabilité : ce sont ces derniers qui nous intéressent ici.

Ces paramêtres ont été établis intuitivement, avec pour certains un retour d'expérience venant de nos tests. Ils ne sont en aucun cas optimaux (et pour certains d'entre eux, l'optimalité n'existe pas : leur adéquation dépend du type d'application qui passe au-dessus de *True Nyms*).

L'annexe A donne les valeurs des paramètres utilisées dans notre prototype.

6.2 Sécurité

Il est légitime de s'intéresser à la sécurité de la non-observabilité fournie par *True Nyms*.

Il n'est probablement pas possible, malheureusement, de fournir une « preuve » de sécurité absolue, mais uniquement de montrer la résistance face aux attaques connues.

Outre les attaques déjà mentionnées précédemment, il nous semble qu'il y en a une nouvelle à laquelle il faut prendre garde : c'est celle que nous appellerons l'*attaque par corrélation globale*.

6.2.1 Attaque par corrélation globale

Pour chaque paire de nœuds, un observateur global sait en permanence si des données sont échangées entre ces nœuds (au niveau IP, sous la couche de non-observabilité bien sûr). Peut-il en déduire des informations sur les communications qui passent au-dessus de la couche de non-observabilité ?

On peut modéliser le problème par un graphe : les nœuds du réseau sont les sommets du graphe, et il y a une arête entre deux nœuds si des données passent entre eux au niveau IP. Avec un tel graphe à un instant donné, tout ce qu'Eve peut déterminer, c'est qu'il n'y a *pas* de communications entre des nœuds appartenant à des parties disjointes du graphe.

En fait, la longueur d'une route est finie, par conséquent Eve peut même conclure à l'absence de communication entre deux nœuds s'il n'y a pas de chemin de longueur inférieure à N dans le graphe, où N est une borne supérieure sur la longueur des routes.

Le problème maintenant est que chaque connexion a une durée de vie moyenne D_c. Si Eve prend périodiquement des « instantanés » du réseau, peut-elle en déduire des informations supplémentaires sur les communications qui passent par *True Nyms* ?

Pour la plupart des usages, la durée de vie d'une communication n'est pas supérieure à celle d'une connexion ; la communication n'existera donc que sur un instantané et Eve ne pourra rien apprendre à son sujet.

Il existe cependant des communications pour lesquelles la durée de vie est plus longue : connexions `ssh` ou `BGP`, transferts de gros fichiers, etc. Bien entendu, les routes entre les participants d'une telle communication changent au fil du temps, mais Eve ne pourrait-elle pas s'apercevoir qu'il existe un chemin (de longueur inférieure à N) sur plusieurs de ses instantanés et en déduire l'existence d'une communication ?

La réponse à cette question dépend de la probabilité d'existence d'un chemin. Chaque nœud est, en moyenne, à l'origine de k routes (leurres et réelles confondues) de longueur moyenne L. En dehors de ceux d'origine et de destination, les nœuds d'une route sont choisis aléatoirement. Nous conjecturons que notre graphe a des propriété similaires à un graphe aléatoire : si c'est effectivement le cas, le théorème d'Erdös nous confirme l'existence de chemins courts entre chaque paire de nœuds. Il serait intéressant de démontrer cela de manière formelle.

Notons que souvent, les communications de longue durée n'ont pas de contraintes fortes en termes de débit et de latence : il serait donc possible d'augmenter la longueur des routes qui composent la communication lors de leur renouvellement pour rendre la détection beaucoup plus difficile.

Good ideas are not adopted automatically. They must be driven into practice with courageous impatience.

— Amiral Hyman Rickover

Writing a book is a little more difficult than writing a technical paper, but writing software is a lot more difficult than writing a book.

— Donald Ervin Knuth *in* [49]

7 Implémentation et évaluation

Afin de valider les idées décrites dans les chapitres précédents[1], nous avons implémenté[2] le protocole *True Nyms* sous la forme d'un démon Unix, écrit en langage C.

Dans ce chapitre, nous présentons d'abord notre implémentation. Bien que nous soyons persuadé que la programmation peut être un Art, il n'est pas question ici de décrire cette implémentation dans tous ses détails. Le code-source, imprimé, ferait plus de 300 pages à lui seul, et il nécessiterait de fréquentes explications complémentaires : il s'agit de programmation système / réseau parfois complexe. C'est pourquoi nous ne décrivons ici que l'architecture générale de l'implémentation, argumentons certains choix qui ont été faits et présentons quelques points saillants, un peu comme nous l'avons fait dans le chapitre précédent pour le protocole.

Dans un second temps, nous étudions les performances de cette implémentation.

7.1 Notre implémentation

Cette implémentation est située dans l'espace utilisateur ; elle a ainsi l'avantage d'être relativement portable (sur à peu près tous les systèmes Posix) et plus simple à écrire : notre objectif était de faire un prototype qui soit néanmoins robuste. Elle fait un peu moins de 20 000 lignes de code C.

On peut cependant imaginer qu'à terme, un protocole de ce type soit plutôt implémenté comme l'est généralement IPsec, c'est-à-dire avec la partie gestion des communications établie dans le noyau, et l'établissement des routes, qui

[1] Il s'agit bien sûr d'une validation sur le plan des performances et de la faisabilité. La sécurité n'est nullement validée par une implémentation.

[2] Au cours du printemps et de l'été 2007, nous avons été assisté dans cette tâche par Benoît Mouthon, alors en stage de master [56].

est conceptuellement séparée, gérée par un démon en espace utilisateur. Cette seconde approche serait bien sûr moins portable, mais permettrait de bénéficier de manière plus efficace d'éventuels accélérateurs cryptographiques matériels.

Dans notre implémentation, les applications communiquent directement avec le démon en utilisant un protocole spécial ; néanmoins, pour éviter d'avoir à modifier tous les programmes, un proxy SOCKS spécial permet de faire l'interface entre les applications qui supportent SOCKS (elles sont nombreuses) et le démon.

Il serait possible d'imaginer un mécanisme d'interposition de bibliothèque pour utiliser les programmes qui ne supportent pas SOCKS sans les modifier (du moins quand ils sont liés dynamiquement). Il serait également possible d'imaginer un proxy qui s'interfacerait avec le système d'exploitation de manière à faire apparaître une interface réseau virtuelle : toutes les communications envoyées sur cette interface passeraient alors par `truenyms`.

Le choix du langage C Nous l'avons dit, notre implémentation est réalisée en langage C (ISO C 99), utilise les appels systèmes POSIX et la partie cryptographique (`libcrypto`) de la bibliothèque OpenSSL.

Le choix d'un langage tourne souvent à la guerre de religion et nous n'avons pas l'intention d'entrer dans ce débat. Nous nous contenterons donc de citer ici quelques éléments qui ont motivé notre choix :

- C permet un contrôle de bas niveau de la machine : le programmeur peut savoir ce qui se passe dans son programme. Ce point n'est pas toujours un avantage : le programmeur doit ainsi gérer lui-même la mémoire, mais dans notre cas où une paranoïa aigüe est nécessaire, c'est un avantage de savoir qu'aucun *garbage collector* ne se déclenchera dans une situation plutôt que dans une autre, retardant une émission réseau de quelques millisecondes et donnant ainsi une information capitale à un attaquant !
- la nécessité de faire confiance à du code écrit par quelqu'un d'autre est moindre ;
- C offre un accès direct à certaines fonctions évoluées du système qui dans un autre langage disparaissent souvent dans la couche d'abstraction (par exemple, le transfert de descripteurs de fichiers entre processus via les messages de contrôle des sockets Unix, dont nous reparlerons en 7.1.2) ;
- enfin, *last but not least*, c'est un langage dont nous avons une connaissance approfondie [73].

7.1.1 Architecture générale du démon `nymsd`

Il nous semble que dans la conception d'un démon réseau, la sécurité doit être primordiale. Pour restreindre les effets sur la machine d'une faille dans notre démon, celui-ci est conçu pour fonctionner sans privilèges particuliers, dans une cage *chroot*.

De plus, il est architecturé autour de plusieurs processus, dans l'idée que même si l'un d'eux est compromis, un attaquant ne puisse obtenir d'informations sur les connexions autres que celles gérées par le processus attaqué.

Processus...

Le découpage en divers processus est destiné à compartimenter autant que possible l'information pour réduire l'impact d'une faille de sécurité potentielle (c'est d'ailleurs pour cette raison que nous avons préféré utiliser des processus plutôt que des *threads*).

Les principaux processus sont les suivants :

- un processus `listener` qui écoute sur la socket réseau. Ses principaux rôles sont :
  - d'attendre les connexions : il se charge de la première partie de l'établissement de connexion (gestion des syncookies). Si cette première partie réussit, il crée un processus fils pour gérer cette connexion,
  - la séparation des données : True Nyms est au-dessus d'UDP et ne bénéficie pas de la séparation des différentes connexions faite automatiquement quand l'on utilise TCP (appel système `accept`). Ce processus s'en charge donc et transmet les données au processus fils qui correspond,
  - il attend également les connexions des clients qui désirent utiliser True Nyms. Il crée un processus `smartprovider` pour gérer une telle connexion, quand elle survient ;
- des processus destinés à gérer les connexions. Ils sont créés par le processus `listener`. Chacun de ces processus connaît la clef de session de la connexion qu'il gère, et il est le seul à la connaître ;
- un processus `writer`. Il centralise les écritures sur le réseau. Un processus qui veut écrire lui transmet les données. Nous revenons sur ce processus en 7.1.2 ;
- un processus `poolmanager` qui est chargé de la gestion du pool de routes préétablies. Les autres processus lui donnent également les routes et connexions qui ne sont plus utiles, et ce processus se charge de les arrêter quand il juge le moment opportun (cf. 3.3) ;
- des processsus `smartprovider` (créés par le processus `listener`) : ce sont eux qui s'occupent des clients. Quand un client veut envoyer des données, ce processus demande une ou plusieurs routes au processus `poolmanager`, le connecte à la destination et se charge de transférer les données du client sur cette communication et inversement ;
- un processus qui s'occupe du réseau pair-à-pair et en particulier de maintenir une liste de nœuds.

La figure 7.1 représente les principaux chemins suivis par les données des routes au sein des processus du démon `nymsd`.

L'un des avantages des processus lourds sur les threads, en termes de sécurité, est que la mémoire et les sockets ne sont pas partagées. C'est aussi un inconvénient puisqu'il faut bien que le processus `writer` y ait accès pour pouvoir écrire. Et le processus writer ne peut hériter des sockets puisqu'il est a priori créé avant la plupart des communications. Il existe heureusement une solution simple bien que peu connue : il est possible de faire passer un descripteur d'un processus à un autre (sans qu'ils soient apparentés) via les messages de contrôle des sockets Unix.

Notre démon fait fréquemment usage de cette possibilité, notamment :

- pour le passage au processus `writer` des sockets sur lesquelles le lissage du trafic sortant doit être effectué ;
- pour le passage des routes pré-établies du gestionnaire de pool aux pro-

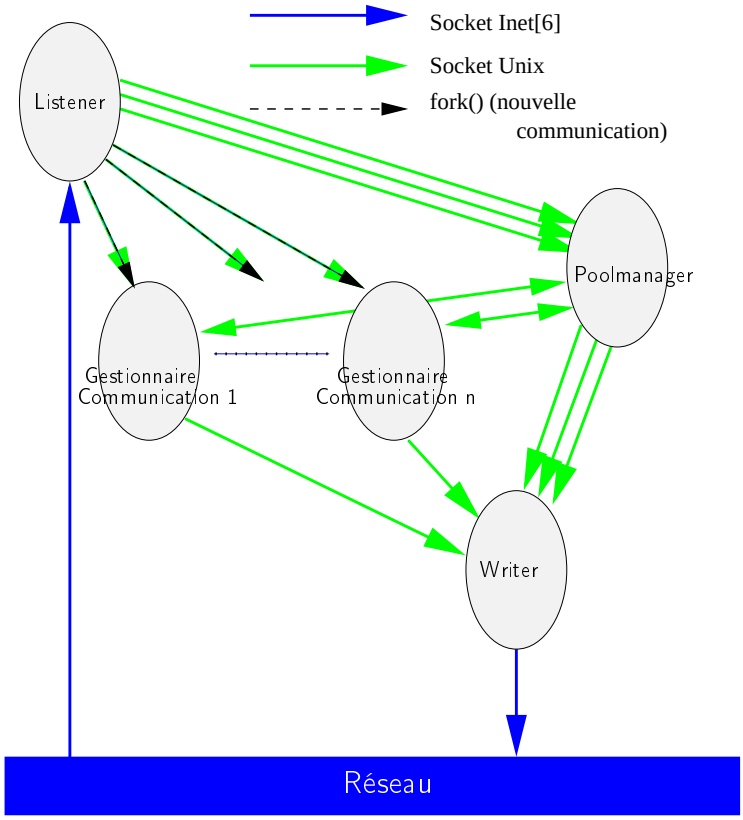


FIG. 7.1 – Principaux chemins des données dans `nymsd`.

cessus qui en ont besoin ;
- pour le passage des routes qui ne sont plus utiles au gestionnaire de pool afin qu'il se charge de désynchroniser la destruction de la route.

... et modules

Les processus du démon sont architecturés autour de modules qui regroupent les fonctions et les structures de données. On trouve principalement deux types de modules (sans qu'il n'y ait toutefois de séparation nette entre les deux, il existe aussi des modules « intermédiaires ») :
- des modules relativement génériques qui sont en quelque sorte des « mini-bibliothèques » de fonctions et pourraient facilement être utilisés dans un autre projet. Certains de ces modules[3] viennent d'ailleurs d'autres de nos projets et / ou sont partagés avec le cache DNS dont traite la seconde partie ;
- des modules qui constituent le coeur logique du programme (plus ou moins un module pour chaque type de processus décrit précédemment) : ils sont

[3] Par exemple des modules de gestion du réseau, de chargement de fichiers de configuration, etc.

généralement assez indépendants les uns des autres et utilisent les fonctions définies dans les modules précédents.

Certaines techniques de programmation orientée objets sont utilisées quand cela permet une plus grande clarté ou modularité du code (C n'est pas orienté objets, mais cela n'empêche pas d'utiliser certaines techniques traditionnellement associées avec l'orientation objets, voir par exemple [67] ; le lecteur familier avec Perl 5 peut faire un parallèle avec la programmation orientée objets dans ce langage).

L'intérêt de cette approche est qu'elle permet de faire naturellement de la programmation défensive.

7.1.2 Le processus `writer`

L'usage d'un processus `writer` unique pour faire les écritures permet d'assurer que les diverses communications se comportent de la même façon, même en cas de fonctionnement anormal du programme ou du système d'exploitation.

Même si un problème survient au niveau de l'ordonnancement des processus (causé par l'attaquant ou non), les écritures sur les communications sortantes continuent à se faire de manière séquentielle, puisque c'est un unique processus qui les fait. Le même comportement est observé si l'un des processus qui gère les communications est occupé et ne peut transmettre un paquet à un moment donné : le processus d'écriture transmettra un paquet leurre au moment opportun. Si jamais un problème survient dans ce processus `writer`, il affectera alors toutes les communications et non une unique : là non plus il ne sera pas possible d'obtenir de l'information. Notons d'ailleurs que le processus en question est particulièrement simple, ce qui réduit d'autant le risque de bugs à cet endroit.

7.1.3 Le processus `listener` et l'utilisation d'UDP

Les connexions utilisent, sauf exception[4], une unique socket UDP, la lecture de cette socket et le tri des messages étant effectués par le processus `listener`.

Cela implique entre autres que si l'un des processus qui gére une connexion connaît une avarie, celle-ci ne sera pas apparente à un observateur, puisque la fiabilité est située au-dessus de la non-observabilité dans `True Nyms`.

En effet, le processus `listener` va continuer à lire les paquets sur la socket UDP, quitte à les détruire si le processus auquel il devrait les passer n'est pas en état de les recevoir. À l'inverse, si chaque processus utilisait sa propre socket (en imaginant que ce soit possible en UDP) et que le gestionnaire de la communication (et donc de la socket en question) ne pouvait pas lire [tous] les paquets, Eve pourrait s'en apercevoir à l'émission de paquets ICMP (dans le cas d'UDP) ou à la réduction de la taille de la fenêtre (dans le cas de TCP).

7.1.4 Notes diverses sur l'implémentation

Outre ce qui a déjà été mentionné, notons que l'implémentation dispose de fonctionnalités de filtrage IP qui peuvent être utilisées pour interdire les redirections vers certaines adresses ou plages d'adresses. Cela permet par exemple

[4] L'usage de TCP est possible, par exemple si un firewall bloque UDP.

d'utiliser le programme en interdisant l'accès aux adresses d'un réseau interne par son intermédiaire (en contournant le firewall).

Cette implémentation devrait être utilisable directement ou avec des adaptations mineures sur la plupart des systèmes Unix/POSIX récents. Nous l'avons testée avec succès sur des systèmes OpenBSD, FreeBSD et Linux récents.

7.2 Évaluation des performances

Dans cette section, nous donnons les résultats de mesures de performances effectuées sur un réseau de tests.

7.2.1 Procédure de test

Tout au long du développement, nous avons testé notre prototype sur un réseau local (LAN), pour vérifier son bon fonctionnement.

Toutefois, la latence entre deux machines sur un LAN est très inférieure à celle que l'on observe entre deux machines sur l'Internet. Les limitations de débit et pertes de paquets ne sont pas les mêmes non plus.

Ce pourquoi, afin d'avoir des tests de performances valables, nous avons établi un réseau de tests distribué entre différents emplacements.

Notre réseau de tests

Notre réseau de tests est constitué de machines placées dans les endroits suivants :

- à l'École Normale Supérieure de Lyon (France) ;
- au Laboratoire d'Informatique de Grenoble (Antenne de Montbonnot, France) ;
- à l'Université de Liège (Belgique) ;
- à l'Université du Luxembourg (Luxembourg) ;
- à l'Université de Namur (Facultés Universitaires Notre-Dame de la Paix, Belgique) ;
- à l'Université Technique de Berlin (Technische Universität Berlin, Institut für Mathematik, Allemagne) ;
- à notre domicile, sur une connexion ADSL de Free (France).

Nous souhaitons d'ailleurs exprimer notre gratitude à toutes les personnes qui ont accepté d'héberger ces machines et rendu le déploiement d'un tel réseau possible.

On notera que ce réseau de tests comporte principalement des universités européennes, ce qui peut donner des résultats légèrement biaisés, celles-ci étant liées les unes aux autres par l'intermédiaire du réseau Géant2[5], généralement via un réseau national (Belnet en Belgique, Renater en France, Restena au Luxembourg, etc.). Toutefois, comme le montre la figure 7.2, la topologie au niveau *systèmes autonomes* (AS) reste typique de l'Internet.

Les ordinateurs utilisés comme nœuds de ce réseau sont des machines basse consommation et à faible coût, mais également de puissance assez faible. Ces

[5] Voir le site `http://www.geant2.net` pour plus de détails sur la topologie de ce réseau.

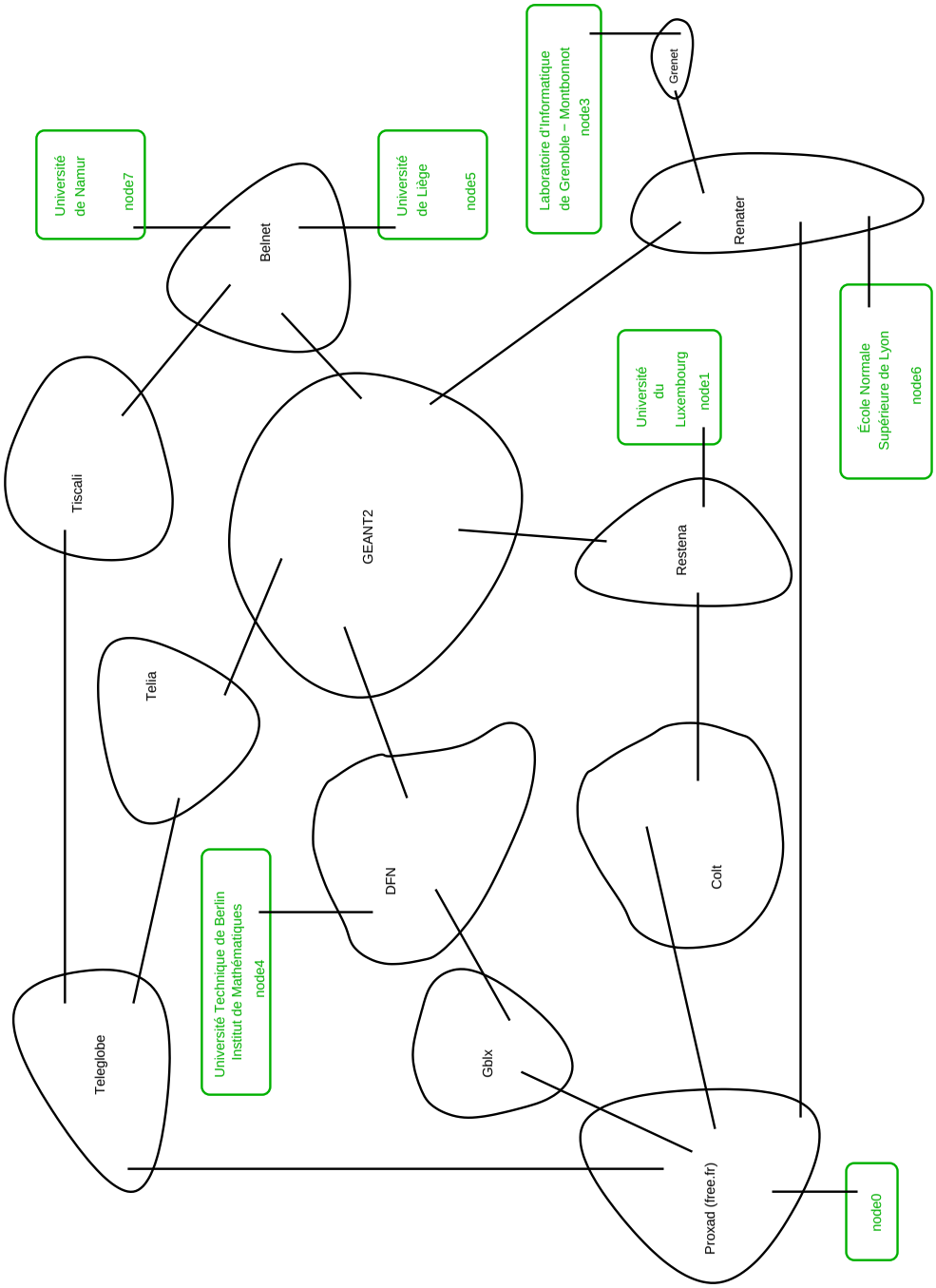


FIG. 7.2 – L'Internet est une interconnexion de *systèmes autonomes*. Voici un aperçu des interconnexions qui séparent les nœuds de notre réseau de tests.

machines sont équipées de processeurs AMD Geode LX cadencés à 433MHz et de 256Mio de RAM. Le système d'exploitation utilisé lors des tests est OpenBSD (version 4.2), installé en mémoire flash (les machines ne disposent pas de disque dur). Le démon *True Nyms* est téléchargé pour les tests via *ssh* et stocké en RAM pour la durée de ceux-ci.

Notons que si le processeur est cadencé à une basse fréquence par rapport aux standards actuels, il dispose en revanche d'instructions de chiffrement AES et d'un générateur de nombres aléatoires intégrés, utilisés de manière transparente par le couple OpenBSD / OpenSSL.

Ces machines ont été financées par le Fonds National de la Recherche (FNR, Luxembourg) dans le cadre du projet TeSeGrAd.

Si un réseau de cette taille n'offre pas un anonymat fiable, il permet néanmoins de faire des mesures de latence sur une route « de taille réelle » au travers de l'Internet.

Pour pouvoir faire certaines mesures sur un plus grand nombre de nœuds, nous avons, le cas échéant, placé plusieurs machines[6] sur notre réseau de tests à l'Université du Luxembourg. Dans ce cas, nous avons utilisé un algorithme de création de routes qui s'assure que deux nœuds successifs d'une route ne se trouvent pas sur le même sous-réseau, afin que les performances bien évidemment supérieures du LAN ne viennent pas biaiser les mesures.

Mesures

Les mesures sont effectuées, selon les cas :
- soit en instrumentant directement le démon *True Nyms* ;
- soit par l'utilisation de clients spécifiques.

