Capture à grande échelle de trafic eDonkey

Frédéric Aidouni, Matthieu Latapy, Clémence Magnien

Capture à grande échelle de trafic eDonkey

Frédéric Aidouni[1], Matthieu Latapy[1] et Clémence Magnien[1]

[1]*LIP6, CNRS et Université Pierre et Marie Curie - 4 place Jussieu, 75005 Paris, France - nom.prenom@lip6.fr*

Nous présentons une capture des requêtes gérées par un serveur *eDonkey* pendant une semaine, totalisant plus de 33 millions d'utilisateurs (IP) et près de 88 millions de fichiers distincts. La capture et l'exploitation ultérieure d'une telle masse de données sont deux défis, auxquels nous apportons des réponses. Nous fournissons finalement un ensemble de données surpassant de plusieurs ordres de grandeur ceux précédemment disponibles.