Dans tous les cas, l'idée est d'avoir une collecte de mesures automatisée de manière à pouvoir laisser le système fonctionner sans surveillance particulière. Cela permet de collecter un échantillon de mesures de taille significative à partir duquel on peut extraire quelques données statistiques simples comme la moyenne et l'écart type. À partir de ces données, des graphiques sont tracés afin d'obtenir un aperçu visuel des performances et de l'impact des différentes idées sur celles-ci.

7.2.2 Établissement des routes

Le temps d'établissement des routes est un point important. En effet, il correspond au temps d'établissement des communications, puisqu'on considère une communication comme établie quand la première route l'est : c'est en effet à partir de ce moment qu'Alice et Bob peuvent commencer à échanger des données.

Il est intéressant de voir comment ce temps d'établissement varie en fonction du nombre de nœuds d'une route, ainsi que l'impact de notre *pool de routes*, puisque celui-ci était destiné à diminuer ce temps.

[6] Il s'agissait alors de machines d'un modèle différent de celui décrit précédemment (machines de bureau de récupération) mais de puissance comparable (Pentium III cadencé à 450MHz avec 128Mio de RAM), fonctionnant sous Linux.

Comme mesures *avec* le pool de route, nous avons considéré le temps entre l'instant où un processus `smartprovider` demande la connexion à Bob (ici en dehors du réseau pair-à-pair) via la route pré-établie qui vient du pool et le moment où celle-ci passe à l'état `ARS_READY` qui signifie que la communication est établie (dans ce cas il s'agit de communication à une seule route).

Pour obtenir ce temps, nous avons ajouté quelques lignes de code dans le module `smartprovider` afin qu'il mesure celui-ci et qu'il l'enregistre dans un fichier avant de se terminer.

Pour les mesures *sans* le pool de routes, nous avons considéré le temps que met justement le processus qui gère ce pool pour établir ces routes (le fait que les routes soient établies par ce processus et non par celui qui les utilisera ne change rien au temps qu'il faut pour les créer)[7]. Le processus `poolmanager` a donc été également modifié pour mesurer le temps entre la décision de créer une route et le moment où celle-ci est considérée comme étant prête à être passée à un autre processus qui en ferait la demande.

Les graphes des figures 7.3 et 7.4 représentent les résultats de ces tests.

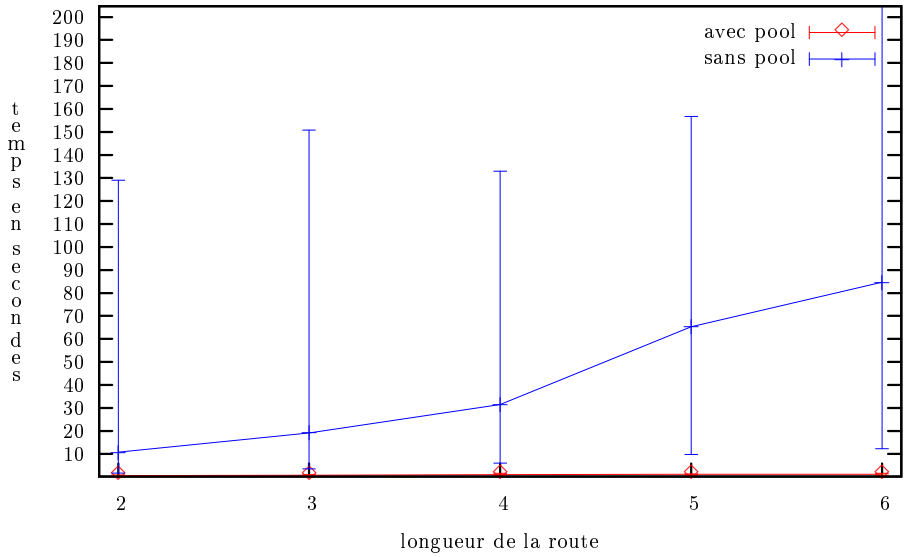


FIG. 7.3 – Temps d'établissement moyen (en secondes) des routes en fonction de leur longueur, avec et sans le pool de routes. Les barres verticales indiquent les temps minimun et maximum de chacun de nos échantillons. *N.B. :* la figure 7.4 présente la courbe *avec pool* uniquement, pour plus de lisibilité.

On observe approximativement deux droites, dont le coefficient directeur diffère.

Bien que dans les deux cas le temps d'établissement augmente linéairement avec le nombre de nœuds sur la route, l'intérêt du pool de routes devient évident : le temps d'attente pour l'utilisateur est beaucoup plus faible. En effet, avec le

[7] En toute rigueur, pour avoir réellement le temps d'établissement *sans* le pool de routes, il faudrait en plus y ajouter le temps pour se connecter à Bob, en dehors du réseau ; dans la pratique, en l'état actuel des choses, ce dernier est négligeable par rapport au temps d'établissement de la route elle-même.

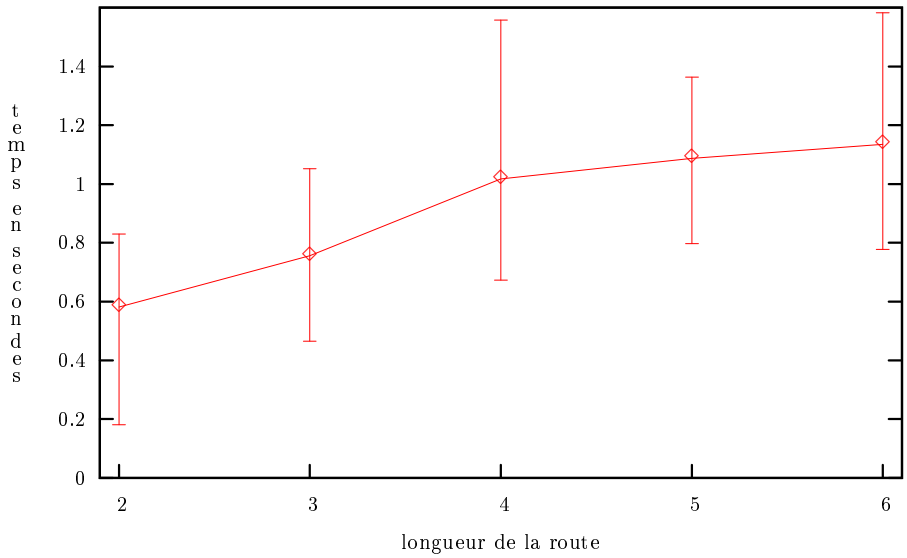


FIG. 7.4 – Temps d'établissement moyen (en secondes) des routes en fonction de leur longueur, avec le pool de routes. Les barres verticales indiquent les temps minimun et maximum de chacun de nos échantillons.

pool de routes, il n'y a plus qu'à établir la dernière connexion de la route, et non son intégralité. Si le temps d'établissement de cette dernière connexion augmente avec le nombre de nœuds car la latence sur la partie déjà établie augmente, le temps gagné est tout de même important.

Notons que sur ces tests, l'attente pour l'établissement de la connexion suivante au niveau de chaque nœud n'était pas activée. S'il l'avait été, les temps *sans* le pool de routes auraient été supérieurs (les temps *avec* auraient été similaires). Il faut en revanche noter que le protocole de retransmission en cas de perte de paquets pourrait être amélioré, ce qui devrait diminuer les temps d'établissement des routes, en particulier quand le nombre de nœuds est élevé.

Notons également que si le temps d'établissement (même avec le pool de routes si la longueur est grande) peut devenir gênant, il est a priori possible d'établir la communication avec une route courte, et d'y ajouter ensuite des routes plus longues si la communication se prolonge pour éviter les attaques par corrélation globale.

7.2.3 Performances des routes

La donnée essentielle en termes de performances des routes est la latence.

En effet, si le débit est limité par le lissage de trafic, il n'y a pas de raison qu'avec des tailles de fenêtres convenables par rapport à la latence on n'obtienne pas un débit moyen proche de la limite théorique (sur un flux d'une durée suffisamment longue pour que les temps de retransmission ne soient pas prépondérants).

Il est toutefois probable que le débit baisse légèrement quand la longueur de la route augmente, si l'on suppose que les pertes de paquets sont propor-

tionnelles à celle-ci. Des tests permettraient de quantifier cela et d'ajuster les mécanismes de retransmission.

La latence est, en théorie, constituée de plusieurs composants qui s'ajoutent :
- les temps de transmission entre deux nœuds ;
- les temps de passage sur chaque nœud, eux-mêmes décomposables en :
  - le temps de traitement proprement dit,
  - le temps d'attente (lissage de trafic), en moyenne égal à la moitié de la durée fixée entre deux paquets.

  Dans la pratique, nous avons noté sur nos machines que le temps de traitement est très petit (moins de 10 ms) devant le temps d'attente (lors de nos tests, la période d'envoi était fixée à 100 ms, le temps d'attente était donc de 50 ms en moyenne).

Sur notre réseau de tests, la latence est bonne : le programme `ping` donne des latences entre les nœuds universitaires comprises entre 20 et 30 ms, tandis que la latence entre le nœud situé sur le réseau de Free.fr et les autres est comprise entre 50 et 70 ms (maximum relevé, 71 ms entre ce nœud et celui situé à l'Université de Berlin).

Sur une route de longueur trois (avec donc six passages dans des nœuds), nous pourrions donc nous attendre à avoir une latence d'environ

$$\underbrace{40\,\text{ms} \times 3}_{\text{réseau}} + \underbrace{60\,\text{ms} \times 6}_{\text{noeuds}} = 480\,\text{ms}.$$

Comme le montre la figure 7.5, nos mesures donnent des résultats compatibles, voire inférieurs (ce qui peut s'expliquer par le recouvrement partiel des temps de traitement et de transmission et des temps d'attente sur les nœuds).

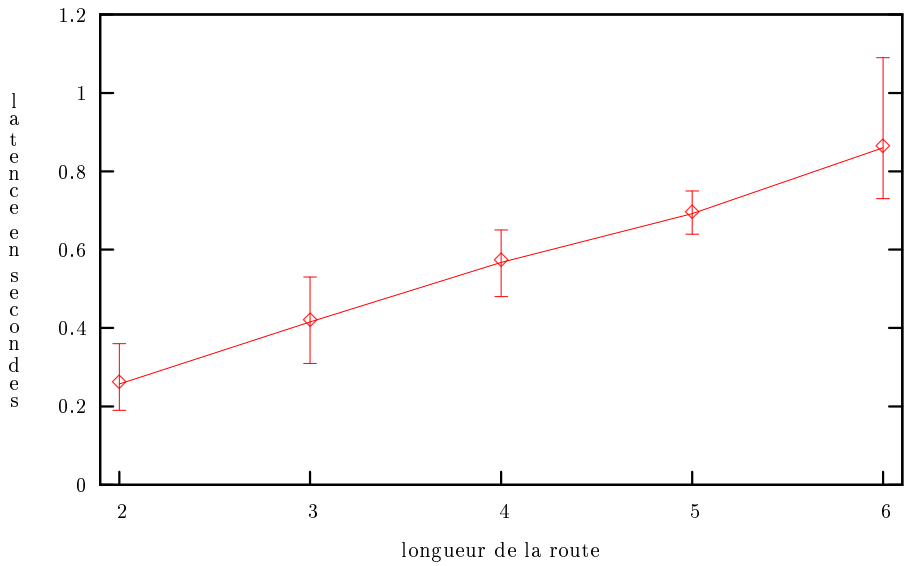


FIG. 7.5 – Latence moyenne d'une route en fonction de sa longueur, avec à chaque fois les valeurs minimale et maximale de l'échantillon.

Ces mesures ont été effectuées en ajoutant des types de messages (cf. A.3) `PING` et `PONG` : un nœud qui reçoit un `PING` renvoie un `PONG` avec le même nonce. Le nœud qui reçoit ce dernier message mesure le temps écoulé depuis qu'il a envoyé le `PING` correspondant.

Il faut toutefois noter que sur une même connexion, la latence peut varier de manière importante d'un paquet à l'autre (gigue), comme le montre l'exemple de la figure 7.6. Cela peut avoir des conséquences néfastes pour certaines applications (VoIP par exemple).

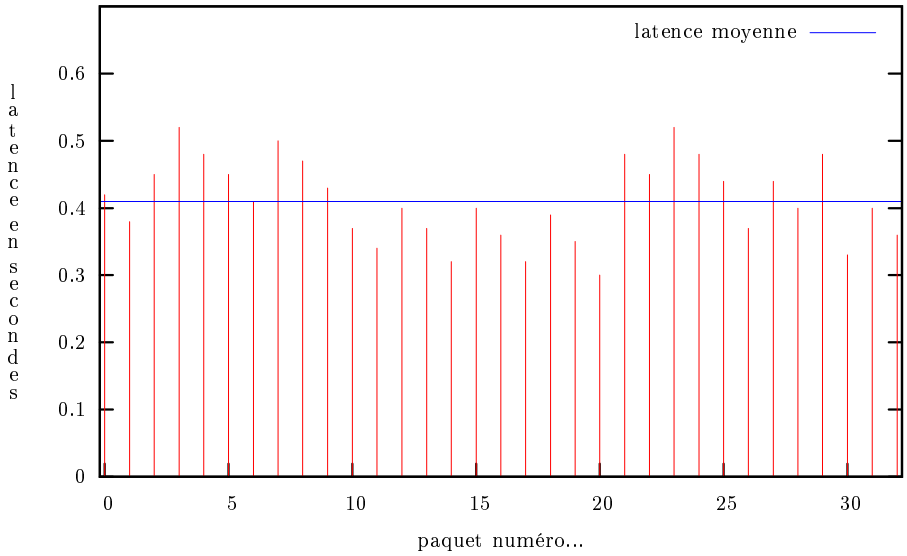


FIG. 7.6 – Exemple de variation de latence des paquets sur une route de longueur 3. Latence minimale sur l'échantillon : 0.30s ; latence maximale : 0.52s ; moyenne : 0.41s.

7.2.4 Conclusion sur les tests

Bien que nous n'ayons pas eu l'occasion d'approfondir nos tests autant que nous l'aurions souhaité pour des questions de temps, ceux que nous avons effectués et présentés dans les paragraphes qui précèdent sont très encourageants.

La latence en particulier est acceptable ; il faut d'ailleurs noter qu'elle pourrait être encore réduite en changeant les paramètres de lissage de trafic, au prix toutefois d'un *overhead* plus important (cf. 4.2).

Il serait intéressant de faire des tests de latence sur des communications utilisant de multiples routes, ce que nous n'avons pu faire pour le moment, les paquets `PING` et `PONG` étant gérés par les sessions de contrôle (qui sont spécifiques à chaque route) en non par les sessions de données (partagées entre les routes d'une communication). Si les diverses routes de la communication ont des latences moyennes comparables, il est notamment possible que cela réduise la gigue que nous avons constatée (cf. figure 7.6).

Le pool de routes remplit son office : le temps pour établir une communication est beaucoup plus faible que si l'on devait bâtir toute la route immédiatement. Il faut toutefois prendre garde à ne pas vider celui-ci plus vite qu'il ne se remplit ; les clients devront y veiller. Notons qu'il est possible de faire passer ce qui aurait normalement été plusieurs connexions TCP successives vers un même serveur au-dessus d'une même communication : cela permet d'économiser les routes du pool.

Il faudrait à présent faire des tests de débit, par exemple en modifiant un programme comme *NetPerf*[8] pour qu'il fonctionne au-dessus de *True Nyms*. L'utilisation de *NetPerf* permettra également de mesurer la latence sur des communications à routes multiples. Des mesures de débit précises permettraient d'améliorer la retransmission des paquets perdus.

Bien que nous n'ayons pas fait de tests de charge à proprement parler, nous pouvons néanmoins constater qu'elle semble raisonnable. Par exemple, sur nos petites machines de tests avec deux communications dans le pool et dix routes passant par le nœud, le programme `top` indique que le processeur est inactif environ 75% du temps.

[8] `http://www.netperf.org/`

Il est essentiel de laisser un espace privé à l'individu, que même l'État ne peut pas transgresser.

— Lucien Thiel (*Paperjam*, Novembre 2007)

8

Usage(s), conclusion et problèmes ouverts

Nous avons présenté dans les chapitres qui précèdent une solution générique qui permet d'obtenir des communications non-observables, *True Nyms*. Au delà de la non-observabilité, ce système peut également avoir d'autres usages proches ; la section 8.1 qui suit en donne quelques exemples.

True Nyms se base sur l'Onion-Routing, et protège celui-ci des attaques par analyse de trafic en utilisant divers mécanismes. Nous avons entre autres montré que l'utilisation de ces derniers n'est pas forcément rédhibitoire en termes de performances, si on les associe à d'autres mesures pour « protéger » celles-ci. Il serait probablement possible d'améliorer encore les performances avec une implémentation en espace noyau ou en choisissant plus finement divers paramètres ; nous revenons sur ce point ci-dessous, en 8.2.1.

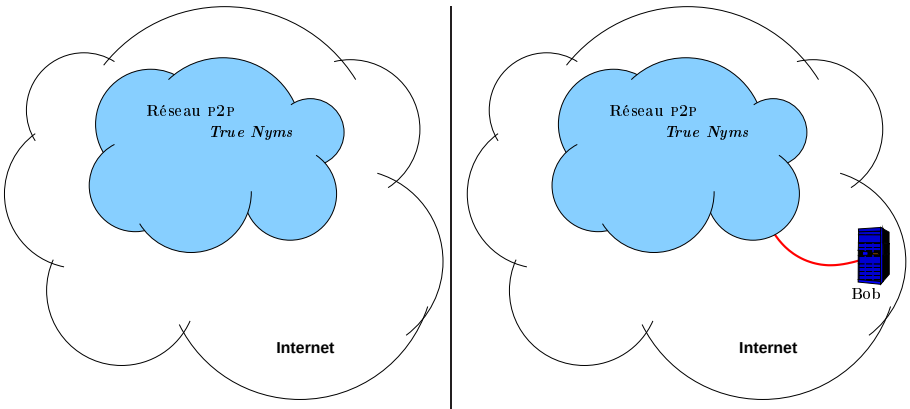
Par ailleurs, en dépit des avancées présentées précédemment, il reste encore des problèmes ouverts, comme le montre la section 8.2.2 ; y apporter une réponse permettrait d'améliorer encore les performances, mais aussi d'étendre l'usage de *True Nyms* à d'autres cas, par exemple celui où l'on veut avoir un *serveur* non-observable (*i.e.*, un attaquant ne peut savoir qu'une machine a un rôle de serveur).

8.1 Usage(s) de *True Nyms*

L'usage principal de *True Nyms*, c'est-à-dire le but dans lequel il est conçu, est la non-observabilité, c'est-à-dire permettre à Alice de communiquer avec Bob sans qu'un observateur (qui n'est ni Alice ni Bob) ne le sache.

Si l'usage de *True Nyms* par Alice est nécessaire pour atteindre ce but, il faut distinguer deux cas : en effet, il n'est pas forcément nécessaire que Bob, lui, utilise *True Nyms* (figure 8.1).

Il est également possible d'utiliser *True Nyms* pour d'autres objectifs. La



Bob est *dans* le réseau de pairs *True Nyms*. Un observateur extérieur ignore tout des communications entre Alice et Bob. Au plus peut-il savoir qu'ils font tous deux partie du réseau (ce qui ne lui apporte pas d'information si le réseau est suffisamment grand).

Bob est *en dehors* du réseau de pairs *True Nyms*. Un observateur voit qu'il y a une connexion entre un utilisateur (il ignore lequel) du réseau *True Nyms* et Bob. Il peut éventuellement tenter d'attaquer la communication (en rouge) entre Bob et le réseau *True Nyms* pour obtenir des informations sur la personne qui est à l'origine de cette communication.

FIG. 8.1 – Bob peut être ou non dans le réseau de pairs *True Nyms*.

non-observabilité est une sorte de super-anonymat, et donc l'anonymat est également possible.

Il faut néanmoins prendre garde au fait que dans ce cas, c'est avant tout de Bob que l'on veut se protéger. Alice doit donc faire très attention à ce que les données transmises au-dessus de *True Nyms* ne la trahissent pas : le cas serait alors semblable à celui de la « lettre anonyme signée » illustré par la figure 8.2.

On peut imaginer dans ce but l'adjonction de *proxies* filtrants afin de « récurer » les protocoles des couches supérieures de toute information permettant l'identification. Avec la plupart des protocoles, la tâche est difficile et, dans le cas général, le problème est sans doute indécidable. La difficulté variera aussi selon qu'Alice désire de l'*anonymat* au sens strict, ou du *pseudonymat*. Notons que ces couches supérieures peuvent également laisser « fuir » de l'information dans le cas où le but recherché est la non-observabilité.

On pourrait également vouloir employer *True Nyms* dans un but de lutte contre la censure. Il ne s'agirait pas ici d'héberger de manière non-censurable des sites ou des données, comme le permet par exemple FreeNet [20], mais d'accéder à des sites ordinaires bloqués par les autorités locales.

L'efficacité est à envisager au cas par cas, selon la méthode de censure utilisée pour bloquer l'accès au site et le contexte politique.

Sur le plan politique, il faut noter que l'usage de *True Nyms* n'est pas discret, en particulier à cause du lissage de trafic (voir 4.5). Un gouvernement autoritaire qui veut contrôler l'usage d'Internet peut donc très bien interdire l'usage de *True Nyms* et détecter les contrevenants.

Planches extraites de *Crie, ô, Génie* (Léonard, tome 15) par Turk et De Groot (Dargaud, 1987).

FIG. 8.2 – Pour l'anonymat, la structure du message ne doit pas permettre d'identifier l'émetteur, mais son contenu non plus ! *True Nyms* n'offre aucune garantie sur ce dernier.

Sur le plan technique, si le filtrage est implémenté au niveau IP, *True Nyms* peut être employé pour le contourner. S'il est implémenté au niveau du cache DNS du fournisseur d'accès (comme c'est souvent le cas dans les pays européens), c'est alors une question de choix de serveur DNS, indépendante de *True Nyms* [1] ; ce que nous décrivons dans la partie suivante pourrait par contre être utile !

8.2 Améliorations possibles et problèmes ouverts

Nous qualifions *True Nyms* de solution générique. Cette qualification de « solution générique » indique-t-elle que nous avons résolu tous les problèmes et que l'on va pouvoir clore toutes les recherches dans ce domaine, pour cause de « plus rien à trouver » ? Non, bien sûr ! Et loin s'en faut !

Dans cette section, nous présentons d'abord quelques idées susceptibles d'améliorer les performances, du moins dans certains cas, puis nous indiquons quelques problèmes plus complexes, auxquels il serait intéressant d'apporter une solution.

8.2.1 Quelques idées pour améliorer les performances

Il faut garder à l'esprit que si notre but était montrer la viabilité de nos idées en termes de performances, celles-ci pourraient sans doute être améliorées, de

[1] *True Nyms* permettra toutefois à Alice d'accéder à un autre cache DNS que celui de son fournisseur d'accès.

manière « simple »[2].

Au niveau de l'implémentation, nous avons par exemple privilégié la sécurité et la facilité de développement plutôt que les performances. Celles-ci pourraient probablement être améliorées, par exemple en implémentant toute la partie « gestion d'une route établie » au niveau du noyau du système d'exploitation et en ne gardant en mode utilisateur que les parties qui concernent la gestion du réseau pair-à-pair et l'établissement des routes, un peu comme cela se fait généralement pour IPsec. Cela éviterait les goulots d'étranglement que sont le passage en mode noyau pour bénéficier du chiffrement matériel (de plus en plus de processeurs disposent de telles fonctions) et le passage de la structure « communication » d'un processus à l'autre via une socket Unix : les communications resteraient alors dans le noyau et pourraient être utilisées via un descripteur.

Le protocole pourrait sans doute également être modifié ou étendu pour obtenir de meilleures performances :

- d'une part, les paramètres choisis (par exemple la taille et la fréquence des paquets pour le lissage de trafic) l'ont été de manière heuristique, notre expérience nous indiquant qu'ils seraient acceptables à la fois en termes de performances des communications et d'encombrement du réseau. Il s'agit donc plus de « l'*art de l'ingénieur* » que d'une approche formelle et nous ne pensons pas que ces paramètres soient optimaux. Il est peut-être possible d'améliorer les performances en jouant dessus. Il faut toutefois prendre garde à ne pas privilégier une application ou un type de réseau particulier ;
- d'autre part, avec certaines applications du moins, une part importante des paquets est constituée de padding, ou même certains paquets envoyés par l'une des extrémités sont des leurres (ils ne contiennent aucune donnée). Il est donc légitime de se demander s'il n'y aurait pas un moyen d'améliorer les choses de ce côté aussi. Nous voyons plusieurs pistes possibles :
  - multiplexer les communications au-dessus des routes non-observables. Ce n'est probablement pas possible pour tous les protocoles, mais par exemple, pour l'accès à un site Web, deux connexions vers deux fichiers liés à la même page pourraient sans doute passer par les mêmes routes. Il y a plusieurs options possibles, selon qu'un même paquet peut contenir des données correspondant à plusieurs connexions ou pas.
    Si sur le plan de la sécurité ce n'est probablement pas une bonne idée de multiplexer aveuglément plusieurs communications, il serait intéressant qu'une implémentation en fournisse la possibilité au logiciel client, en laissant celui-ci libre de l'utiliser ou non,
  - remplacer le padding des paquets par des codes correcteurs, l'idée étant que si un paquet est perdu, il pourrait ainsi être reconstitué à partir des données contenues dans le ou les paquets qui le suivent. Cela permettrait d'éviter de signaler une erreur et d'attendre la retransmission, ce qui est forcément long à cause de la latence de la route ;
- à l'inverse, avec d'autres applications (transferts de fichiers par exemple), tous les paquets sont remplis et *True Nyms* devient le goulot d'étranglement du débit.
  Actuellement, nous ne compressons pas les données avant de les transmettre. Le faire de manière aveugle ne serait pas une bonne idée : pour

[2] Le qualificatif simple dénote ici le fait qu'il n'y a pas de problème difficile à résoudre. Les choses ne sont bien sûr pas si simples que cela. Il ne faut d'ailleurs pas s'attendre à des miracles avec ces propositions.

les protocoles qui envoient peu de données et où le débit n'est pas limité par notre système, cela ne ferait qu'augmenter encore la latence.
Par contre, il serait possible de compresser les données de manière adaptative : si l'on a beaucoup de données à envoyer dans la file d'attente, c'est que les routes sont le facteur limitant et on les compresse beaucoup avant de les envoyer ; si au contraire les files d'attente sont vides on les compressera moins, ou même pas du tout (voir à ce sujet, en dehors du cadre de l'anonymat et de la non-observabilité, les travaux de Jeannot [41, 40]).

8.2.2 Quelques problèmes (toujours) ouverts

Comme nous l'avions annoncé, nous présentons ici quelques problèmes non résolus ; leur apporter une réponse permettrait sans doute d'améliorer *True Nyms*. Cette liste n'est en aucune manière exhaustive.

Prise en compte de la topologie du réseau

Les routes sont actuellement créées par les nœuds de manière aléatoire. Cela implique que pour se connecter à un ordinateur proche, une communication utilisant *True Nyms* fera peut-être plusieurs fois le tour de la Terre, ce qui se ressent sur la latence. Il n'est bien sûr pas possible de se connecter directement, sinon la non-observabilité serait perdue. La question est donc : « est-il possible de prendre en compte la topologie du réseau pour avoir des communications avec une plus faible latence, sans perdre les garanties d'anonymat ? »

Si l'on connaît réellement la topologie du réseau, cela devrait être possible, à condition toutefois d'introduire assez d'aléa pour qu'un observateur ne puisse obtenir d'information sur l'origine à l'aide de la « direction » d'où semble provenir la connexion. L'un des problèmes principaux est que la topologie du réseau, ou du moins les performances entre une paire de nœuds, serait rapportée par les nœuds eux-mêmes, or l'on ne peut pas faire confiance à ceux-ci. Il serait certes possible de détecter un nœud malintentionné, mais ce serait beaucoup plus difficile d'en détecter une coalition. Cette question reste donc, à l'heure actuelle, ouverte.

Notons que Tor (cf. 1.3.1) a tenté d'utiliser une telle solution ; cela a conduit à des problèmes de sécurité, pour les raisons que nous venons de mentionner [6].

Points de rendez-vous non-observables

Les points de rendez-vous permettent d'avoir des communications où à la fois Alice et Bob sont anonymes / non-observables, leurs communications aboutissant sur un nœud tiers qui les met en relation. Le problème n'est pas lié directement à ce nœud tiers, mais à la manière dont Alice et Bob fixent un rendez-vous.

Nous l'avons vu, Tor (cf. 1.3.1) utilise pour ce faire un mécanisme de table de hachage distribuée. Si ce mécanisme est élégant et résout bien le problème dans un contexte d'anonymat, il reste malheureusement observable par un observateur qui a une vue générale du réseau.

Une solution à ce problème pourrait être qu'Alice envoie son message indiquant le point de rendez-vous à Bob, chiffré avec la clef publique de Bob, à l'ensemble du réseau ; ainsi, seul Bob pourrait lire le message et un observateur

ne saurait pas, lui, qui est le destinataire, puisque tout le monde l'a reçu. Il y aurait malheureusement alors un gros problème de tenue en charge, sans même parler des dénis de service. Là aussi, la question reste pour le moment ouverte.

Des solutions pour les protocoles au-dessus d'UDP

True Nyms, tel que décrit dans les chapitres précédents, fonctionne avec UDP. Cela ne veut malheureusement pas dire qu'il est utilisable avec tous les protocoles qui utilisent UDP.

Il y a principalement deux cas à problèmes :

- les protocoles qui ont besoin d'une très faible latence ; un exemple qui vient à l'esprit est celui de la téléphonie (VoIP) : une latence de plus de quelques centaines de millisecondes est gênante. Les protocoles de synchronisation des horloges, comme NTP, se basent sur la latence (et sa variation, la gigue) : il est peu probable qu'utilisé au-dessus de *True Nyms* NTP soit très précis ! Il y a d'autres cas moins critiques ; par exemple DNS. Dans ce cas, cela a néanmoins un impact sur d'autres protocoles — y compris des protocoles au-dessus de TCP. DNS est suffisamment répandu et nécessaire pour que nous proposions une solution spécifique, c'est l'objet de la seconde partie ;
- les protocoles sans états, où le serveur répond immédiatement aux requètes, par exemple, une fois encore, NTP ou DNS. Le problème est que les serveurs sont souvent sollicités par un très grand nombre de clients ; utiliser *True Nyms* (en les intégrant dans le réseau) les forcerait à lier un état à chaque requète. Cela n'est guère possible, pour des questions de ressources, tant en calcul (cryptographie à clef publique pour l'établissement de session) qu'en mémoire (stockage des informations de session jusqu'à l'envoi de la réponse).

En dépit de ce sombre tableau, il existe néanmoins des protocoles au-dessus d'UDP qui fonctionnent correctement au-dessus de *True Nyms* ! Par exemple ceux de *streaming* audio ou vidéo. Notons également que le problème en ce qui concerne les protocoles sans états est avant tout du côté du serveur. Ces protocoles sont généralement utilisables si le serveur reste en dehors du réseau de nœuds de *True Nyms*, et que seul le client l'utilise. Le surcoût pour le client est néanmoins très important (création / utilisation d'une route pour envoyer un unique paquet !).

Dans la partie suivante...

Ceci conclut notre première partie, consacrée aux méthodes pour obtenir des communications à faible latence avec des performances acceptables. En dépit des limites que nous venons d'évoquer, notre solution fonctionne parfaitement.

Dans la partie qui suit, nous nous intéressons cette fois au cas de DNS, en particulier au niveau du client ; c'est un protocole important, mais dont les caractéristiques font que son usage avec *True Nyms* dégrade les performances d'autres services qui l'utilisent. Nous allons le voir, il est là aussi possible d'améliorer les choses.

Deuxième partie

Vie privée dans le *Domain Name System*

9

Le fonctionnement du DNS

Le *Domain Name System* (DNS) est peu connu. C'est pourtant l'une des briques de base de l'Internet, et l'on peut affirmer sans grand risque d'erreur que tous les internautes l'utilisent.

Cependant, bien qu'il repose au sommet de la hiérarchie dans les modèles de réseau en couches (OSI, TCP/IP), au même titre que le Web par exemple, il n'est pas aussi visible que celui-ci, car il n'est a priori pas employé directement par les utilisateurs. Généralement, seuls les administrateurs l'utilisent explicitement.

En effet, le DNS sert principalement à *résoudre les noms de domaines*, c'est-à-dire à transformer un nom qu'un être humain peut retenir en une adresse IP utilisable par un ordinateur.

DNS est un protocole ancien, son origine est à peu près concomitante à celle d'IP, au début des années 1980. Si des extensions et des modifications ont été proposées depuis, il reste très semblable à ce qu'il était à ses débuts. Parmi les extensions existantes, on citera en particulier DNSsec, créée pour répondre à des inquiétudes en termes de sécurité et sur laquelle nous reviendrons. Cependant, il reste un problème lié à la sécurité : celui de la vie privée.

Avant de revenir plus longuement sur ce problème dans le chapitre suivant puis de proposer une (des ?) solution(s) dans le reste de cette partie, nous allons dans ce chapitre examiner plus avant le fonctionnement de ce protocole.

9.1 DNS : une base de données distribuée et hiérarchisée

Le *Domain Name System* [13, 53, 54] est une base de données distribuée, répartie sur une hiérarchie de serveurs et utilisant un ensemble de caches pour obtenir de meilleures performances et une bonne tenue en charge.

9.1.1 Une architecture arborescente

En haut de la pyramide se trouvent les *serveurs racines*. Leur rôle est d'indiquer le serveur (ou, généralement, les serveurs) qui connaît les informations relatives à un *top level domain* (ou *tld*), par exemple le `.fr` de `www-id.imag.fr` ou le `.lu` de `mail.uni.lu`.

Ces serveurs, juste au-dessous des serveurs racines dans la hiérarchie, sont donc appelés des *serveurs de tld*. Ils disposent des informations relatives aux domaines qui se trouvent sous le tld en question. Par exemple, le serveur de tld `.fr` saura à quel serveur demander les informations concernant le domaine `imag.fr`.

Enfin (en général), on arrive aux serveurs DNS d'une organisation spécifique, par exemple d'une université ou d'un fournisseur d'accès. Ce serveur connaît, lui, les adresses des machines de l'organisation en question, comme `www-id.imag.fr` ou `mail.uni.lu` (mais il peut éventuellement encore y avoir plusieurs niveaux de sous-zones).

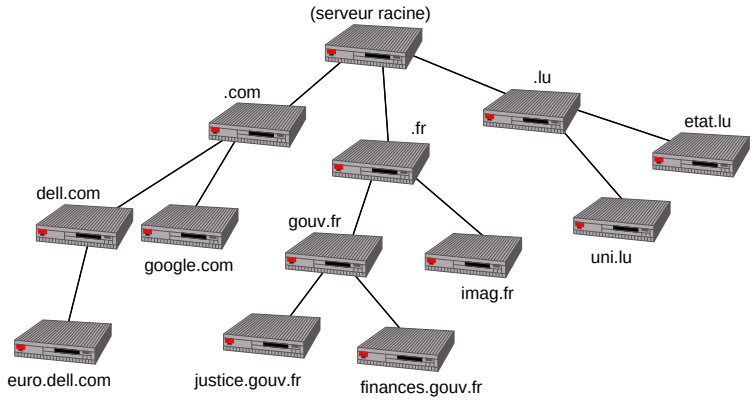


FIG. 9.1 – Les serveurs DNS forment une hierarchie arborescente.

9.1.2 Principe de la résolution d'un nom

Quand un ordinateur veut résoudre une adresse, il parcourt l'arbre des serveurs en commençant par la racine.

Suivons par exemple ce qui se produit si l'on veut résoudre `mail.uni.lu`. L'ordinateur commence par demander à un des serveurs racines « Qui est `mail.uni.lu` ». Le serveur racine lui répond avec quelque chose comme « pour tout ce qui concerne `*.lu`, demandez à 192.36.125.2 »[1], ce que fait l'ordinateur. 192.36.125.2 peut à son tour répondre « pour tout ce qui concerne `*.uni.lu`, demandez à 158.64.1.23 », puis finalement 158.64.1.23, quand il est interrogé, répond « `mail.uni.lu` à l'adresse 158.64.76.51 », comme illustré dans la Figure 9.2. Le logiciel client qui fait cette résolution de nom est appelé *solveur*.

[1] On dit que 192.36.125.2 *fait autorité* sur le domaine `.lu`.

Client	⟶	Serveur racine	: {Qui est `mail.uni.lu` ?}
Serveur racine	⟶	Client	: {Pour `*.lu`, demandez à 192.36.125.2.}
Client	⟶	192.36.125.2	: {Qui est `mail.uni.lu` ?}
192.36.125.2	⟶	Client	: {Pour `*.uni.lu`, demandez à 158.64.1.23.}
Client	⟶	158.64.1.23	: {Qui est `mail.uni.lu` ?}
158.64.1.23	⟶	Client	: {`mail.uni.lu` a l'adresse 158.64.76.51.}

FIG. 9.2 – Exemple de résolution d'un nom.

9.2 La manière dont DNS est déployé

Bien que la section précédente décrive le principe de base du système, il faut encore y ajouter un mécanisme essentiel pour obtenir quelque chose qui ressemble au déploiement réel du DNS : un réseau de caches.

9.2.1 Un réseau de caches

Les caches DNS se comportent vis-à-vis du client comme un serveur, à la différence que s'ils ne connaissent pas la réponse, ils font eux-mêmes la résolution puis ils stockent la réponse. Ainsi, si un utilisateur (le même ou un autre) envoie une requête similaire avant que la réponse expire[2], celle-ci sera connue et pourra être directement renvoyée.

Les caches DNS ont deux effets :

- ils réduisent la charge sur les serveurs qui forment la base de données (ceux du haut de la hiérarchie seraient sinon très sollicités) ;
- ils réduisent le temps de réponse moyen pour l'utilisateur.

Il existe souvent plusieurs niveaux de caches : les niveaux les plus bas de la hiérarchie ne pratiquent pas la résolution si la réponse n'est pas connue et se contentent alors de transmettre la requête au niveau supérieur.

Il faut noter que la cohérence entre la copie et la donnée originale n'est pas garantie ; lorsque les données changent sur le serveur, les caches n'auront la mise à jour qu'à la première requête pour cette donnée suivant l'expiration de l'ancienne. Un cache peut donc servir l'ancienne donnée de manière « erronée » pendant un temps qui peut aller jusqu'à la durée de vie indiquée par le serveur (cas où le cache avait demandé la donnée juste avant la modification). Dans la pratique, c'est rarement un problème.

9.2.2 La résolution de nom dans la pratique

Dans la pratique, la résolution de nom n'est donc pas faite par les machines clientes elles-mêmes (voir aussi la figure 11.1. Elles enverront les requêtes à un « serveur DNS » (en fait un cache) qui leur donnera la réponse, soit parce qu'il la connaît déjà, soit parce qu'il fait la résolution.

Notons finalement que sur les implémentations courantes, le mécanisme de résolution de nom est intégré dans les bibliothèques (en général la bibliothèque C) en espace utilisateur et non dans le noyau du système d'exploitation.

Cela implique notamment qu'il n'est guère judicieux d'intégrer un cache à ce niveau, puisque les données ne seraient ni partagées entre les applications ni

[2] La durée de vie est fournie dans le message, comme nous le verrons dans la section 9.3.

conservées entre deux appels à une même application (sauf à changer profondément l'esprit des bibliothèques standard). Certains langages de plus haut niveau le font toutefois (par exemple Java tel qu'implémenté par la JVM de Sun), de même que certaines applications (*e.g.*, Firefox où la durée maximale de conservation des données est paramétrable via l'option `network.dnsCacheExpiration`).

9.3 Un aperçu du protocole

Le protocole fonctionne par envoi de messages. Chaque message est constitué d'un en-tête, de questions, de réponses, d'« autorités » et d'informations supplémentaires. L'en-tête indique le nombre de chacun des autres types d'information et le format du message est illustré par la figure 9.3.

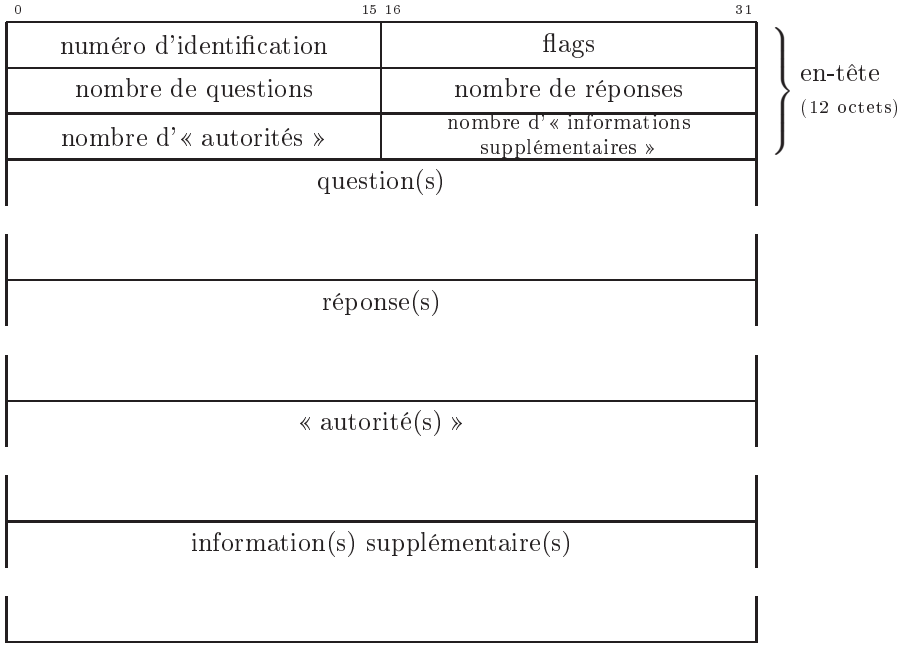


FIG. 9.3 – Le format d'un message DNS.

Le champ de *flags* est constitué ainsi :

Le sens de ces flags est le suivant :

QR indique si le message est une question (1) ou une réponse (0) ;

opcode spécifie le type de l'opération demandée (0 pour une requête standard) ;

AA vaut 1 si la réponse est envoyée par un serveur qui fait autorité sur le domaine en question ;

TC indique que la réponse a été tronquée (les messages sur UDP ne peuvent pas, dans la version d'origine, dépasser 512 octets, le client doit alors recommencer la requête en utilisant TCP) ;

RD veut dire « récursion désirée » : si le serveur ne fait pas autorité pour la question posée, doit-il indiquer à qui reposer la question (0) ou faire la résolution lui-même (1) ?

RA est utilisé par le serveur pour indiquer s'il accepte les requêtes récursives (cf. flag précédent) ;

rcode permet de renvoyer une erreur.

Notons que la zone noire est *réservée à de futurs usages (RFU)* dans la description originale et est normalement remplie de zéros ; dans la pratique ses possibilités ont été attribuées et épuisées depuis longtemps, et les nouvelles extensions, comme DNSsec sur laquelle nous reviendrons en 9.4.1, passent par un autre mécanisme (basé sur la définition d'un pseudo-RR `OPT` pouvant se trouver dans la zone *information(s) supplémentaire(s)* des requêtes et des réponses), défini dans la RFC 2671 [77], dont la description sort du cadre de ce document.

Les questions comprennent trois champs (figure 9.4) : le *nom* cherché, le *type* des informations demandées, et la *classe* de la requête, qui vaut 1 pour l'utilisation sur un réseau IP.

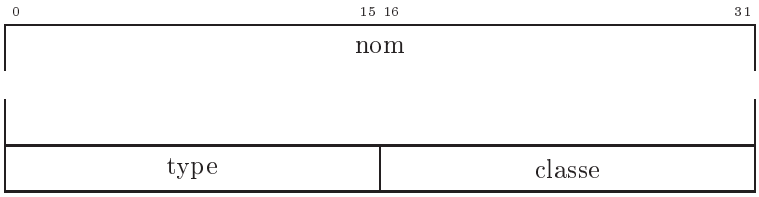


FIG. 9.4 – Le format des *questions* DNS.

Les principaux types sont `A` (adresse IPv4), `AAAA` (adresse IPv6), `CNAME` (nom *canonique* d'une machine), `MX` (machine qui gère les mails), `NS` (serveur qui fait autorité sur un domaine) et le type spécial `ANY` qui n'est valable que pour les requêtes.

Les « réponses », « autorités » et « informations supplémentaires » sont organisées en *resource records* (ou RR), qui commencent comme des questions et ajoutent des champs supplémentaires (figure 9.5). On notera la présence d'un champ *durée de vie* (ou TTL pour *Time to Live*) qui indique la validité des données : il permet aux caches de savoir combien de temps ils peuvent stocker les informations contenues dans le RR.

Une requête comporte en général une question, tandis que la réponse répète la question et y ajoute un ou plusieurs RR qui constituent la réponse proprement dite.

On remarquera que la taille des noms n'est pas précisée ; ceux-ci sont donc codés d'une manière qui permet de la déterminer, à savoir : les octets qui représentent normalement un point dans un nom de domaine sont remplacés par la taille du mot qui suit. Un octet qui indique la taille du premier mot est ajouté au début, ainsi qu'un octet nul à la fin d'un nom. Par exemple, `www.google.lu` sera codé en :

3	w	w	w	6	g	o	o	g	l	e	2	l	u	0

Notez que ce codage implique une borne maximale sur la longueur d'un nom : le caractère « a » a la valeur 64 en ascii.

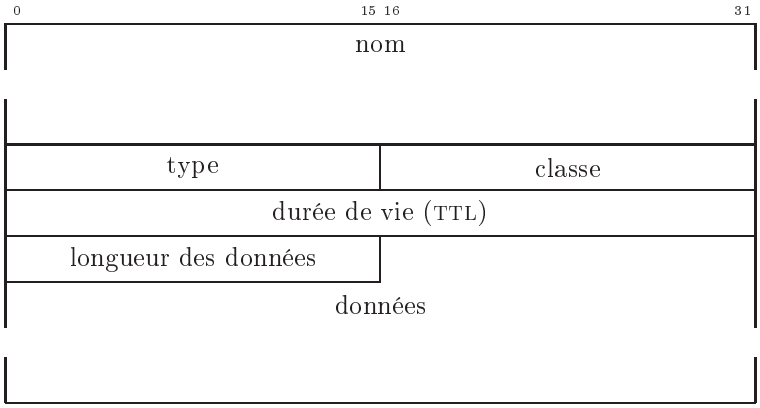


FIG. 9.5 – Le format des *resource records* DNS.

La zone *réponse(s)* n'est utilisée que si le serveur qui répond renvoie directement l'information demandée ; si par exemple, comme le serveur racine dans l'exemple de la figure 9.2, il veut indiquer qu'il faut interroger un autre serveur, il utilisera la partie *autorité(s)* en y mettant un RR de type `NS` pour indiquer le serveur à interroger pour une certaine zone.

L'exemple 9.2 est d'ailleurs simplifié : le serveur racine, dans ce cas indique par exemple dans la partie *autorité(s)* qu'il faut demander pour tout ce qui concerne `.lu` au serveur `A.DNS.lu` (RR de type `NS`), et ajoutera un RR de type `A` dans la partie *information(s) supplémentaire(s)* pour indiquer que l'adresse IP de `A.DNS.lu` est `204.74.112.251`.

Il faut enfin préciser qu'il y a encore divers détails à prendre en compte dans l'implémentation : par exemple pour réduire la taille des données, il est possible d'indiquer un bout de nom plus haut dans le message plutôt que de le répéter. Nous n'en dirons pas plus sur ces sujets, le lecteur intéressé par ces points pourra se reporter à [70] ou aux RFCs.

9.4 La sécurité dans le DNS

Comme pour les autres protocoles qui datent de cette époque, la sécurité n'est pas incluse à la base dans le DNS. Dans cette section, nous allons donner un aperçu des problèmes et des solutions qui existent.

9.4.1 Authentification et intégrité : *Spoofing* et *Cache poisonning*

La sécurité dans le DNS est avant tout un problème d'intégrité et d'authentification des données. Nous expliquons ici pourquoi celles-ci sont plus que lacunaires dans le DNS classique, et décrivons l'extension DNSsec conçue pour remédier à ce problème.

Authentification et intégrité dans le DNS classique

Elles sont faibles, puisqu'elles reposent avant tout sur le numéro d'identification de la requête. Si une réponse arrive avec l'IP du serveur auquel on a

fait la demande et un numéro d'identification correct, elle sera acceptée comme valide. Le protocole DNS étant basé sur UDP, l'usurpation de l'IP du serveur est aisée, et le numéro d'identification peut de plus en plus souvent être attaqué par force brute, ce champ ne faisant que 16 bits et le débit des réseaux étant élevé (figure 9.6).

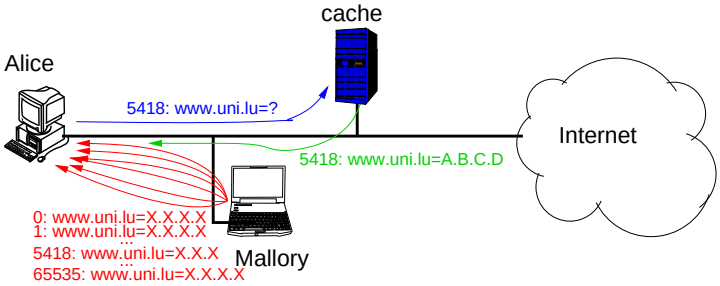


FIG. 9.6 – Si la réponse de Mallory avec le bon numéro d'identification arrive avant celle du cache, elle sera utilisée par Alice.

Si la requête émane d'un cache et non d'un simple client, la réponse fausse sera stockée dans le cache – jusqu'à ce qu'elle expire, mais l'attaquant peut utiliser la durée de vie maximale, soit en général une semaine[3] – et transmise à tout client du cache demandant cette information. On dit alors que le cache est empoisonné (*cache poisonning*).

Ce type d'attaque est de plus facilité par des implémentations qui génèrent des numéros d'identification d'une manière prévisible. La tâche n'est pas aisée et bien que ces attaques soient connues depuis des années, une nouvelle faille dans le générateur de nombres aléatoires de *bind* (un logiciel serveur DNS très répandu) a encore été publiée en juin 2007 [46].

Pour limiter cette vulnérabilité, certaines implémentations[4] utilisent l'astuce consistant à utiliser des numéros de ports clients aléatoires pour leurs requêtes : il faut alors que correspondent à la fois le numéro d'identification et le numéro de port (16 bits également, néanmoins certains ports ne peuvent être utilisés), ce qui augmente considérablement la difficulté de l'attaque (mais ne permet pas de l'exclure sur les réseaux actuels).

DNSsec

Pour remédier à ces problèmes, l'IETF a défini DNSsec [2, 4, 3], qui permet de signer les informations publiées.

L'idée de DNSsec est d'ajouter des RR spéciaux pour authentifier les RR habituels : le type `RRSIG` contient la signature numérique d'un autre RR.

Cela implique donc également d'ajouter des mécanismes de PKI pour distribuer les clefs qui servent pour la signature des informations : les RR `DNSKEY` contiennent les clefs publiques qui permettent de vérifier les signatures, tandis que les RR `DS` permettent de déléguer la gestion de parties de l'arborescence.

[3] Bien que le champ TTL ait une longueur de 32 bits, la plupart des caches fixent une borne supérieure à la conservation des données, en général une semaine.

[4] Addenda (2008-07-14) : depuis que nous avons écrit les lignes ci-dessus, Dan Kaminsky aurait découvert une manière d'exploiter facilement cette vulnérabilité. Il semble que l'utilisation de numéros de ports aléatoires soit en passe de devenir la règle.

Enfin, un RR `NSEC` a été introduit pour indiquer de manière sûre la *non*-existence de noms.

On notera que la validité des signatures étant précisée par des dates, DNSsec nécessite une certaine synchronisation des horloges des machines (alors que ce n'était pas nécessaire avec le DNS standard). Par ailleurs, les RR de DNSsec sont souvent de taille importante. Pour éviter de recourir trop souvent à TCP, une implémentation qui supporte DNSsec doit également implémenter la RFC 2671 [77] qui ajoute la possibilité d'avoir des messages au-dessus d'UDP de plus de 512 octets.

DNSsec tire aussi parti de la RFC 2671 pour ajouter des flags qui permettent aux implémentations de signaler le support de cette extension, l'incorporation de signatures, etc.

Bien que progressivement déployé, DNSsec reste actuellement peu utilisé.

9.4.2 Dénis de service

Un autre problème de sécurité vient du fait qu'une requête consiste juste à émettre un paquet UDP. Il n'y a en particulier aucune session à établir. Si cela permet une grande rapidité des interrogations, le client qui n'est pas intéressé par la réponse n'est pas obligé de garder des états en mémoire. Il peut donc générer des requêtes pseudo-aléatoires très rapidement, sans avoir besoin de grandes ressources. Un tel client est en général limité par sa bande passante.

La requête demande plus de ressources au serveur, puisqu'il doit interroger sa base de données pour y répondre[5]. Cela ouvre la porte à des attaques de type « déni de service ». Si un serveur dispose généralement d'une puissance de calcul et d'une bande passante supérieures à celles du client (en particulier pour les serveurs haut placés dans la hiérarchie), ils restent vulnérables face à des attaques distribuées (DDoS) lancées par un grand nombre de clients (par exemple les *zombies* d'un *bot-net*).

Il faut cependant noter que s'il y avait établissement de session, il faudrait s'arranger pour le faire d'une manière qui ne nécessite pas que le serveur garde des états en mémoire, sous peine d'aggraver le risque de dénis de services au lieu d'y remédier.

Le 22 octobre 2002 (le 21 dans certaines zones horaires), neuf des treize serveurs DNS racines sont tombés face à un déni de service distribué. D'après [78], le trafic n'était lors de cette attaque pas constitué de requêtes DNS, mais de paquets divers dont le but était de saturer les liens réseaux vers ces serveurs et non les serveurs eux-mêmes.

En dépit de cette attaque et d'autres ayant eu un moindre impact, il faut constater que le DNS résiste bien aux dénis de service.

Les serveurs DNS racines restent néanmoins un point critique, puisque leur chute bloquerait progressivement l'ensemble du DNS (et de l'Internet), à mesure de l'expiration des données dans les caches. Des propositions ont été faites pour remplacer le système hiérarchique par un réseau pair-à-pair [24, 15] et éliminer ainsi ce problème.

[5] Dans le cas des serveurs racines, cette base de données est très petite : la zone racine, en format ascii fait (au 23 août 2007) moins de 70 Kio. Elle est disponible à l'adresse `http://www.internic.net/zones/`. La recherche est donc très rapide. En revanche, dans le cas du serveur d'un gros tld, la base de données est beaucoup plus grosse, jusqu'à plusieurs Gio ! La recherche est donc plus longue.

9.4.3 Confidentialité

Appliquée au DNS, la confidentialité peut avoir deux sens :

1. la *confidentialité des données* stockées sur un serveur, *i.e.*, empêcher que n'importe qui y ait accès ;
2. la *confidentialité des échanges* entre les serveurs ou entre un serveur et un client.

Confidentialité des données

Il n'y a à notre connaissance pas de moyens spécifiques pour obtenir la confidentialités des données. L'identification et l'authentification du client par le serveur sont difficiles car le protocole sous-jacent est UDP.

On peut atteindre avec plus ou moins de succès cette forme de confidentialité en déployant des serveurs séparés et des méthodes de segmentation de réseau (firewalls) et / ou d'authentification (par exemple IPsec) d'une manière séparée du DNS. On parle de *split zone*.

De telles méthodes sont par exemple utilisées par les ISP pour indiquer des serveurs de mails différents selon que l'on se trouve sur leur réseau ou à l'extérieur de celui-ci.

Confidentialité des échanges

À notre connaissance, aucun effort n'a été fait pour assurer la confidentialité des échanges. Ce serait d'ailleurs difficile à introduire sans faciliter une attaque de type déni de service. En effet, si l'on tente d'ajouter une couche cryptographique, l'on va être confronté aux deux faces d'un même problème :

- soit on établit une connexion pour chaque requête, celle-ci étant refermée dès que le serveur a répondu ;
- soit on garde une connexion ouverte entre deux requêtes.

Les deux solutions sont malheureusement difficilement adaptables à l'architecture et à la charge actuelles du système :

- la première implique qu'à chaque requête le serveur doit faire des opérations de cryptographie à clef publique, ce qui est prohibitif en termes de calculs. Une autre conséquence serait une augmentation de la latence (plusieurs aller-retours de paquets pour l'authentification et l'établissement de la clef de session) ;
- la seconde implique que le serveur garde en mémoire les paramètres des différentes connexions. Cette solution est, elle, prohibitive en termes d'espace mémoire nécessaire.

Dans un cas comme dans l'autre, la vulnérabilité du serveur à une attaque de dénis de service est considérablement accrue, car ces méthodes favorisent un épuisement de ses ressources, respectivement en puissance de calcul et en mémoire. Même sans tenir compte de ce risque, cela limite alors le nombre maximum de clients qui peuvent utiliser le serveur ;

On pourrait imaginer s'en sortir avec des protocoles basés sur l'identité [68, 16], malheureusement ils restent très lourds en calculs ; un serveur pourrait difficilement, là encore, supporter un grand nombre de requêtes simultanées.

C'est pourtant ce que nous allons tenter d'obtenir, car c'est à cette condition seulement que l'on pourra garantir la protection de la vie privée de l'utilisateur.

9.4.4 Autres abus

Pour conclure cette section, nous devons ajouter qu'il y a encore d'autres problèmes de sécurité liés à DNS et à l'usage qui en est fait. Par rapport à ce que nous venons d'évoquer, ils sont à la fois secondaires — bien que très réels — et plus « spécialisés ».

DNS peut ainsi être utilisé comme un canal de communication caché, par exemple pour contourner un firewall qui bloquerait les communications TCP mais laisserait passer les échanges DNS. Il existe d'ailleurs plusieurs implémentation de tunnels *IP over DNS*, par exemple Ozymandns[6] ou Iodine[7].

Par ailleurs, une analyse de la *forme* des messages DNS permet généralement d'obtenir des renseignements sur les ordinateurs qui les envoient, de la même manière qu'une analyse de la forme des paquets IPs permet souvent de déterminer le système d'exploitation d'une machine [82].

Dans certains cas, DNS peut également être utilisé pour contourner une politique de sécurité ; les attaques dites de *DNS Rebinding* ont par exemple permis à des sites webs malintentionnés de « jeter un oeil » dans des webmails ou des comptes de banque en ligne via la gestion des *iframe*s du navigateur Web : le nom était le même, mais l'adresse IP qui se cachait derrière était différente !

Le lecteur qui souhaiterait en apprendre plus sur les abus possibles de DNS trouvera un certain nombre de présentations intéressantes et de pointeurs sur le site de Dan Kaminsky, `http://www.doxpara.com/`.

Finalement, notons qu'en dépit de tous ces problèmes, DNS est tout de même utilisé pour améliorer la sécurité, dans des contextes où la solution qu'il permet d'obtenir est « mieux que rien ».

C'est par exemple le cas dans la lutte contre le spam ; les enregistrements SPF, qui indiquent quelles adresses IP peuvent envoyer des mails pour un domaine, sont stockés dans le DNS.

Un autre exemple est celui du chiffrement opportuniste, qui permet de chiffrer une connexion même si l'on n'est pas certain de l'identité du destinataire, afin d'éviter qu'Eve puisse écouter. Le chiffrement opportuniste a été popularisé par le défunt projet FreeS/WAN [34] (continué par les projets Openswan [58] et strongSWAN [72]), qui utilise DNS pour stocker des clefs publiques utilisables par IPsec.

[6] `http://www.doxpara.com/ozymandns_src_0.1.tgz`
[7] `http://code.kryo.se/iodine/`

10

L'intimité dans le DNS

Comme expliqué dans le chapitre précédent, le DNS de base, en dépit de son remarquable passage à l'échelle (l'Internet était bien moins grand lors de sa création), souffre de plusieurs problèmes, en particulier liés à la sécurité.

Si, nous l'avons dit, des solutions comme DNSsec existent pour remédier à la plupart de ces problèmes, il en reste un qui est très peu abordé, celui de l'intimité de l'utilisateur.

En effet, la résolution des noms se faisant de la manière décrite en 9.2.2, la situation ressemble souvent à celle de la figure 10.1 ; si Eve ou Mallory peuvent se placer entre Alice et le cache, ils voient toutes ses requêtes. Or celles-ci trahissent l'usage que fait Alice de l'Internet : nous aimerions donc trouver un moyen pour qu'elles restent confidentielles, sans pénaliser trop les performances de DNS, essentielles à la « qualité » de l'Internet telle que perçue par un utilisateur.

S'il est possible d'utiliser des systèmes d'anonymisation ou de non-observabilité génériques — comme *True Nyms* que nous avons présenté dans la partie précédente — pour protéger l'intimité des requêtes DNS, la latence reste dans le meilleur des cas très médiocre. Il semble donc qu'il faille une solution spécifique à DNS.

Le problème que nous cherchons à résoudre peut donc s'exprimer ainsi :

> « ***Comment préserver la confidentialité des échanges clients-serveurs dans DNS en gardant une bonne latence, d'une manière compatible avec l'existant ?*** »

La compatibilité avec l'existant est malheureusement quasi indispensable : il suffit pour s'en convaincre d'observer la lenteur de l'adoption d'IPv6 ou de DNSsec. Il faut donc que notre solution puisse être utilisée par ceux qui le désirent sans nécessiter de déploiement global.

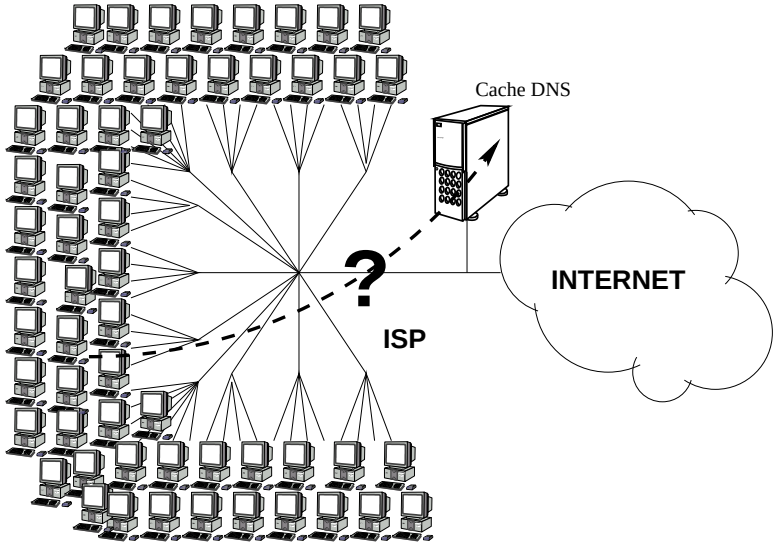


FIG. 10.1 – En général, les clients d'un fournisseur d'accès à l'Internet utilisent le cache de celui-ci.

10.1 Quelques solutions naïves

À première vue, le problème, tel que formulé ci-dessus, ne semble pas très compliqué, et on peut imaginer des solutions naïves.

10.1.1 Interroger directement la hiérarchie DNS

On peut commencer par se demander s'il serait possible qu'Alice interroge directement les serveurs de la hiérarchie DNS (sans passer par un cache), d'une manière à la fois rapide et telle qu'un observateur ne puisse observer ces échanges.

Pourtant, même en imaginant qu'il soit possible de modifier l'existant et que l'usage de la cryptographie n'ait pas de coût en calculs (seulement en mémoire), la solution consistant à chiffrer les données entre les clients et la hiérarchie DNS ne semble pas possible.

Même des protocoles de chiffrement qui fonctionnent avec UDP, comme IPsec et DTLS, nécessitent l'établissement d'une session avant de transmettre des données, pré-requis qui n'existait pas avec UDP sans protection. Même si l'on peut éviter d'ouvrir autant de *sockets* que de sessions (cf. par exemple [9]), ces sessions requièrent le stockage d'un minimum d'informations. Même en supposant que ces états prennent peu de place, cela introduirait une charge insupportable pour les serveurs en haut de la hiérarchie du DNS — qui doivent répondre à des millions sinon des milliards de clients — et augmenterait la sensibilité du système à des attaques de type déni de service.

De plus, les solveurs :

- interrogent souvent les serveurs au sommet de la hiérarchie, mais ceux-ci ne peuvent se permettre de garder des connexions entre deux requêtes ;

– sollicitent peu les serveurs en bas de la hiérarchie et rarement de manière répétitive. Il n'y a donc guère d'intérêt à maintenir des connexions avec ces serveurs.

La conséquence est qu'une connexion chiffrée devrait être établie pour chaque requête, ce qui augmenterait de manière importante la latence, puisque la requête proprement dite ne pourrait pas être émise immédiatement.

Il n'est donc pas possible d'interroger directement la hiérarchie DNS, autrement dit, il faut passer par un cache. Comme Alice ne veut pas utiliser le cache de son fournisseur d'accès à l'Internet (ou du moins il ne faut pas que celui-ci sache que les requêtes viennent d'elle), quelles sont les solutions envisageables ?

10.1.2 Un cache DNS chiffré

Une première idée, logique, serait de fournir un cache DNS, indépendant, qui permettrait aux clients de se connecter à l'aide de connexions chiffrées, par exemple par IPsec ou DTLS.

En supposant qu'Alice puisse authentifier le cache, son fournisseur d'accès à l'Internet n'a plus de moyen d'intercepter ses requêtes.

Il peut encore tenter de déduire des informations, par exemple en étudiant la taille des requêtes, leur nombre et la façon dont elles sont groupées, mais si c'est sa seule source, il semble assez peu probable qu'il puisse en déduire des informations utiles.

Cette solution n'est néanmoins pas viable :

1. l'utilisation de la cryptographie empêche le cache d'être utilisé par un grand nombre de clients, pour les raisons données en 9.4.3 ;
2. surtout cela ne fait que reporter le problème : Alice n'a plus à faire confiance à son fournisseur d'accès à l'Internet, mais elle doit cette fois se fier au gestionnaire du cache.

10.1.3 Un ensemble de caches DNS chiffrés

On pourrait bâtir sur la solution précédente, en remplaçant *le* cache DNS indépendant par un ensemble de N caches indépendants :

– il serait possible d'avoir plus de clients (N fois plus) ;
– si les caches sont opérés de manière indépendante, chacun ne verrait qu'une partie des requêtes de chaque client (la proportion exacte variant selon la manière dont les clients distribuent les requêtes, mais l'on peut imaginer descendre jusqu'à $\frac{1}{N}$).

Cette solution n'est que moyennement satisfaisante :

– même si le cache n'a qu'une chance sur N d'avoir une requête d'Alice, quand il en reçoit une, il *sait* qu'elle vient d'Alice. Comme de plus l'usage de l'Internet fait qu'Alice demandera régulièrement la résolution des noms des services/sites/domaines qu'elle utilise, chaque cache pourra à terme construire un profil ;
– l'usage de la cryptographie fera que le ratio

$$\frac{\text{nombre d'utilisateurs}}{N}$$

sera assez faible ;
- le déploiement d'un certain nombre de ces caches spéciaux est nécessaire.

Peut-on mieux faire ? Nous le pensons.

10.2 Vers une solution évoluée

Le fait que DNS soit facilement observable a principalement deux causes :

1. l'utilisateur envoie ses requêtes au(x) cache(s) d'un (unique) prestataire de services. Un observateur qui a un accès (qu'il soit légal ou non) au système de ce prestataire peut intercepter ces requêtes ;
2. même s'il configure son système de manière à ne pas utiliser ce cache (par exemple en installant son propre cache qui fera la résolution lui-même), comme les serveurs DNS présents sur l'Internet ne permettent pas de chiffrer les communications, Eve pourra toujours prendre connaissance de celles-ci en observant le réseau de son ISP.

Une solution devrait donc :
- s'assurer que le fournisseur de services qui fournit l'accès à l'Internet ne puisse observer les requêtes DNS d'un utilisateur qui passent sur son réseau ;
- fournir un moyen pour qu'un cache ne puisse pas savoir qui fait réellement la requête.

Nous décrivons une telle solution dans les chapitres qui suivent.

11

Un réseau pair-à-pair de caches DNS inspiré de Crowds

Il semble difficile de rendre confidentielles les requêtes DNS avec la topologie actuelle (figure 11.1), aussi proposons-nous de modifier légèrement celle-ci, comme illustré par la figure 11.2, de manière à remplacer (mais pas à supprimer) le cache de l'ISP par un réseau pair-à-pair de caches, inspiré de Crowds ([61], cf. 1.3.2).

11.1 Description

L'idée est que chaque machine dispose d'un cache local et est liée à n autres machines du réseau pair-à-pair par une connexion chiffrée. Lorsqu'une requête arrive, si la réponse se trouve dans le cache, on la renvoie immédiatement, sinon :

- avec une probabilité p, on transmet la requête à l'un des autres membres du réseau auquel on est connecté ;
- avec une probabilité $1 - p$ on utilise la voie normale (utilisation du cache de l'ISP) pour obtenir la réponse (figure 11.3).

Quand la réponse arrive, chaque cache la transmet à celui qui a fait la demande et la stocke. La réponse parcourt le chemin inverse de la requête, et la réponse sera cachée sur toute cette route.

Un membre du réseau peut sans problème garder établies les n connexions (de l'ordre d'une dizaine), en changeant les membres avec lesquels les connexions sont établies régulièrement. Les connexions entre les membres étant chiffrées, un observateur ne peut déterminer en observant uniquement celles-ci le contenu des requêtes qui y transitent, d'autant plus que les requêtes DNS étant limitées en taille, il est facile d'utiliser un padding pour éliminer une analyse de trafic simple de ce genre.

Quand une requête arrive au cache d'un ISP, celui-ci ne peut supposer qu'elle émane réellement de la machine qui la lui a envoyée, puisque cette dernière

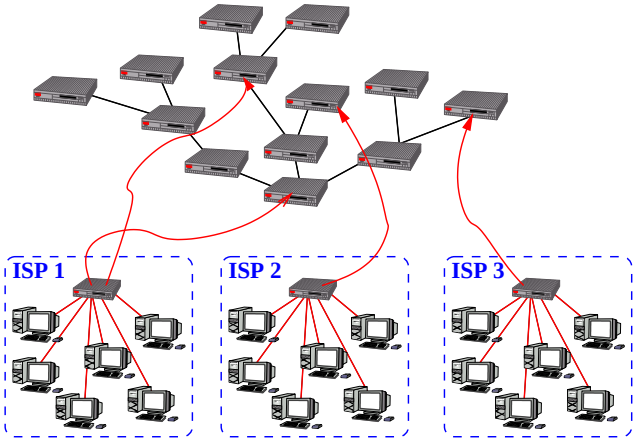


FIG. 11.1 – La topologie actuelle : chaque client interroge le cache de son ISP, qui interroge à son tour la hiérarchie DNS (cf. 9.2.2).

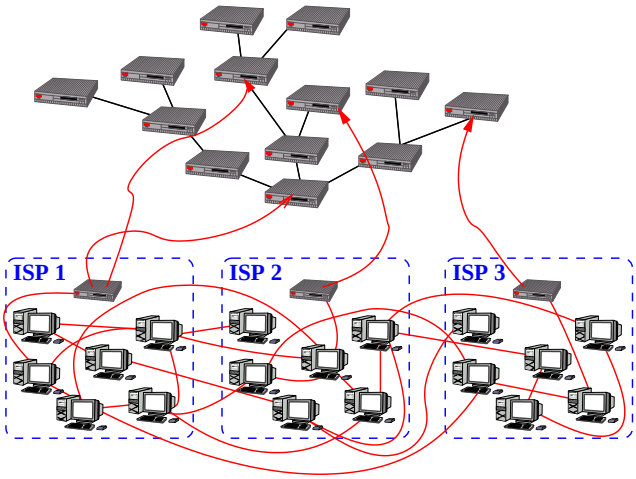


FIG. 11.2 – La topologie proposée : les clients forment un réseau pair-à-pair où chaque nœud est aussi un cache DNS. Ces caches n'interrogent pas directement la hiérarchie DNS. Les connexions entre les nœuds sont chiffrées.

peut ne faire que la transmettre. Il en va de même si l'observateur contrôle des machines du réseau pair-à-pair.

Un observateur global pourra toutefois tenter de « suivre » une requête de nœud en nœud jusqu'à ce que l'un des des cas de figure suivants se produise :

- la requête sort du réseau pair-à-pair (elle ne sera alors plus chiffrée) ;
- la requête arrive à un nœud que l'observateur contrôle.

Toutefois, il se peut que pour une requête donnée, aucun de ces deux cas ne se produise : si la requête arrive rapidement sur un nœud qui dispose de la réponse dans son cache, par exemple.

Un intérêt de cette solution est qu'elle peut être déployée par les personnes intéressées sans aucune modification du réseau DNS existant. Toutefois, elle

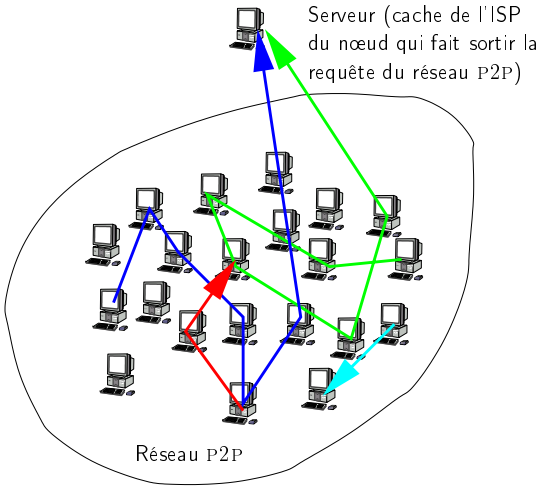


FIG. 11.3 – Une requête est renvoyée de cache en cache jusqu'à ce que l'un d'eux connaisse la réponse (cas des requêtes rouge et cyan) ou décide (probabilité $1-p$) de faire la requête par la voie traditionnelle (requêtes verte et bleue).

requiert pour être sûre un déploiement préalable de DNSsec : en effet, dans le cas contraire, un nœud malintentionné pourrait empoisonner une partie conséquente du réseau en distribuant de fausses données avec un TTL élevé (d'autant plus que ces données empoisonnées seraient à leur tour transmises aux autres nœuds).

11.2 Que doit-on cacher ?

On peut se demander s'il faut *tout* cacher. En effet, la taille d'un cache est limitée, et si des données « inutiles » s'y trouvent, c'est de la place perdue et cela augmente les chances de devoir envoyer une requête sur le réseau.

Bien sûr, aucune donnée n'est totalement inutile, néanmoins il faut rappeler que notre but ici est de permettre la résolution des noms en adresses IP. Pour cela, nous n'avons nullement besoin du contenu des champs TXT par exemple.

Un autre problème est celui des réponses négatives. Mallory pourrait demander en permanence aux caches de résoudre des noms inexistants (par exemple générés semi-aléatoirement). Si ceux-ci stockent les réponses, elles vont finir par *chasser* des données plus légitimes, provoquant à la fois un ralentissement du temps de réponse du système et augmentant la probabilité que Mallory puisse voir une requête.

Que devrait-on donc cacher exactement ? Nous proposons de nous limiter :
- aux adresses (IPv4 et/ou IPv6) ;
- aux données relatives à DNSsec.

Les autres types de données (à quel serveur envoyer un mail, etc.) sont a priori moins importants et/ou peuvent accepter un temps de latence plus grand.

Ceci dit, s'il faut protéger le cache contre les remplissages abusifs, il faut peut-être se demander également comment son remplissage doit évoluer natu-

rellement.

11.3 Remplissage du cache

L'un des paramètres importants de notre système concerne le remplissage moyen des caches des nœuds. En effet, si les caches contiennent un grand nombre de données, le parcours moyen d'une requête avant de trouver la réponse est plus court. À l'inverse, si tous les caches sont pratiquement vides, cela signifie que, pour la plupart des requêtes, il va falloir attendre la sortie du réseau pair-à-pair pour obtenir une réponse, et donc que la latence est comparable à celle que l'on aurait observé avec l'Onion-Routing.

Le problème se complique du fait qu'une requête qui sort du réseau pair-à-pair voit sa réponse stockée dans tous les nœuds de la route qu'elle a suivie. Elle apporte donc des informations à ces caches. A contrario, une requête qui trouve tout de suite sa réponse n'ajoute rien aux caches avoisinants.

On semble donc être dans une situation où

$$\begin{array}{ccc} \text{caches faiblement remplis} & \Longrightarrow & \text{chemin des requêtes long} \\ \Uparrow & & \Downarrow \\ \text{chemin des requêtes courts} & \Longleftarrow & \text{caches bien remplis} \end{array}$$

D'autre part, à un instant donné, chaque cache compte environ n « utilisateurs » et, nous l'avons dit, n est a priori petit. Cela revient donc plus ou moins à avoir un cache pour n utilisateurs.

Or, il est connu [43] que l'utilité d'un cache quand l'on n'a qu'un petit nombre d'utilisateurs est faible, car le taux de défauts de cache est élevé. En effet :

- l'intersection des données qui intéressent à la fois des utilisateurs A et B est faible ;
- la durée de vie de ces données est faible par rapport à leur fréquence d'utilisation (voir aussi la figure 11.4).

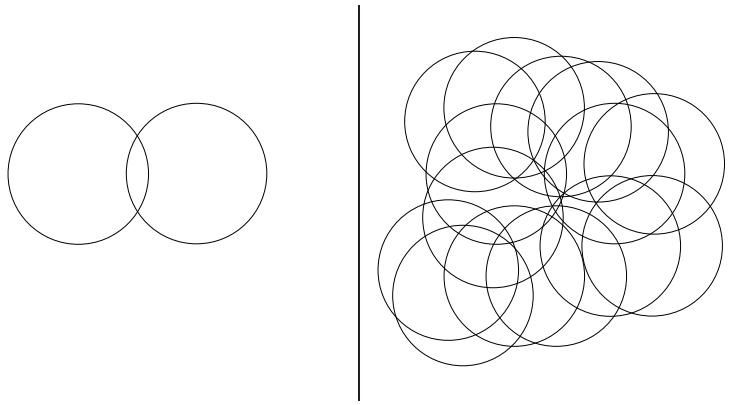


FIG. 11.4 – Avec un petit nombre (x) d'utilisateurs (à gauche) peu de données sont utiles à plusieurs utilisateurs. Avec un grand nombre (X) d'utilisateurs (à droite), la plupart des données dans le cache sont utiles à un grand nombre de personnes, même si l'intersection de chaque petit groupe de x personnes est faible.

Une analyse plus fine semble nécessaire, en tenant compte :

- du fait que nos n utilisateurs ne sont pas les mêmes sur la période donnée ;
- du fait que nos n utilisateurs sont eux-mêmes des caches avec n utilisateurs de ce même type chacun (autrement dit, l'un de ces « utilisateurs » fait-il plus ou moins de requêtes qu'un utilisateur ordinaire ?).

Malheureusement, cette étude rigoureuse reste hors de portée pour le moment, puisqu'il n'existe pas à notre connaissance de modélisation des requêtes DNS (tout au plus sait-on qu'elles ne suivent pas une loi de Poisson [75]).

Il semble donc qu'il faille envisager le cas où chaque cache ne serait que peu rempli. Si cette hypothèse se confirme et si l'on adopte ce système en choisissant p de telle sorte qu'en cas de défaut de cache, en moyenne, la latence soit similaire à celle d'une requête au-dessus de *True Nyms*, la latence moyenne totale (en tenant compte de toutes les requêtes) ne sera que marginalement meilleure (il faudrait toutefois tenir compte de la probabilité que la donnée soit par exemple dans le cache auquel on renvoie la requête).

Dans le chapitre qui suit, nous étudions les moyens pour nous assurer que notre cache contient un grand nombre de données.

12
Remplir le cache

Dans le chapitre précédent, nous avons proposé de remplacer *le* cache traditionnellement utilisé par un *réseau* de caches d'une manière inspirée de *Crowds*.

L'une des inconnues de ce système est le niveau de remplissage du cache ; si celui-ci devait être bas, l'utilité de ces caches serait faible et les performances guère meilleures que si l'on faisait simplement les requêtes de la manière traditionnelle en utilisant *True Nyms* pour les protéger.

Pourrait-on imaginer un ou des moyens pour maintenir le cache plein ?

Il nous semble que la réponse est nettement « oui ».

12.1 Auto-rafraîchissement des données

Pour empêcher les données du cache d'expirer, une solution serait de rafraîchir celles-ci lorsqu'elles sont sur le point d'expirer. On récupérerait ainsi les données avec un nouveau TTL.

Il nous semble toutefois qu'il serait trop rigide de vouloir absolument rafraîchir toutes les données au moment où elles expirent. Nous envisageons plutôt une approche où le cache choisit des enregistrements au hasard, périodiquement, et les rafraîchit si leur TTL restant est inférieur à un temps t_{seuil}.

Une telle méthode non-déterministe ajoute un peu d'aléa dans le système, ce qui dans notre contexte peut difficilement être une mauvaise chose. Le revers de la médaille est bien sûr qu'il reste possible qu'une donnée ait expiré au moment où elle est demandée.

Toutefois, les sections qui suivent décrivent des méthodes qui permettent de limiter ce problème.

12.2 Tricher sur la date de péremption

L'un des problèmes est que certains domaines diffusent leurs adresses avec une durée de vie très faible, parfois de l'ordre de la dizaine de secondes.
Par exemple :

- Akamai fournit des services de cache Web et semble utiliser le DNS à des fins de *load-balancing* ;
- DynDNS est un prestataire de services DNS qui permet de conserver une URL avec une adresse IP dynamique. Les adresses sont distribuées avec des TTL faibles afin qu'un changement d'adresse IP ne « casse » l'`url` que peu de temps.

Cela pose le double problème suivant :

- le cache va demander ces données très souvent, ce qui encombre inutilement le réseau ;
- en dépit du rafraîchissement, il reste très probable que les données soient périmées au moment où un client les demande. On ne pourra donc pas lui répondre rapidement.

Une solution « politiquement correcte » à ce problème serait de ne pas utiliser le mécanisme d'auto-rafraîchissement proposé dans la section précédente pour les données qui ont un TTL inférieur à une certaine valeur T quand on les reçoit. Cela implique toutefois que les requêtes pour ces données seront lentes, puisqu'il faudra généralement les recharger.

Nous proposons une autre solution, que nous pourrions qualifier d'« incorrecte » et qui sera sans doute même qualifiée d'hérésie par certains. Elle consiste à noter que dans la pratique, la durée de validité des données est supérieure à celle du TTL[1]. Cette solution revient à *changer le* TTL, c'est à dire à décider que :

$$\text{TTL}_{\text{stocké}} := \max(\text{TTL}_{\text{indiqué}}, T).$$

Bien entendu, il est hors de question de transmettre les données avec ce nouveau TTL aux autres machines : en effet, comme dans notre réseau pair-à-pair les relations entre les machines ne sont pas exemptes de cycles, on aurait des données qui n'expireraient jamais, même si elles n'existent plus sur le serveur d'origine ! Toutefois, on peut transmettre ces données à un client local sans grand risque. Dans la section suivante, nous examinons comment diminuer encore ce problème.

12.3 Servir des données périmées

Dans un cache normal, si l'on s'aperçoit au moment d'envoyer des données que celles-ci ne sont plus valides, on les rafraîchit alors.

[1] Les administrateurs de réseaux utilisent souvent des valeurs assez faibles (inférieures à 24 heures) pour que les changements soient propagés rapidement si jamais ils devaient faire, pour une raison quelconque, une modification non planifiée en urgence. Dans la pratique, de telles modifications sont bien sûr très rares.

Pourtant, une donnée périmée n'est pas forcément une donnée avariée ! C'est d'ailleurs ce qui nous a permis de tricher sur la date de péremption dans la section précédente — il faut juste faire attention à ne pas repousser éternellement cette date.

Pourrait-on utiliser ce principe pour aller plus loin ? Il semble parfaitement légitime de supposer que notre système ne sera pas utilisé comme serveur par des programmes de résolution DNS ne se trouvant pas en bout de chaîne. Par conséquent, il est possible de se mettre d'accord sur une convention : un TTL de 0 signifie à la fois que :

- ces données sont officiellement périmées ;
- ces données sont probablement encore valables ;
- une version à jour suit.

Ainsi, le programme qui fait la requête peut avoir le choix entre :

- la rapidité au prix d'un léger risque ;
- la sûreté au prix d'un délai.

Pour que le client puisse faire ce choix lui-même, cela nécessite bien sûr une mise à jour de l'interface par laquelle il fait la résolution d'adresse. À défaut, il serait possible de faire ce choix pour tous les programmes de la machine, par exemple dans la configuration de l'implémentation de notre système.

Notons qu'en C, les programmes récents utilisent la fonction `getaddrinfo` (bien que d'anciennes interfaces comme `gethostbyname` soient encore d'un usage courant). Cette fonction `getaddrinfo` permet de passer en paramètre une `struct addrinfo` contenant des « indices » sur la manière dont la résolution doit être faite, définie ainsi :

```
struct addrinfo {
        int ai_flags;
        int ai_family;
        int ai_socktype;
        int ai_protocol;
        socklen_t ai_addrlen;
        char *ai_canonname;
        struct sockaddr *ai_addr;
        struct addrinfo *ai_next;
};
```

On note en particulier que le champ `ai_flags` de cette structure permet de passer des options. On pourrait imaginer l'ajout d'une option permettant de faire le choix sus-mentionné.

Notons que la RFC [54] prévoit que des données avec un TTL nul ne doivent être utilisées que pour la connexion en cours. Un client qui recevrait des données avariées et non-fonctionnelles ne restera donc pas bloqué avec ces données mais refera la requête.

12.4 Synchroniser réponses et rafraîchissements

Dans le système décrit, si notre observatrice, Eve, voit une requête chiffrée arriver sur un nœud n (elle ignore le contenu de la requête puisque celle-ci est chiffrée), elle peut déduire des informations selon le comportement observé :

1. des données repartent en sens inverse, aucune autre donnée n'est émise : la réponse se trouvait dans le cache de n ;
2. des données repartent en sens inverse, d'autres données sont émises vers un autre nœud ou au DNS : la réponse était dans le cache mais périmée. C'est elle qui est renvoyée en sens inverse, les autres données émises sont destinées à rafraîchir la réponse. Si Eve prend connaissance de ces données, elle connaîtra le contenu de la requête originale ;
3. des données sont émises vers un autre nœud ou vers le DNS normal ; aucune donnée n'est émise immédiatement vers le nœud d'origine : les données demandées n'étaient pas dans le cache. La situation pour Eve est la même que dans le cas précédent.

Pour compliquer un peu la tâche d'Eve, on peut de plus imaginer synchroniser le rafraîchissement des données d'un de nos nœuds avec l'arrivée d'une requête. Ainsi, même si Eve réussit à obtenir le contenu de la requête, elle ne saura pas si c'est celle qu'a reçu le nœud n, ou s'il en a profité pour faire une de ses requêtes propres, c'est-à-dire qu'elle ne pourra plus distinguer les cas 1 et 2.

Par ailleurs, un nœud pourrait envoyer des « informations gratuites », aléatoirement, mais aussi quand il reçoit une requête dont il ne connaît pas la réponse. Ainsi,

- Eve ne pourra plus distinguer les cas 2 et 3 (ni, d'après le paragraphe précédent, le cas 1) ;
- le nœud qui reçoit ces données gratuites peut, s'il a de la place, les stocker (on suppose qu'elles ne peuvent être fausses, car protégées par DNSsec).

12.5 A-t-on encore besoin du réseau P2P ?

À la limite, on peut pratiquement se demander si le réseau pair-à-pair dédié est encore utile : en effet, les données dont Alice a besoin seront alors presque toujours dans son cache local ; les rares fois où elles n'y sont pas, ne serait-il pas suffisant de les récupérer en passant au-dessus d'un système plus générique comme *True Nyms* ?

Nous examinons cette idée et quelques variantes dans le chapitre suivant.

13
Trois options envisageables

Nous terminions le chapitre précédent en nous demandant si nous avons réellement besoin du réseau pair-à-pair « à la Crowds ».

Dans ce chapitre, nous examinons les avantages et inconvénients de différentes options :

- un cache désynchronisé isolé (section 13.1) ;
- un cache désynchronisé isolé qui utilise *True Nyms* pour ses communications avec l'extérieur (section 13.2) ;
- le réseau pair-à-pair de caches désynchronisés tel que nous l'avons décrit jusqu'ici (section 13.3).

Nous terminons ce chapitre sur une comparaison récapitulative de ces options.

13.1 Un cache désynchronisé local

Imaginons en effet qu'Alice utilise une version modifiée de ce système de protection de DNS avec tous les mécanismes d'*auto-rafraîchissement* et l'usage de données périmées, mais fonctionnant de manière autonome, sans le réseau pair-à-pair. À côté de cela, elle utilise un système d'Onion-Routing simple, comme Tor pour protéger ses connexions TCP et surfer sur le Web. Le risque qu'elle veut éviter est qu'Eve ou Mallory sachent à quel site elle se connecte à un instant donné et utilisent cette information pour suivre sa navigation en se basant sur la taille des données, à la manière de ce que nous décrivons contre SSL au chapitre 1.

Grâce à notre protection de DNS, Eve et Mallory voient le contenu des requêtes DNS, mais l'émission de celles-ci n'est plus synchronisée avec la navigation d'Alice sur le Web. Si les attaquants peuvent toujours tenter l'analyse de trafic, ils doivent à présent comparer les données chiffrées non pas avec *un* site Web, mais avec *l'ensemble* des sites dont le cache d'Alice contient l'adresse.

Leur attaque par analyse de trafic est donc rendue plus délicate, mais d'une

manière difficilement quantifiable : cela dépendra en grande partie du contenu du cache d'Alice, et donc de l'usage de l'Internet que fait celle-ci.

Par rapport à un système qui dissimule également les requêtes DNS (comme *True Nyms* ou Tor en s'assurant que DNS est encapsulé), un tel cache provoque une fuite d'informations, puisque Eve peut prendre connaissance du trafic DNS.

La nécessité d'utiliser DNSsec est moins importante que dans le cas du système pair-à-pair décrit précédemment : le cache utilisé est celui du fournisseur d'accès. Toutefois, si DNSsec n'est pas utilisé, Mallory peut tenter de fournir de fausses informations.

Un autre problème potentiel est que le cache contient alors exclusivement des adresses dont *Alice* a demandé la résolution. Si la machine d'Alice est compromise, le contenu du cache permet de connaître en partie son usage de l'Internet.

13.2 Un cache désynchronisé qui utilise TN pour le trafic DNS

L'unique différence par rapport à la solution précédente est qu'au lieu de faire les requêtes DNS de manière ordinaire, on les fait passer au-dessus de *True Nyms*. Le cache DNS pourrait ainsi maintenir en permanence une route, et l'utiliser pour envoyer les requêtes.

Il y a trois variantes possibles selon que ce mécanisme est utilisé :

1. pour interroger le cache de son ISP (pas forcément possible, suivant la configuration de celui-ci) ;
2. pour interroger un (ou plusieurs) autres caches ;
3. pour faire la résolution selon la manière décrite en 9.1.2.

Dans tous les cas, on suppose que le ou les serveurs DNS ne se trouvent pas eux-mêmes dans le réseau de pairs de *True Nyms*.

Dans le premier cas, s'il y a un certain nombre de personnes qui utilisent ce système, il ne sera pas possible de lier les requêtes arrivant au cache avec un utilisateur. Il en va de même dans le second cas, si l'on suppose que plusieurs utilisateurs utilisent le même cache ou que l'association utilisateur/cache reste secrète.

Il en est a fortiori de même dans le troisième cas, puisqu'il n'y a plus de cache.

Pour éviter à Alice de connaître les adresses de caches autres que celui de son ISP, on pourrait imaginer l'ajout dans le protocole *True Nyms* d'une requête « résolution d'un nom » dans ce que l'on peut demander à un nœud une fois la connexion établie (cf. 6.1.2).

Dans tous les cas, par rapport à la solution proposée dans la section précédente, Eve ne peut plus avoir connaîssance des requêtes d'Alice ; la sécurité en est donc améliorée. Mallory peut toujours, par contre, s'il a des soupçons sur le cache qu'utilise Alice, tenter de l'empoisonner. La solution à ce problème existe, il s'agit de DNSsec. Encore une fois, il n'« y a qu'à » la déployer.

Si l'on considère le cas où l'on a ajouté une option de résolution dans *True Nyms*, Mallory (le Mallory qui espionne Alice) ne peut a priori pas attaquer la

résolution d'Alice ; par contre, des nœuds *True Nyms* malintentionnés peuvent le faire (même s'ils ne connaissent pas l'identité de leurs victimes).

Un problème demeure toutefois : le cache d'Alice continue à être un enregistrement des machines auxquelles Alice s'est connectée.

Un nouveau problème apparaît : si les performances quand Alice demande la résolution d'un nom connu sont bonnes — la réponse est normalement déjà dans le cache — il n'en va pas de même quand la réponse n'est pas connue, par exemple parce qu'elle visite un nouveau site.

En effet, dans ce cas, la résolution du nom se fait au-dessus de *True Nyms*, avec les problèmes de latence déjà évoqués.

13.3 Un réseau P2P de caches (désynchronisés)

Cette troisième option n'est autre que le réseau de caches pair-à-pair conçu dans les chapitres précédents : nous ne le décrirons donc pas davantage ici.

Avec ce système, Eve ne peut prendre connaissance des requêtes d'Alice. Mallory peut dans certains cas en prendre connaissance (si un pair qu'il contrôle reçoit une requête venant de la machine d'Alice alors qu'il sait que celle-ci n'en a pas reçu), mais sans avoir la certitude qu'il s'agit réellement de données demandées par Alice : il pourrait s'agir du rafraîchissement d'une donnée du cache, demandée précédemment par un autre pair.

Les données dans le cache ne sont pas uniquement celles demandées par Alice, sa compromission ponctuelle ne révèle pas ou peu d'information.

Notons que bien que basé sur Crowds (cf. 1.3.2), ce système n'est pas sensible à l'attaque des prédécesseurs :

- une requête DNS contient beaucoup moins d'éléments spécifiques qu'une requête Web ;
- la présence de caches agressifs fait qu'une requête sera répétée par des hôtes autres que celui qui en fait la demande.

Les inconnues du système sont l'évolution de l'état des caches, et donc les performances moyennes :

- les adresses qu'utilise Alice vont-elles rester dans le cache, ou seront-elles chassées par la mise en cache des données demandées par les autres pairs ?
- si une donnée demandée par Alice n'est pas dans son cache local, quelle est la probabilité qu'elle la trouve rapidement en envoyant la requête à un pair ?

13.4 Comparaison récapitulative

Le tableau 13.1 compare les principaux points de ces différentes options au DNS classique.

On voit que notre première option (section 13.1), si elle est, a priori, celle qui offre le moins de protection, a l'avantage d'offrir des résultats connus et

	Cache désynchronisé seul (section 13.1)	Cache désynchronisé + *True Nyms* (section 13.2)	Réseau de caches désynchronisés P2P (section 13.3)
Vitesse si les données sont dans le cache local	+	+	+
Vitesse si les données ne sont pas dans le cache local	≈ DNS classique	−−−	? ? ?
Probabilité que les données soient dans cache local	++	++	+ (?)
Protection offerte	+	+++	++
Dépendance du système vis-à-vis de DNSsec	≈ DNS classique	+ / −	+++

TAB. 13.1 – Résumé de la comparaison entre les différentes propositions (les symboles '+' et '−' se réfèrent à un client DNS classique interrogeant le cache d'un ISP). Notons qu'un '+' ne signifie pas forcément « mieux » : ainsi une plus grande dépendance que le DNS classique à DNSsec n'est pas un avantage.

meilleurs ou équivalents à ceux du DNS classique.

Inversement, notre seconde option (section 13.2), qui combine la précédente à *True Nyms*, offre une très forte protection, mais avec de très mauvaises performances quand les données demandées ne sont pas dans le cache.

Finalement, notre troisième option (section 13.3), qui est aussi celle que nous avons construite depuis le début de cette partie, offre une très bonne sécurité, mais il n'est pas possible de prédire ses performances a priori. Celles-ci détermineront donc s'il vaut mieux choisir cette dernière option (si les performances sont bonnes), ou si chaque utilisateur devrait plutôt, selon ses choix personnels (le fameux compromis performances / sécurité) choisir l'une des deux options précédentes. C'est pourquoi il est important d'implémenter et de tester cette option.

14

Implémentation et tests de notre proposition

Nous l'avons déjà mentionné, il n'existe pas à notre connaissance de bonnes modélisations de la loi de probabilité qui régit les requêtes DNS. C'est pourquoi, afin de valider les propositions décrites dans les chapitres précédents, nous avons implémenté celles-ci. Le programme résultant est un démon au sens Unix du terme, qui devrait fonctionner sur la plupart des systèmes respectant la norme POSIX et disposant d'un compilateur C99.

Dans ce chapitre, nous décrivons cette implémentation, puis testons ses performances.

14.1 L'implémentation

Ce programme implémente les trois options du chapitre précédent, des options de compilation permettant de choisir l'une ou l'autre.

Notons que ce prototype est beaucoup plus « expérimental » que celui de *True Nyms*. Certaines fonctionnalités nécessaires pour un déploiement réel n'ont pas été implémentées. Par exemple, étant donné le déploiement actuel de DNSsec, il ne nous a pas semblé utile de le gérer.

Notre implémentation communique avec l'existant en se faisant passer pour un cache DNS normal, qui écoute sur le port UDP 53 de l'interface locale de la machine. Sur la plupart des systèmes Unix, il suffit donc de mettre la ligne « `nameserver 127.0.0.1` » dans le fichier `/etc/resolv.conf` pour l'utiliser.

Les idées directrices de notre style de programmation sont les mêmes que pour `truenyms` (cf. chapitre 7). Notre implémentation est donc découpée en

plusieurs modules, chacun regroupant des fonctions sur le même thème, et l'on peut pour la plupart d'entre eux avoir une vision orientée objets : chaque module implémente un objet, et les fonctions qui y sont définies en sont les méthodes.

Les principaux modules sont les suivants :

- `pdns.c` qui contient les fonctions de base (`main` et fonctions assemblant les différents autres modules) ;
- `dns.c` qui contient les fonctions de lecture et d'écriture de paquets DNS ;
- `cachedatastruct2.c` implémente le cache et les fonctions qui permettent d'y accéder ;
- `p2p.c` contient les fonctions liées à la gestion du réseau pair-à-pair.

Le programme utilise également des modules / bibliothèques que nous avions développés précédemment, tels qu'un module de gestion de listes, un module chargeant le fichier de configuration ou encore un module contenant diverses fonctions liées au réseau (ces modules sont d'ailleurs également utilisés par `truenyms`).

Les sous-sections suivantes reviennent sur les quatre principaux modules.

14.1.1 Le cache lui-même : `cachedatastruct2.c`

Le cache lui-même est constitué d'un gros fichier mappé en mémoire et comportant des enregistrements de taille fixe (512 octets).Ces enregistrements sont gérés à la manière des *inodes* d'un système de fichiers : la structure logique est donc une structure d'arbre qui suit la topologie des noms DNS (figure 14.1), avec trois types d'enregistrements :

- des enregistrements `struct dnsdata`. Nous pourrions les qualifier d'« enregistrements par défaut » : il contiennent un nom de domaine et la possibilité d'accueillir une quantité limitée de données et de liens vers des fils. Ils pointent[1] vers des enregistrements des deux autres types, décrits ci-dessous, si un nœud a plus de fils ou s'il est nécessaire de stocker plus de données ;
- des enregistrements `struct othersons` qui sont utilisés quand tous les emplacements pour indiquer des fils d'une `struct dnsdata` sont occupés. Les `struct othersons` d'un enregistrement `dnsdata` forment une liste doublement chaînée ;
- des enregistrements `struct otherdata`. Ils sont utilisés quand il faut stocker plus de données à un endroit de l'arbre que peut en contenir une `struct dnsdata`. Les `struct otherdata` d'un enregistrement `dnsdata` forment une liste simplement chaînée.

Les enregistrements `struct dnsdata` sont de plus liés entre eux par une structure de liste doublement chaînée (non représentée sur la figure 14.1) modifiée lors des accès : les éléments en début de liste sont ceux qui ont été demandés le plus récemment (et réciproquement). Cette liste est utilisée lorsque le cache est plein pour éliminer les éléments a priori les moins utiles.

14.1.2 Analyse et création de paquets DNS : `dns.c`

Le module de gestion et d'analyse des paquets n'a rien de notable, si ce n'est qu'il s'agit probablement de la partie du programme la plus sensible en termes

[1] Il ne s'agit pas de « vrais » pointeurs, mais d'index (en nombre d'enregistrements) dans notre cache.

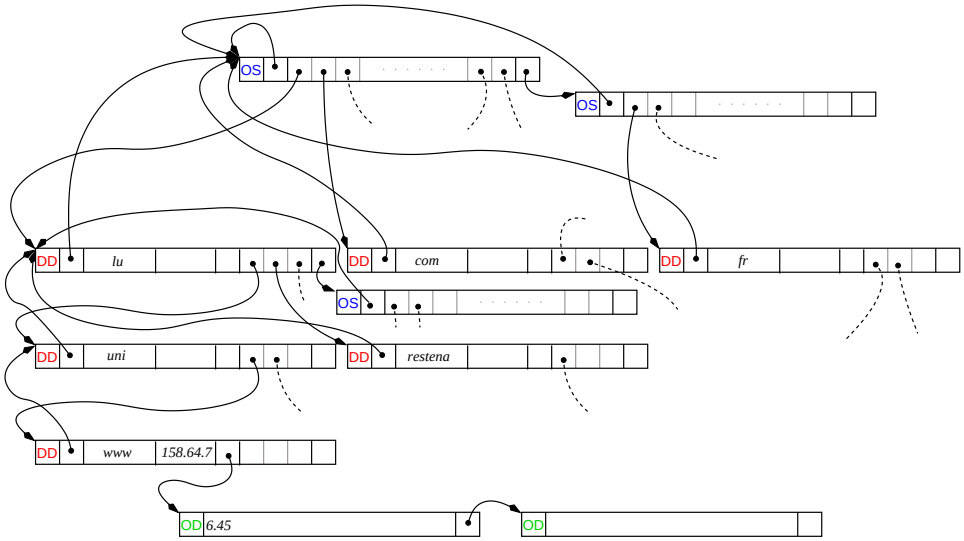


FIG. 14.1 – Un aperçu de la structure logique du cache : exemple de liens entre les enregistrements de type `struct dnsdata` (DD), `struct othersons` (OS) et `struct otherdata` (OD). Les pointillés représentent des liens vers des structures non représentées.

de sécurité. C'est en effet ce module qui transforme les messages DNS lus sur le réseau en des structures de données C.

Il doit donc entre autres purger et valider les données, pour s'assurer que les informations passées au reste du programme sont aussi fiables que possible (et ne tentent pas par exemple de faire déborder un tampon). Bien entendu, la présence d'un module qui purge les données ne doit pas être prise comme une autorisation au laxisme pour la programmation des autres modules ! Un démon réseau se doit d'être programmé de manière défensive et d'utiliser autant que possible des principes de défense en profondeur.

14.1.3 La gestion du réseau P2P : `p2p.c`

La gestion du réseau P2P proprement dite n'a rien de remarquable sur le plan de la programmation.

Dans les points notables du protocole, on remarquera que nous avons profité de ce que nos pairs sont des caches DNS : les échanges de messages dans le cadre du réseau P2P se font par des requêtes DNS spéciales dans un domaine que nous contrôlons (`pdns.lafraze.net`).

Ainsi par exemple, un pair qui reçoit une requête demandant la résolution de `list.pdns.lafraze.net` n'interrogera pas son cache mais renverra directement une réponse avec les adresses de quelques-uns des pairs qu'il connaît.

Ce procédé permet également de *bootstrapper* de manière très simple un client : en demandant la résolution de `list.pdns.lafraze.net` par le DNS classique, il obtiendra une liste de pairs racines[2].

[2]Pour paraphraser Orwell, « tous les pairs sont égaux, mais certains sont plus égaux que

Les communications entre les pairs sont chiffrées avec TLS[3], les requêtes sont *paddées* au multiple de 512 octets le plus proche. Notons que l'emploi d'un padding n'est pas gênant, même si l'une de ces requêtes devait s'« échapper » et arriver à un serveur DNS ordinaire : en effet, la structure même des requêtes DNS (cf. 9.3) fait que ce padding serait ignoré (certaines implémentations affichent toutefois un avertissement de faible priorité dans un journal).

Notons que cette partie peut-être exclue du programme (cas des options décrites en 13.1 et 13.2).

14.1.4 L'interface entre les modules et la gestion du DNS : `pdns.c`

Le module `pdns` est le point central du programme : c'est lui qui écoute le réseau.
Quand une requête arrive, ce module :

1. la passe au module `dns` qui convertit la requête d'une forme « message DNS » en une structure de données (au sens de C) nettoyée ;
2. examine s'il s'agit d'une requête spéciale P2P en regardant si elle concerne le domaine `pdns.lafraze.net` :
   - si c'est le cas, il la transmet au module `p2p`, qui lui rend alors une réponse (sous forme d'une structure C également),
   - sinon, il interroge le module `cachedatastruct2`, celui-ci lui renvoie les données (toujours sous forme de structure C) s'il les a, et indique éventuellement si elles sont périmées.
3. s'il a une réponse à renvoyer, il passe celle-ci au module `dns` pour la convertir en un message DNS puis la transmet à la machine qui a fait la requête ;
4. s'il n'a pas de réponse pour le moment ou si la réponse est périmée,
   - il stocke la requête dans une liste de requêtes en cours,
   - il passe la requête au module `dns` pour création d'un message DNS correspondant,
   - il lance un dé et selon le résulat il envoie ce message à un pair ou au cache de l'ISP local.

Quand une réponse arrive, ce module :

1. la passe au module `dns` qui convertit la réponse d'une forme « message DNS » en une structure de données (au sens de C) nettoyée ;
2. regarde s'il attendait cette réponse (présence d'une requête correspondante dans sa liste des requêtes en cours) :
   - s'il ne l'attendait pas et qu'il a de la place dans le cache, il lance un dé pour savoir s'il la stocke (et le traitement s'arrête),
   - s'il l'attendait, le traitement continue à l'étape suivante ;
3. si le TTL est non-nul, il enlève la requête correspondante de la liste des requêtes en cours ;

d'autres ».

[3] Les communications sont donc au-dessus de TCP. Nous avons étudié la possibilité d'utiliser UDP et DTLS [9], avant de conclure que cela n'offrait aucun avantage réel dans notre cas.

4. si le TTL est non-nul, il stocke la réponse dans le cache (par l'intermédiaire du module `cachedatastruct2`) ;
5. il demande au module `dns` de convertir la réponse en un message DNS, et envoie celle-ci.

Ce module parcourt périodiquement la liste des requêtes en cours : il réitère l'envoi d'une requête (éventuellement à un autre pair) si certaines sont « anciennes » et n'ont pas reçu de réponse (avec un TTL strictement positif).

On note que dans tous les cas, aucun message DNS reçu n'est transmis directement : ils sont toujours « déconstruits » puis reconstruits après passage par une forme « structure C », les paramètres tels que le numéro d'identification de la requête étant changés. Cela permet notamment de s'assurer qu'aucun canal caché ne se trouve dans la structure des messages[4].

14.2 Tests

Nous n'avons malheureusement pas pu faire de tests sur les performances de nos différentes propositions : en effet pour être intéressant, il faudrait que chaque noeud de notre réseau de cache ait au moins un utilisateur humain. Si l'on déploie un réseau de caches mais que les requêtes n'ont qu'une seule source, tous les caches auront tendance à avoir le même contenu et les résultats des tests ne seront pas significatifs.

Les tests de bon fonctionnement du programme (lors de son développement) ont toutefois permis quelques observations informelles, qui suivent.

Nous avons constaté qu'avec un cache agressif isolé (faisant la résolution via le cache de l'ISP local en cas de défaut), les performances de DNS étaient au moins aussi bonnes qu'en son absence (le contraire eût été étonnant), voire bien meilleures dans le cas où la liaison réseau est mauvaise (cas d'une mauvaise ligne ADSL).

L'évolution du nombre d'entrées dans le cache est rapide au début (environ 900 entrées après une semaine), puis augmente lentement (environ 7000 entrées au bout de cinq mois). L'« agressivité » du cache n'a pas d'impact notable sur la charge du réseau.

Sur la durée de ces tests, nous n'avons observé qu'une seule fois le cas où une entrée servie est périmée alors que la donnée réelle a changé : il s'agissait d'un serveur Web sur une IP dynamique. Le rechargement de la page dans le navigateur a suffi à résoudre le problème, l'enregistrement de notre cache ayant été mis à jour entre-temps.

[4] Voir par exemple [82] pour les possibilités d'identification d'une machine en se basant sur la position de la valeur du champ d'identification par rapport à un attracteur étrange constitué avec l'ensemble des valeurs reçues.

15
Conclusion sur la protection du DNS

Dans cette partie, nous avons présenté une solution pour protéger les échanges entre clients et serveurs DNS : Alice peut ainsi résoudre des adresses sans qu'Eve ou Mallory n'obtiennent d'informations sur celles-ci.

Nous avons présenté différentes options, avec chacune leurs avantages et leurs inconvénients. Elles sont globalement basées sur deux idées :
- un réseau pair-à-pair de caches DNS ;
- le rafraîchissement agressif des caches.

Bien qu'il n'ait pas été possible de tester réellement toutes ces options, les premiers résultats sont encourageants.

La prochaine étape en vue de la poursuite de ces travaux est donc claire : il faut faire des tests approfondis des différentes options. Ces tests permettraient de mesurer l'impact des diverses protections, et donc de faire un choix informé en fonction du compromis sécurité / performances de chacune.

Notons par ailleurs que, même si notre proposition n'est pas conçue à cet effet, elle peut également réduire l'impact d'un déni de service sur le DNS, puisque les données dans les caches n'expirent pas. Dans ces circonstances, elle n'offre toutefois aucune garantie : il n'est pas certain que l'on trouve la réponse. Comme le chemin suivi par les requêtes n'est pas déterministe, deux requêtes successives peuvent l'une réussir et l'autre échouer.

Il serait possible d'étendre ce travail pour fournir une protection plus efficace contre les dénis de services, les requêtes n'étant plus renvoyées « au hasard » en cas de défaut de cache (des propositions existent pour utiliser un réseau P2P pour servir des données DNS, comme [24], toutefois elles visent à *remplacer* l'architecture existante et non à la compléter). Cependant, il faudrait prendre soin qu'un attaquant ne puisse obtenir des informations sur les requêtes en analysant le chemin qu'elles suivent.

That's the kind of society I want to build. I want a guarantee – with physics and mathematics, not with laws – that we can give ourselves real privacy of personal communications.

John Gilmore, *in* [36]

16
Conclusion générale

À une époque où l'usage de l'Internet est à la fois constant et sensible, la vie privée se doit d'y être protégée.

Ici, la *protection légale* est illusoire : quand bien même elle existe, comment pourrait-elle être réellement dissuasive quand elle est spécifique à un pays alors que l'Internet est mondial ? Quand la probabilité que l'attaquant soit détecté est proche de zéro ? Quand le rôle même de l'attaquant est tenu par des états ?

Par conséquent, il faut se tourner dans une autre direction, plus satisfaisante : une *protection technique efficace*.

Notre but dans ce document est de proposer une *base solide* pour une telle protection. Après avoir expliqué pourquoi les communications dites « sécurisées » ne sont pas une solution et briêvement rappelé les principales pistes existantes pour résoudre ce problème, nous avons décrit deux solutions complémentaires :

- l'une, générique, est destinée à assurer la non-observabilité des communications à faible latence en général et impose peu de contraintes sur le protocole de niveau supérieur (*True Nyms*, partie 1). Cette solution ajoute à l'Onion-Routing diverses méthodes de protection contre l'analyse de trafic afin de prévenir toute observation par un attaquant, que celui-ci soit passif ou actif. Le système est générique, mais augmente toutefois la latence par rapport à une connexion ordinaire, ce qui peut poser des problèmes pour certaines applications ou certains protocoles ;
- l'autre est spécifique : elle vise à protéger la vie privée dans le *Domain Name System* (DNS) uniquement, mais avec une faible latence (partie 2).

Il ne s'agit néanmoins que d'une *base* solide. Si elle est suffisante pour certaines applications, il reste encore bien du chemin à accomplir avant que l'intégralité de l'Internet soit « *privacy-compliant* ».

Nous l'avons dit en introduction, l'anonymat est bien sûr à double tranchant : un utilisateur malintentionné peut en profiter pour lancer des attaques sans que

l'on puisse remonter jusqu'à lui.

Nous pensons néanmoins que les avantages l'emportent sur les inconvénients, voire même que ces derniers peuvent, pourquoi pas, provoquer des réflexes salutaires. Par exemple la possibilité même de faire une attaque de manière anonyme ne devrait-elle pas, si elle est connue de tous, inciter à prendre de vraies mesures de sécurité informatique au lieu de se réfugier, comme le font certains, oubliant qu'une loi ne protège que de ceux qui la respectent, ce qui est rarement le cas des pirates informatiques, derrière des « personne ne va attaquer car c'est interdit » ?

Nous ne reviendrons pas ici sur les améliorations envisageables pour les solutions que nous avons proposées elles-mêmes car nous avons déjà abordé ces points dans les conclusions respectives de nos deux parties (chapitres 8 page 73 et 15 page 115).

Ces deux propositions se situent dans la gamme des « services aux applications » : *True Nyms* permet le transport de données et notre solution pour DNS la résolution des noms. Toutefois, *True Nyms* fournit une protection contre un observateur tiers ; pour que la vie privée soit protégée, encore faut-il :

- soit que la partie à l'autre bout de la communication soit digne de confiance (ce qui restreint les applications) ;
- soit que l'*application* qui utilise les services offerts ne divulgue pas d'informations sensibles.

C'est sur ce dernier point que nous pensons qu'il faut insister : la plupart des applications actuelles ne prennent pas la vie privée en compte.

Sur le plan technique, il faudrait donc s'attaquer à la protection de la vie privée par les niveaux supérieurs :

- en fournissant des moyens pour « nettoyer » ce que transmettent les applications existantes, à la manière de ce que font *filterproxy* [51] ou *privoxy* [45] pour les navigateurs Web. La tâche est difficile et à traiter au cas par cas. Pour de nombreuses applications le problème sous-jacent est sans doute souvent indécidable et il n'est donc pas possible d'assurer une protection absolue de la vie privée de manière transparente ;
- mais aussi en s'assurant du respect de la vie privée lors du développement de nouvelles applications et de nouveaux protocoles. Là aussi, la tâche n'est pas toujours aisée.

Pour la plupart des applications, des protocoles qui assurent une meilleure protection de la vie privée existent, mais ne sont que peu ou pas utilisés. On peut citer par exemple le *e-cash* de Chaum qui permettrait de payer de manière anonyme sur l'Internet.

Cette non-utilisation est en partie due à la lucrativité du *profiling* face à une faible demande de la part du grand public. Il nous semble toutefois que la situation évolue et qu'un retournement brutal serait même possible : les tollés des utilisateurs de Facebook — une population qui n'est pas a priori des plus sensibles aux problèmes de vie privée — lors de l'apparition des *News feeds* personnels ou de *Beacon* [1] nous semblent être un indice révélateur.

[1] Facebook est un site Web de « réseau social ».
Les *News Feeds* sont une fonctionnalité du site qui permet d'être informé immédiatement des « actions » de vos amis sans consulter spécifiquement leur page. Introduite en septembre 2007 et non désactivable, elle a provoqué la colère de nombreux utilisateurs jusqu'à ce que Facebook

Si les mécanismes de protection de la vie privée doivent devenir courants, ils arriveront sans doute par les quelques domaines où la nécessité de la protection fait pratiquement l'unanimité.

Parmi les milieux les plus sensibilisés aux problèmes de confidentialité et de vie privée figure le secteur bancaire. *True Nyms* a été présenté à différents acteurs de la place financière luxembourgeoise. Ceux-ci se sont montrés fort intéressés, bien que conscients du côté « dangereux » de ce type d'application et ont posé la question de la faisabilité d'une version fournissant une protection similaire mais où l'anonymat serait levable sur requête judiciaire. Nous revenons sur ce sujet dans l'annexe B.

cède aux pressions et permette de la désactiver.
Beacon étend l'idée aux actions sur le Web, en dehors de Facebook : si vous achetez un objet sur un site partenaire de Facebook, vos amis en sont informés : l'idée est qu'ils ont a priori les mêmes goûts que vous et peuvent donc être intéressés (publicité ciblée).

A

Le protocole *True Nyms* : couches et format des paquets

Le but de ce chapitre est de donner quelques informations sur des détails concrets du protocole *True Nyms* décrit, implémenté et testé dans les chapitres 6 et 7. La description qui est donnée ici reste sommaire et n'est en aucun cas exhaustive ; une description technique complète du protocole sera l'objet d'un document ultérieur et séparé. En particulier, si nous décrivons le format des paquets, nous ne décrivons pas les différents états. Ceux-ci peuvent toutefois être déduits des informations données dans les chapitres de la première partie.

Dans la version testée, les paquets ont une taille de 1024 octets et sont émis toutes les 100 millisecondes.

Le protocole est constitué d'une couche transport (cf. A.1), sur laquelle passent les données. Les données sont partagées en deux types : des données proprement dites, qui sont celles de l'utilisateur, et des données de contrôle. Ces dernières respectent un protocole de niveau supérieur, destiné à contrôler les connexions (cf. A.3).

Notons toutefois que si chaque connexion dispose d'une session de contrôle, seule la communication de bout en bout dispose d'une session de données utilisateur.

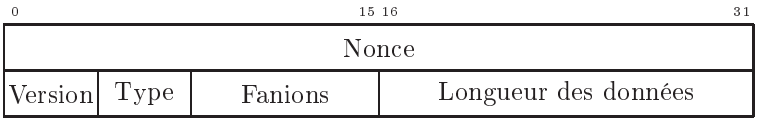
Nota Bene :
- une fois une connexion établie (par le protocole de contrôle, de niveau supérieur, cf. A.3), l'intégralité des paquets définis ici est chiffrée en mode Bi-IGE ;
- ce protocole se situe a priori au-dessus d'UDP(/TCP), mais peut facilement être adapté pour fonctionner directement au-dessus d'IP.

Dans les particularités du protocole, on notera qu'un paquet incompréhen-

sible est considéré comme étant à transmettre au nœud suivant. Un nœud donné ne sait pas s'il s'agit d'un paquet leurre, aléatoire, ou d'un paquet chiffré qui sera compris par un nœud en aval.

A.1 Couche transport

A.1.1 En-tête



Version

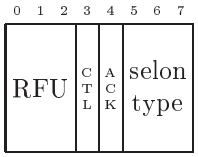
Le champ version vaut zéro (0) dans la version du protocole décrite dans ce document. *Il s'agit d'une version expérimentale.*

Type

Si `Type` vaut 0, il s'agit d'un paquet en mode datagramme (une sorte de pseudo-UDP, cf. section A.1.2) ;
Si `Type` vaut 1, il s'agit d'un paquet en mode flux (un pseudo-TCP, cf. section A.1.3).

Fanions

Les fanions peuvent être vus comme des valeurs booléennes. 1 correspond alors à « vrai » et 0 à « faux ».



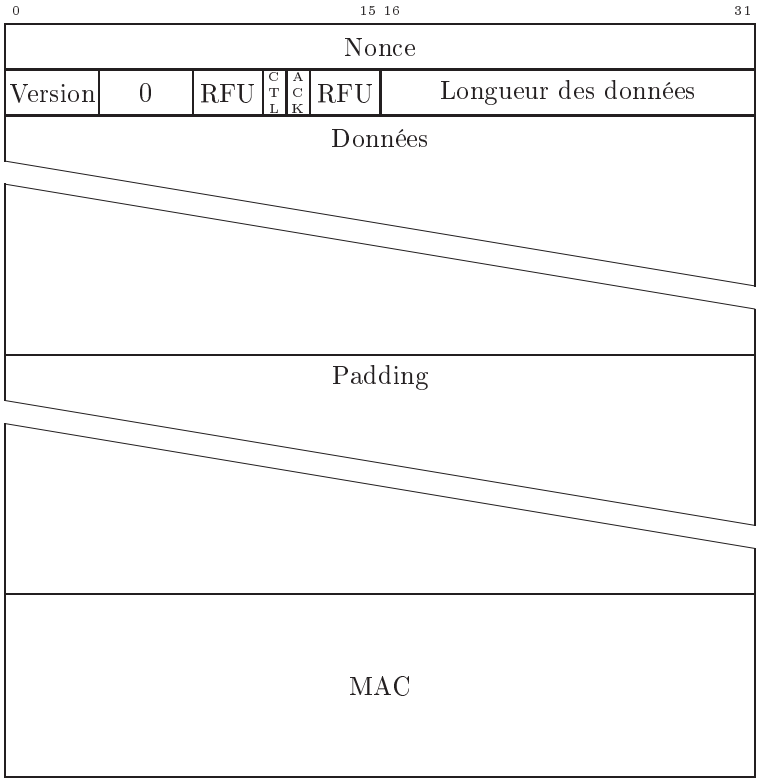
Le fanion `CTL` indique s'il s'agit d'un message de contrôle de la connexion (cf. A.3). S'il n'est pas positionné, il s'agit d'une session de données (cf. A.2).

Longueur des données

Ce champ indique la taille des données, *i.e.*, le nombre d'octets entre la fin du champ `longueur des données` et le début du padding.

A.1.2 Paquet non-acquiescé (≈ UDP)

Le protocole comporte des paquets dont la transmission n'est pas garantie aux couches supérieures. Cela fournit une sorte d'équivalent d'UDP.

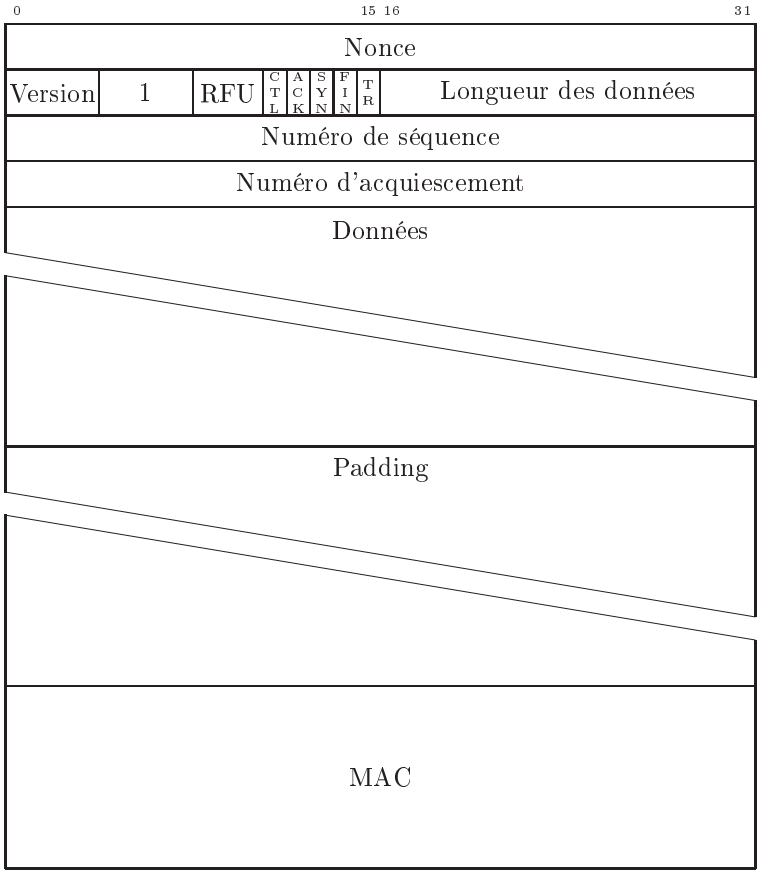


A.1.3 Paquet acquiescé (≈ TCP)

Ce mode fournit un service fiable, les retransmissions étant gérées de manière assez proche de ce qui se fait avec TCP, les différences venant des contraintes imposées par le lissage de trafic : il n'est par exemple pas possible d'envoyer autant de paquets `ACK` que l'on veut.

Étant donné que ce protocole se situe au-dessus de la couche d'anonymat, il n'est pas utile / possible d'intégrer des mécanismes de gestion de flux à ce niveau.

De même, il ne nous a pas semblé utile de gérer plusieurs sessions, une communication non-observable correspondant à une session pour la couche applicative. Il y a néanmoins deux pseudo-sessions différentes, l'une destinée aux données à transmettre à la couche de niveau supérieur, l'autre destinée aux messages de contrôle (respectivement fanion `CTL` à 0 et 1).



NB : en cas de perte d'un paquet et retransmission, le nonce (et donc la somme de contrôle) change.

Le rôle des fanions `ACK`, `SYN` et `FIN` est proche de celui des fanions de même nom dans TCP. Contrairement à ce qui se passe dans TCP, l'aquiescement se fait sur les paquets et non sur les données.

Le fanion `TR` a un rôle particulier. Normalement, le chiffrement seul est utilisé pour savoir si un paquet est destiné à / provient d'un nœud. Cependant, dans certains cas (par exemple avant la négociation d'une clef de session lors de l'établissement d'une connexion), il faut pouvoir distinguer le nœud i du nœud $i+1$ d'une route. Si le nœud i reçoit un paquet qu'il peut lire mais avec le fanion `TR` positionné à 1, il met ce fanion à 0 et le transmet au nœud suivant. Les choses se passent de manière inverse dans l'autre sens, et Alice sait qu'un paquet qu'elle reçoit, compréhensible après déchiffrement avec la clef du nœud i mais avec ce fanion positionné vient en fait du nœud $i+1$.

A.2 Session données

Les données sont transportées au-dessus des paquets décrits précédemment, avec le fanion `CTL` à 0. La partie du paquet de longueur `longueur des données`

après la fin de l'en-tête contient des données utilisateur.

Elles peuvent être transmises

1. de manière fiable, dans ce cas elles sont considérées comme formant un flux. Il y a retransmission si nécessaire, et réassemblage, puis elles sont passées à la couche application ;
2. de manière non-fiable. Dans ce cas, chaque paquet est considéré comme indépendant et est tranmis à la couche applicative dès son arrivée (pas de garantie d'ordre).

A.3 Session de contrôle

Les paquets de contrôle, utilisés pour la gestion des communications, sont transmis avec le fanion `CTL` positionné à 1. Ils sont tous envoyés en mode fiable, à quelques exceptions près.

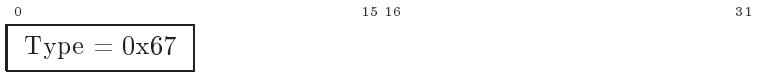
La transmission se fait comme pour les données, mais le contenu est alors destiné à l'implémentation qui le reçoit, et non à la couche supérieure.

Ce contenu est lui même structuré en messages. Voici les principaux messages qui peuvent être transmis dans une session de contrôle :

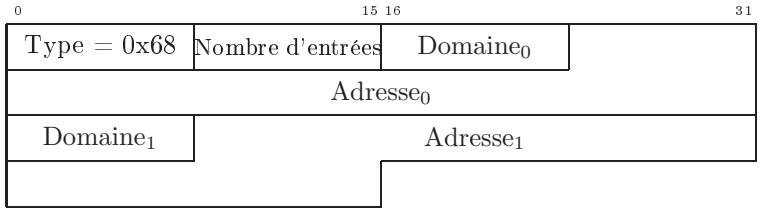
1. `LIST` (0x67) et LISTREPLY (0x68)

   Ces requêtes sont utilisées pour la gestion du réseau pair à pair : un nœud peut demander à un autre les adresses de membres du réseau.

   – requête :

   – réponse :

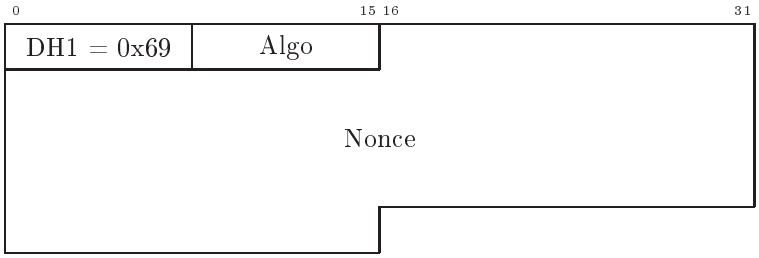
   `Domaine`$_i$ vaut 0 pour IPv4 (l'adresse est alors codée sur 32 bits) et 1 pour IPv6 (adresse sur 128 bits).

2. `REGISTER` (0x65) et `REGISTERREPLY` (0x66)

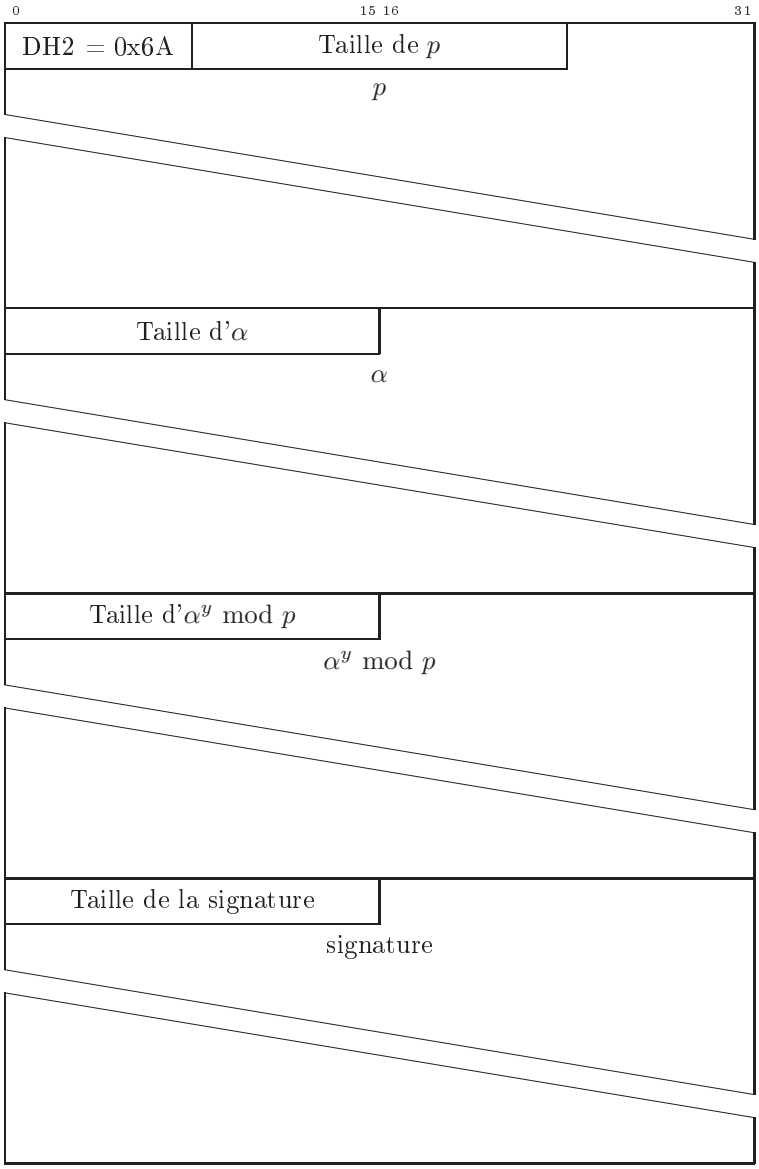
   Ces requêtes ont deux rôles :
   – elles permettent à un nœud de signaler son existence à un autre ;
   – elles servent à l'établissement d'arbres couvrants au-dessus du réseau, utilisés pour la transmission de messages d'établissement de communications (cf. [8]).

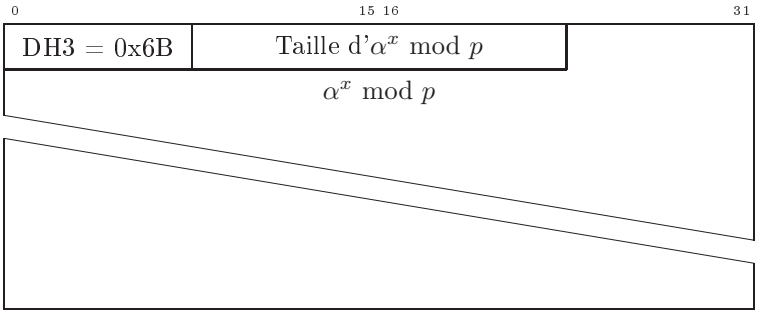
3. `DH1`, `DH2` et `DH3`

   La famille des messages DH sert à la négociation d'une clef de session pour une connexion. Les trois messages `DH1`, `DH2` et `DH3` correspondent aux étapes décrites en 3.1.3.



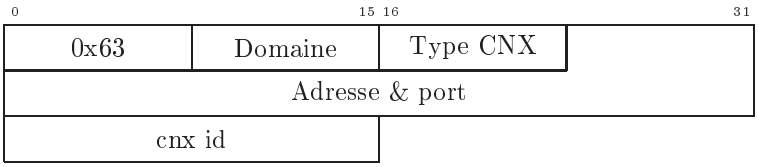
NB : Le paramètre *Algo* permet de choisir l'algorithme de chiffrement symétrique qui sera utilisé ensuite. Il influe donc sur la taille de la clef négociée.





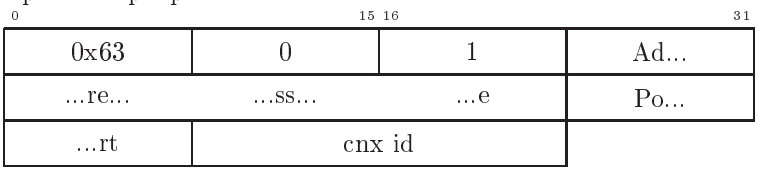
4. `CONNECT`

   Ce type de message est utilisé pour demander à un nœud de transmettre les paquets qu'il reçoit vers un autre nœud (première phase de l'établissement d'une connexion).

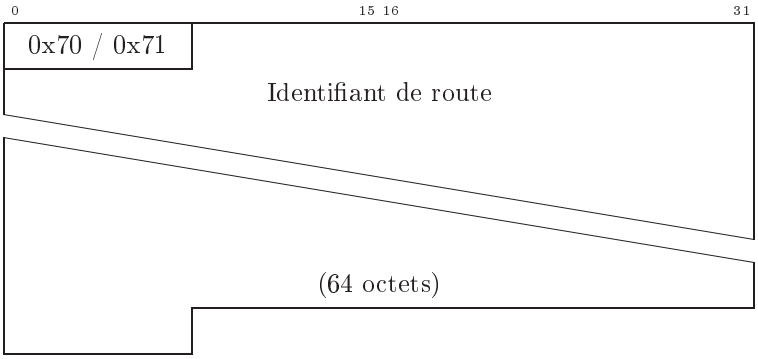


   `Domaine` vaut 0 pour IPv4 (l'adresse est alors codée sur 32 bits) et 1 pour IPv6 (adresse sur 128 bits). `Type CNX` vaut 0 pour une « orientation flux » et 1 pour une « orientation paquet » (correspondance avec respectivement `SOCK_STREAM` et `SOCK_DGRAM` dans les options des sockets).

   Ainsi par exemple pour une connection établie sur une socket TCP IPv4 :



5. `MERGE` (0x70) et `MERGE_ACCEPT` (0x71)
   Ces messages servent respectivement à demander de fusionner avec une autre route et à accepter la fusion avec une autre route.



6. `RDV`
   Ces messages servent à se connecter à un rendez-vous (cf. [8]).

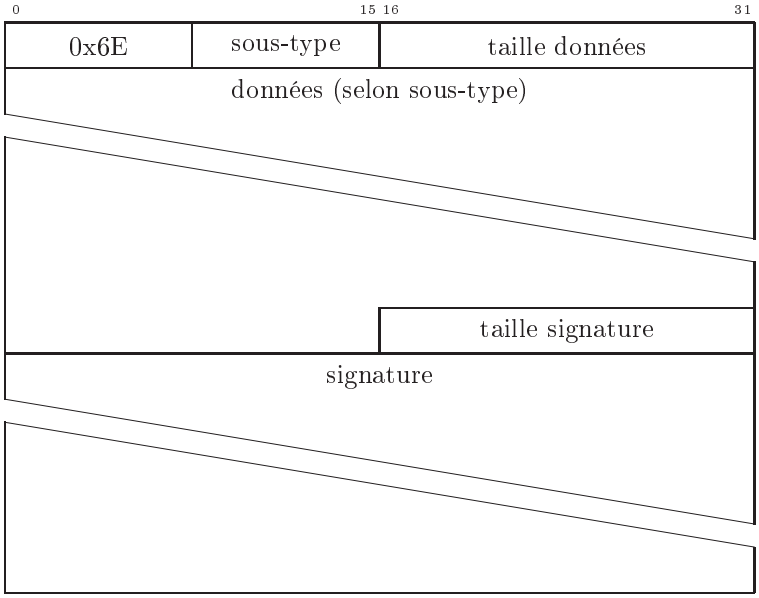
7. `ETB`

   Ces messages servent à établir un rendez-vous (cf. [8]).

8. `INFO`

   Ces messages sont utilisés pour signaler qu'une opération s'est déroulée correctement, ou au contraire qu'une erreur s'est produite.

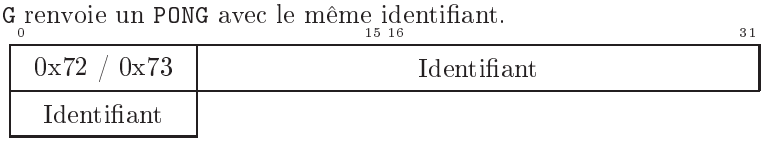
   Plusieurs *sous-types* existent, en particulier les sous-types génériques `OK` (une opération s'est déroulée correctement) et `ERROR` (une opération s'est déroulée incorrectement). Des sous-types spécialisés existent également, par exemple pour les opérations de fusion de routes (pour les communications utilisant plusieurs routes), ou pour des erreurs spécifiques (client sur un NAT, route interrompue, etc.).



   Notons que ces messages sont signés s'ils sont émis avant la négociation d'une clef de session.

9. `PING` (0x72) et `PONG` (0x73)

   Ces messages sont utilisés pour mesurer la latence. Un nœud qui reçoit un `PING` renvoie un `PONG` avec le même identifiant.



Messages de contrôle en mode non fiable

Notons que certains paquets de contrôle sont envoyés de manière non-fiable.

Le fanion `ACK` n'a de sens que dans le cas où `CTL` vaut 1 : il s'agit alors d'un paquet « freeack » : ces paquets sont émis dans deux cas :

- le récepteur n'envoie pas de données dans l'autre sens, mais il doit tout de même acquiescer les paquets qu'il reçoit ;

– le récepteur a des « trous » dans la séquence reçue. Le paquet « freeack » comporte alors le numéro du dernier paquet de la séquence ne contenant pas de trous, et une liste des paquets suivants reçus. On peut faire un parallèle avec l'option SACK de TCP.

Il faut noter que ces paquets ne sont pas réellement des messages de contrôle, puisqu'ils sont destinés en réalité à la couche transport.

Les messages `PING` et `PONG`, qui permettent de mesurer la latence, sont également envoyés en mode non fiable.

B

True Nyms : adaptation pour le « monde réel »

True Nyms, tel que décrit dans ce qui précède est totalement pair-à-pair et distribué. Il est par conséquent, très difficile à contrôler.

Pour des raisons légales et politiques, le diffuser tel quel est donc difficile. C'est pourquoi, dans ce chapitre, nous décrivons quelques modifications destinées à permettre la traçabilité du système, sur demande judiciaire par exemple.

Ces modifications centralisent en partie le système, le rendant à la fois moins élégant et moins robuste, mais également plus acceptable pour les autorités.

Ces modifications sont le fruit d'une réflexion engagée au sein d'un groupe de travail, constitué d'informaticiens et de juristes spécialisés plus particulièrement dans les problèmes de sécurité de la place financière luxembourgeoise.

Les sections qui suivent sont issues d'un document intitulé *le chaînon manquant du secret bancaire* [11] et présentent de manière non-technique les problèmes des protocoles de sécurité actuels, notre version « incontrôlable », et une évolution centralisée permettant un contrôle sur requête judiciaire.

B.1 Contexte et objectif

Les protocoles « sécurisés » actuels (par exemple SSL ou IPsec) ne protègent pas tous les aspects des communications sur l'Internet.

En particulier, ils permettent à un observateur :

- de connaître l'identité des parties qui communiquent ;
- dans certaines conditions spécifiques de connaître le contenu (en partie du moins) de la communication. Par exemple, si un utilisateur télécharge via `https` des fichiers d'un site, il est parfois possible à l'observateur de déterminer quels sont ces fichiers en dépit du chiffrement, en se basant (entre autres choses) sur la quantité de données transférées.

Si ce n'est pas toujours gênant — le numéro de carte bancaire reste dissimulé lors des achats en ligne — c'est néanmoins un frein au développement d'applications où la préservation de la sphère privée est essentielle, qu'elles soient médicales, bancaires ou juridiques.

Notre but est de remédier à cela, d'une manière particulièrement résistante aux diverses attaques, sans pour autant pénaliser les performances.

B.2 Anonymat version peer-to-peer (APP)

À compter de juin 2008, nous aurons implémenté et testé le système d'anonymat version peer-to-peer (APP) tel que décrit dans la figure B.1.

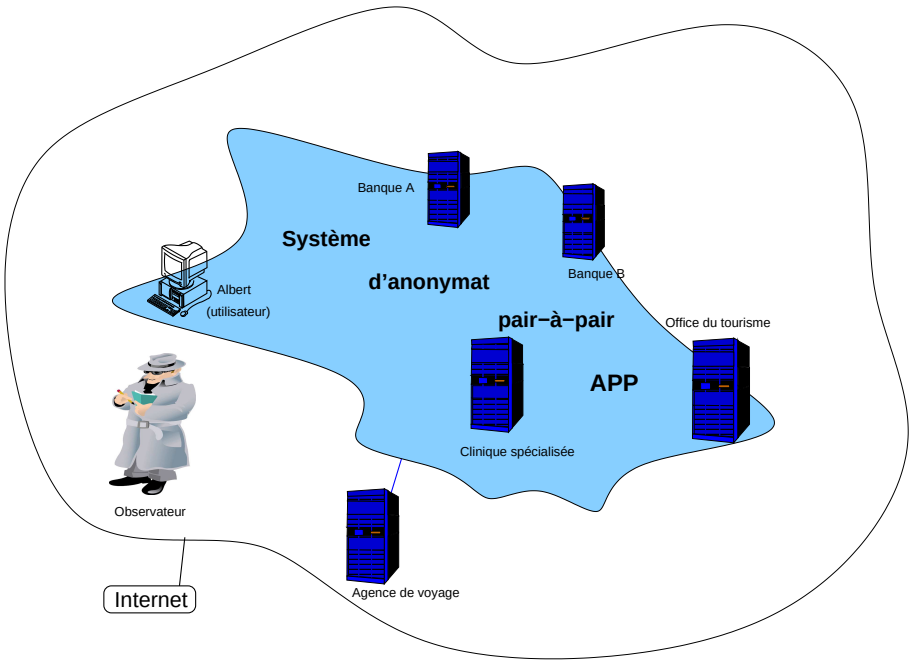


FIG. B.1 – L'observateur ne peut pas voir ce que font les membres du système APP. Il peut tout au plus savoir que des membres (il ignore lesquels) se connectent à des serveurs extérieurs à APP.

Dans APP, chaque participant est un noeud du réseau : il suffit d'installer notre logiciel `APP`. L'installation du logiciel `APP` est aussi simple que l'installation d'Acrobat Reader, pour citer un exemple. Un participant peut être le serveur d'une banque, un ordinateur personnel, etc. Chaque noeud est anonyme et contribue à l'anonymat des autres.

Un observateur, que ce soit l'ingénieur système d'un ISP ou même Echelon, peut uniquement voir qu'un ordinateur est un noeud d'APP, et pas davantage. En particulier, il est incapable de connaître :

- le destinataire des communications émises par le noeud ;
- *quand* le noeud communique réellement ;

– et, bien entendu, le contenu des communications.

Et ceci même si l'observateur contrôle un ou plusieurs noeuds.

Le cas d'Albert : Albert est dentiste à Munich. Il a une liaison Internet ADSL. Albert a installé sur son PC le logiciel `APP`. Il veut contacter la Banque B située au Luxembourg, soit pour y créer un compte (dans ce cas, il n'est pas techniquement nécessaire qu'il soit déjà connu de la Banque B), soit parce qu'il y dispose déjà d'un compte.

- Si la Banque B ne fait pas partie du réseau APP, un observateur puissant (Echelon) saura seulement qu'un noeud du réseau APP souhaite contacter la Banque B. Mais rien d'autre, en particulier il ne saura pas lequel ;
- si la Banque B fait partie du réseau APP, le même observateur puissant n'aura même pas cette information.

Albert lance par exemple Internet Explorer et tape http ://www.BanqueB.lu ; il accède normalement au site web en question. Si Albert est un client de la banque, il peut s'identifier et s'authentifier auprès de celle-ci normalement : l'utilisation d'APP fournit l'anonymat par rapport à un observateur mais ne l'impose pas vis-à-vis du destinataire. L'utilisation d'APP est totalement transparente pour Albert.

Le cas du Joker : le Jocker est un psychopathe terroriste disposant d'un PC portable et d'un wifi puissant. Il désire d'une part blanchir de l'argent à l'aide de la Banque B' corrompue (située en dehors du Luxembourg) et d'autre part pirater le système de contrôle des missiles nucléaires (ICBM) russes pour les lancer sur Gotham City. Du point de vue technique, la procédure utilisée par le Joker avec la banque B' est similaire à celle utilisée par Albert ci-dessus. Il utilise le réseau APP pour attaquer les systèmes russes sans que l'on puisse l'identifier et / ou le localiser, par exemple en utilisant les noeuds chinois du réseau APP.

Les limites et dangers d'APP sont les suivants :

- il est impossible de « tracer » les communications, même sur commission rogatoire ;
- il est impossible d'en restreindre l'utilisation : des noeuds peuvent être créés et intégrés à APP de manière indécelable et non contrôlée.

Il est parfaitement concevable de créer un virus qui installe APP sur un très grand nombre de machines. Cela a deux conséquences : d'une part cela augmente encore la robustesse et donc l'« intraçabilité » du système ; d'autre part, cela permet de communiquer anonymement en passant par les machines ainsi compromises à l'insu de leurs propriétaires légitimes.

Par conséquent, le système APP que nous avons développé peut parfaitement être utilisé en toute impunité par les Cavaliers de l'Infocalypse : terroristes, pédophiles, trafiquants de drogue, blanchisseurs d'argent.

Pour ces raisons, il nous paraît souhaitable d'aller vers une version plus contrôlable, telle que décrite dans la section qui suit.

B.3 Anonymat version contrôlée (AC)

L'objectif du système d'anonymat version contrôlée (AC) est d'obtenir une situation telle que décrite dans la figure B.2, c'est à dire d'obtenir les mêmes services qu'avec APP et en outre la traçabilité sur demande judiciaire.

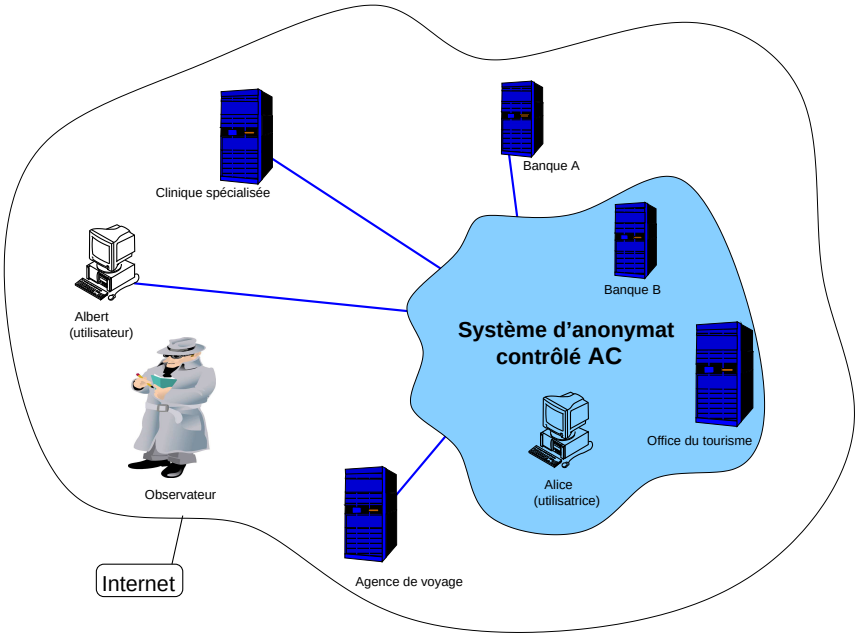


FIG. B.2 – L'observateur ne sait pas à quel(s) serveur(s) se connectent les utilisateurs.

Dans AC, les noeuds (*i.e.*, les machines qui participent à l'obtention de l'anonymat) sont mis en œuvre par une même entité : appelons-la « Titi[1] ». Les clients et serveurs ne sont pas des noeuds, même si l'anonymat peut s'étendre jusqu'à eux. Par exemple, sur la figure B.2 :

- la banque A n'est pas un noeud et l'observateur peut savoir que des usagers du système AC s'y connectent et quand ils s'y connectent. Par contre, il ne peut pas savoir *qui* s'y connecte.
- la banque B n'est pas non plus un noeud. Par contre elle est intégrée dans le système AC : l'observateur ne sait même pas quand un usager s'y connecte, et *a fortiori* quel est cet usager.

Il est aussi envisageable d'intégrer un usager dans le réseau de la même manière que la banque B : c'est le cas d'Alice sur la figure B.2.

Du point de vue du déploiement, la situation est la suivante :

- les noeuds disposent d'un logiciel spécialisé `AC-noeud`, proche du logiciel APP ;
- les « serveurs intégrés » (Banque B) installent un logiciel `AC-serveur`, qui permet uniquement de recevoir des connexions avec le protocole utilisé par le logiciel `AC-noeud` ;

[1] Le nom définitif sera déterminé ultérieurement.

- les « serveurs non-intégrés » (Banque A) n'ont aucun logiciel à installer ;
- les « clients intégrés » (Alice) installent un logiciel `AC-clientinteg`. Ce logiciel permet d'accéder à AC d'une manière qui empêche toute analyse des communications ;
- les « clients non-intégrés » (Albert) installent également un logiciel `AC-client-noninteg`. Ce logiciel, qui permet d'accéder à AC, ne fournit pas toutes les contre-mesures, mais est plus souple pour les connexions bas-débit.

Les informations contenues dans les logiciels `AC-serveur`, `AC-client-integ` et `AC-client-noninteg` ne sont pas suffisantes pour qu'un attaquant puisse reconstituer les fontionnalités d'`AC-noeud` ou d'`APP` par *reverse engineering*. Ces logiciels peuvent donc même être Open Source (les utilisateurs, clients et serveurs, peuvent ainsi auditer plus facilement les logiciels qu'ils installent ; cela permet d'améliorer la confiance).

Titi, pour s'assurer qu'il n'y a pas de noeuds exogènes introduits dans le système, met en place un système interne de certification de ces noeuds. Titi s'assure que les serveurs sont *trustés* à l'aide de certificats d'authentification « serveurs » spéciaux. Ceci concerne aussi bien les serveurs intégrés que non-intégrés. Titi **authentifie** les clients (par exemple via un certificat ou un jeu login / mot de passe). Cette liste de clients autorisés et le moyen de les authentifier lui sont communiqués par les serveurs, intégrés ou non, qui peuvent être les seuls à connaître l'**identité** réelle des clients.

Un observateur du type « ingénieur système d'un ISP » n'obtient pas plus d'information que dans le cas APP. Cependant, toutes les communications d'un client ne passent pas par AC et seules ces dernières (*i.e.*, celles à destination des serveurs accessibles par AC) sont protégées (avec APP, toutes les communications du client passaient par le système d'anonymat et étaient protégées). Un observateur global (Echelon) a connaissance de l'existence de communications entre le système AC et les serveurs non-intégrés : il peut mesurer leur durée et la quantité de données transmises, mais non déterminer le client qui a initié la communication (que celui-ci soit intégré ou non).

En particulier, l'un comme l'autre sont incapables de connaître :
- le destinataire des communications émises par un client ;
- *quand* un client intégré communique réellement ;
- et, bien entendu, le contenu des communications.

Le cas d'Albert : Albert a installé sur son PC le logiciel `AC-client-noninteg`. Il veut contacter la Banque A (respectivement B) située au Luxembourg, parce qu'il y dispose déjà d'un compte. La banque lui a fourni un moyen de s'authentifier auprès du système AC.
- La Banque A ne fait pas partie du réseau AC, un observateur puissant (Echelon) aura une faible probabilité de déterminer qu'Albert se connecte à la Banque A ;
- la Banque B fait partie du réseau AC, le même observateur puissant saura seulement qu'Albert établit une communication avec un serveur intégré au système AC mais ne pourra pas déterminer lequel.

Albert doit s'authentifier auprès du réseau AC. Il démarre ensuite, par exemple, Internet Explorer et tape http ://www.BanqueA.lu ; il accède normalement au site web en question. L'utilisation d'AC est alors totalement transparente pour Albert.

Le cas d'Alice : Alice est la femme d'Albert. Elle a installé sur son PC le logiciel `AC-client-integ`. Elle veut contacter la Banque A (respectivement B) située au Luxembourg, parce qu'elle y dispose déjà d'un compte. La banque lui a fourni un moyen de s'authentifier auprès du système AC.

- La Banque A ne fait pas partie du réseau AC, un observateur puissant (Echelon) saura seulement qu'un noeud du réseau AC souhaite contacter la Banque A. Mais rien d'autre, en particulier il ne saura pas lequel ;
- la Banque B fait partie du réseau AC, le même observateur puissant n'aura même pas cette information.

L'utilisation du système est similaire à celle d'Albert.

Le cas du Joker : le Joker désire d'une part blanchir de l'argent à l'aide de la Banque B' corrompue (située en dehors du Luxembourg) et d'autre part pirater le système de contrôle des missiles nucléaires (ICBM) russes pour les lancer sur Gotham City. Le Joker ne peut pas utiliser le système directement car il ne dispose pas de moyen pour s'authentifier. Pour contourner ce problème, le Joker peut tenter :

- soit d'obtenir des moyens d'authentification propres, par exemple en ouvrant un compte sous une fausse identité auprès de la banque A ou de la clinique spécialisée.
- soit s'emparer des moyens d'authentification d'Albert ou Alice.

Il s'authentifie donc auprès du réseau AC, mais ne peut accéder à la banque B', car celle-ci n'est pas un serveur autorisé du réseau AC. Déçu, il tente alors d'attaquer les sites russes. Ceux-ci sont des serveurs légitimes du réseau AC, et il peut donc s'y connecter. Les administrateurs des systèmes attaqués s'aperçoivent de l'attaque et portent plainte ; une demande judiciaire d'information est transmise à la justice luxembourgeoise qui décide d'y répondre et la transmet donc à Titi. Titi consulte ses journaux, ce qui permet de lier les données que le Jocker a utilisées pour s'authentifier à ces attaques, qui sont transmises à la justice luxembourgeoise.

Bibliographie

[1] Ross Anderson. *Security Engineering*. Wiley Computer Publishing. John Wiley & Sons, Inc, 2001.

[2] Roy Arends, Rob Austein, Matt Larson, Dan Massey, and Scott Rose. DNS Security Introduction and Requirements. RFC 4033, mars 2005. Disponible à l'adresse suivante : `ftp://ftp.rfc-editor.org/in-notes/rfc4033.txt`.

[3] Roy Arends, Rob Austein, Matt Larson, Dan Massey, and Scott Rose. Protocol Modifications for the DNS Security Extensions. RFC 4035, mars 2005. Disponible via ftp : `ftp://ftp.rfc-editor.org/in-notes/rfc4035.txt`.

[4] Roy Arends, Rob Austein, Matt Larson, Dan Massey, and Scott Rose. Resource Records for the DNS Security Extensions. RFC 4034, mars 2005. Disponible via ftp : `ftp://ftp.rfc-editor.org/in-notes/rfc4034.txt`.

[5] Olivier Aumage. Correspondance privée. Échange d'emails sur le thème d'Ibis, PM2, Java, la philosophie Unix et les avantages et inconvénients de la programmation orientée objet, mars 2003.

[6] Kevin Bauer, Damon McCoy, Dirk Grunwald, Tadayoshi Kohno, and Douglas Sicker. Low-Resource Routing Attacks Against Tor. In *Proceedings of the Workshop on Privacy in the Electronic Society (WPES 2007)*, Washington, DC, USA, octobre 2007. `http://systems.cs.colorado.edu/~bauerk/papers/wpes25-bauer.pdf`.

[7] Mihir Bellare, A. Boldyreva, Lars Knudsen, and C. Namprempre. On-Line Ciphers and the Hash-CBC Construction. In Joe Kilian, editor, *Advances in Cryptology – Proceedings of Crypto'2001*, number 2139 in LNCS, pages 292–309. Springer-Verlag, 2001.

[8] Nicolas Bernard. Non-observabilité des communications interactives. Master's thesis, Université Grenoble 1 – Joseph Fourier, juin 2004.

[9] Nicolas Bernard. Utilisation de l'implémentation de DTLS fournie par OpenSSL. Technical report, INRIA, to appears.

[10] Nicolas Bernard, Yves Denneulin, and Sébastien Varrette. Sécurité Unix. In Franck Leprévost, Touradj Ebrahimi, and Bertrand Warusfel, editors, *Cryptographie et sécurité des systèmes et réseaux*. Hermes, 2006.

[11] Nicolas Bernard and Franck Leprévost. Le chaînon manquant du secret bancaire. Document de travail, diffusion restreinte, janvier 2008.

[12] Nicolas Bernard, Franck Leprévost, and Michael Pöhst. Jacobians of genus 2 curves with a rational point of order 11. *Journal of Experimental Mathematics*, to appears.

[13] Daniel Julius Bernstein. How does DNS works ?, date inconnue. Disponible à l'adresse `http://cr.yp.to/djbdns/intro-dns.html` (fait partie de la documentation de DJBDNS).

[14] Burton Howard Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7) :422–426, 1970.

[15] Thomas Bocek. DDNS, distributed DNS, a P2P software that aims to build a decentralized, distributed DNS. site Web / programme, 2003. Pour plus d'informations : `http://distributeddns.sourceforge.net/`.

[16] Dan Boneh and Matthew Franklin. Identity-Based Encryption from the Weil Pairing. In Joe Kilian, editor, *Advances in Cryptology – Proceedings of Crypto'2001*, number 2139 in LNCS, pages 213–229. Elsevier, 2001.

[17] David Brin. *The Transparent Society – Will Technology Force Us to Choose Between Privacy and Freedom ?* Perseus Books, 1998.

[18] David Chaum. The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. *Journal of Cryptology*, 1(1) :65–75, 1988.

[19] David L. Chaum. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Communications of the ACM*, 24(2) :84–88, février 1981.

[20] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet : A distributed anonymous information storage and retrieval system. In *Proceedings of Designing Privacy Enhancing Technologies : Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, juillet 2000.

[21] Collectif. TheOnionRouter/TorFAQ. site Web (wiki), consulté en février 2008. La page en question se trouve à cette adresse : `https://wiki.torproject.org/noreply/TheOnionRouter/TorFAQ`.

[22] Benjamin Constant. De la liberté des anciens comparée à celle des modernes. Discours prononcé à l'Athénée royal de Paris, 1819. Disponible sur `http://www.panarchy.org/constant/liberte.1819.html`.

[23] Lance Cottrell. Mixmaster & Remailer Attacks. Disponible à cette adresse : `http://www.obscura.com/loki/remailer/remailer-essay.html`.

[24] Russ Cox, Athicha Muthitacharoen, and Robert T. Morris. Serving DNS using a Peer-to-Peer Lookup Service. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, mars 2002.

[25] Nathalie Dagorn and Nicolas Bernard. Web hacking. In *Proceedings of the Fourth IADIS International Conference on the World Wide Web and the Internet (Internet/www 2005)*. IADIS, octobre 2005.

[26] Nathalie Dagorn and Nicolas Bernard. Security and privacy in the digital economy : the case of electronic commerce. In *Proceedings of the IADIS e-Society International Conference (Dublin, 13-16 July 2006)*. IADIS, juillet 2006.

[27] Nathalie Dagorn and Nicolas Bernard. Security and privacy in the digital economy : How to provide anonymity in electronic commerce. In *Proceedings of the 6th IADIS e-Society International Conference (Algarve, April 9-12 2008)*. IADIS, avril 2008.

[28] Nathalie Dagorn, Nicolas Bernard, and Sébastien Varrette. Practical authentication in distributed environments. In *Proceedings of the First International Computer System and Information Technology Conference (ICSIT'05)*, juillet 2005.

[29] Wei Dai. A new way to do anonymity. Message sur la liste *cypherpunks*, 07 février 1995.

[30] Yves Denneulin, Nicolas Bernard, Pascal Bouvry, and Sébastien Varrette. Sécurité réseaux. In Franck Leprévost, Touradj Ebrahimi, and Bertrand Warusfel, editors, *Cryptographie et sécurité des systèmes et réseaux*. Hermes, 2006.

[31] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6) :644–654, novembre 1976.

[32] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor : The Second-Generation Onion Router. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[33] Michael J. Freedman and Robert Morris. Tarzan : A Peer-to-Peer Anonymizing Network Layer. In *Proceedings of the 9th ACM Conference on Computer and Communications Security* . ACM, 2002.

[34] The FreeS/WAN Team. logiciel/site Web, 1996-2003. Pour plus d'informations, voir `http://www.freeswan.org/`.

[35] James Frenkel, editor. *True Names and the opening of the cyberspace frontier*. Tor, 2001.

[36] John Gilmore. Privacy, Technology, and the Open Society. In *First Conference on Computers, Freedom, and Privacy*, mars 1991. Disponible en ligne : `http://www.toad.com/gnu/cfp.talk.txt`.

[37] Ken Guggenheim. Pentagon Abandons Terrorism Betting Plan. Dépèche Associated Press, juillet 2003.

[38] David H. Holtzman. *Privacy Lost : How Technology Is Endangering Your Privacy*. Jossey-Bass (Wiley), 2006.

[39] The Invisible Internet Project. *Introducing I2P*. `http://www.i2p2.de/`.

[40] Emmanuel Jeannot. Improving Middleware Parformance with AdOC: an Adaptive Online Compression Library for Data Transfer. In *Proceedings of the 19th International Parallel & Distributed Processing Symposium (IEEE IPDPS'05)*, avril 2005. L'article est disponible sur le site de son auteur : `http://www.loria.fr/~ejeannot/publications/ipdps05.pdf` ; voir aussi `http://www.loria.fr/~ejeannot/adoc/adoc.html` (bibliothèque AdOC).

[41] Emmanuel Jeannot, Björn Knutsson, and Mats Björkman. Adaptive Online Data Compression. In *Proceedings of the IEEE High Performance Distributed Computing (HPDC'11) Conference*, juillet 2002. L'article est disponible ici : `http://www.loria.fr/~ejeannot/publications/hpdc11.pdf`.

[42] Antoine Joux, Gwenaëlle Martinet, and Frédéric Valette. Blockwise-Adaptive Attackers, Revisiting the (In)Security of Some Provably Secure Encryption Modes : CBC, GEM, IACBC. In M. Yung, editor, *Advances in Cryptology – CRYPTO 2002*, number 2442 in LNCS, pages 17–30. Springer-Verlag, 2002.

[43] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. Dns performance and the effectiveness of caching. *IEEE/ACM Transactions on Networking*, 10(5) :589–603, 2002.

[44] David Kahn. *The Codebreakers*. Scribner, troisième edition, 1996.

[45] Fabian Keil and David Schmidt *et al.* `privoxy`. Logiciel, 2001-2008. Disponible sur `http://www.privoxy.org`.

[46] Amit Klein. BIND 9 DNS Cache Poisoning, mars-juin 2007. Disponible à l'adresse `http://www.trusteer.com/docs/bind9dns.html`.

[47] Lars Knudsen. Block Chaining Modes of Operation. Technical report, Department of Informatics, University of Bergen, 2000. Disponible en ligne : `http://www.ii.uib.no/publikasjoner/texrap/ps/2000-207.ps`.

[48] Donald Ervin Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, seconde edition, 1998.

[49] Donald Ervin Knuth. All Questions Answered. *Notices of the* AMS, 49(3) :318–324, mars 2002.

[50] Peter Ludlow, editor. *Crypto Anarchy, Cyberstates, and Pirate Utopias*. Digital Communications. MIT Press, 2001.

[51] Robert McElrath. `filterproxy`. Logiciel, 2002. N'est plus maintenu. Voir `http://filterproxy.sourceforge.net/`.

[52] 2003 Wiretap Report. Technical report, Administrative Office of the United States Courts, Leonidas Ralph Mecham (dir), mai 2004. Consultable à cette adresse : `http://www.uscourts.gov/wiretap03/contents.html`.

[53] Paul V. Mockapetris. DOMAIN NAMES - CONCEPTS AND FACILITIES. RFC, novembre 1987. Disponible à l'adresse suivante : `ftp://ftp.rfc-editor.org/in-notes/rfc1034.txt`.

[54] Paul V. Mockapetris. DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION. RFC, novembre 1987. Consultable et téléchargeable à l'adresse suivante : `ftp://ftp.rfc-editor.org/in-notes/rfc1035.txt`.

[55] Ulf Moeller, Lance Cottrell, Peter Palfrader, and Len Sassaman. Mixmaster Protocol Version 2. Internet-draft, mai 2004.

[56] Benoît Mouthon. Non-observabilité des communications interactives sur Internet. Master's thesis, Université Grenoble 1 – Joseph Fourier, septembre 2007.

[57] The OpenSSL Project. Site web. `http://www.openssl.org`.

[58] The Openswan Project. logiciel/site Web, depuis 2003. Pour plus d'informations, voir `http://www.openswan.org/`.

[59] Eric Steven Raymond. *The Art or Unix Programming*. Addison-Wesley, 2003. Des informations complémentaires et une copie du livre sont disponibles en ligne : `http://www.catb.org/~esr/writings/taoup/`.

[60] Michael G. Reed, Paul F. Syverson, and David M. Goldschlag. Anonymous connections and Onion Routing. *IEEE Journal on Selected Areas in Communications*, 16(4) :482–494, mai 1998.

[61] Michael K. Reiter and Aviel D. Rubin. Crowds: anonymity for Web transactions. *ACM Transactions on Information and System Security*, 1(1) :66–92, 1998.

[62] Eric Rescorla. `ssldump`, 1999–... `http://www.rtfm.com/ssldump/`.

[63] Eric Rescorla. *SSL and TLS – Designing and Building Secure Systems*. Addison-Wesley, 2001. Voir `http://www.rtfm.com/sslbook/` pour plus de détails.

[64] RSA Laboratories. *PKCS #1 v2.1: RSA Cryptography Standard.* RSA Security, 2002. Le document ne donne pas d'auteurs explicites, mais la RFC 3447, écrite par Jakob Jonsson et Burt Kaliski et datée de février 2003, a exactement le même contenu. Ses auteurs étaient à l'époque employés par les RSA Laboratories ; ils sont donc probablement les auteurs de ce document.

[65] Bruce Schneier. *Cryptographie appliquée.* International Thompson Publishing, deuxième edition, 1996.

[66] Bruce Schneier. The War on the Unexpected. In *Crypto-Gram.* novembre 2007. `http://www.schneier.com/crypto-gram-0711.html`.

[67] Axel-Tobias Schreiner. Object-oriented Programming with ANSI-C, 1993. Disponible à l'adresse suivante : `http://www.cs.rit.edu/~ats/books/`.

[68] Adi Shamir. Identity-Based Cryptosystems and Signature Schemes. In George Robert Blakley and David Chaum, editors, *Advances in Cryptology – Proceedings of Crypto'84*, number 196 in LNCS, pages 47–51. Springer, 1984.

[69] Daniel J. Solove. "I've Got Nothing to Hide" and Other Misunderstanding of Privacy. *San Diego Law Review*, 44, 2007. Disponible à l'adresse suivante : `http://ssrn.com/abstract=998565`.

[70] W. Richard Stevens. *TCP / IP Illustrated*, volume 1. Addison-Wesley, 1994.

[71] W. Richard Stevens. *TCP / IP Illustrated*, volume 3. Addison-Wesley, 1996.

[72] The strongSwan Project. logiciel/site Web, depuis 2003. Pour plus d'informations, voir `http://www.strongswan.org/`.

[73] Sébastien Varrette and Nicolas Bernard. *Programmation Avancée en C avec exercices et corrigés.* Hermes Science Publication, février 2007. Des informations complémentaires sont disponibles :`http://c.lafraze.net/`.

[74] John Viega, Matt Messier, and Pravir Chandra. *Network Security with OpenSSL.* O'Reilly and Associates, 2002.

[75] Jean-Marc Vincent. Évaluation de performances. Cours de Master 2 recherche Systèmes et Logiciels, Université Grenoble 1 – Joseph Fourier et Institut National Polytechnique de Grenoble, 2004.

[76] Vernor Vinge. *True Names.* In Frenkel [35], 1981.

[77] Paul Vixie. Extension Mechanisms for DNS (EDNS0). RFC 2671, août 1999. Disponible à l'adresse `http://www.rfc-editor.org/rfc/rfc2671.txt`.

[78] Paul Vixie, Gerry Sneeringer, and Mark Schleifer. Events of 21-Oct-2002, octobre 2002. `http://d.root-servers.org/october21.txt`.

[79] David A. Wheeler. *Secure Programming for Linux and Unix HOWTO*, 3.010 edition, mars 2003. `http://www.dwheeler.com/secure-programs`.

[80] Gary R. Wright and W. Richard Stevens. *TCP / IP Illustrated*, volume 2. Addison-Wesley, 1995.

[81] Michal Zalewski. *p0f.* 2000–... `http://lcamtuf.coredump.cx/p0f.shtml`.

[82] Michal Zalewski. *Silence on the Wire: a Field Guide to Passive Reconnaissance and Indirect Attacks.* No Starch Press, 2005.

[83] Michal Zalewski. *`fl0p` – passive flow fingerprinter*. 2006. La version de développement est téléchargeable sur le site de l'auteur : `http://lcamtuf.coredump.cx/soft/fl0p-devel.tgz`.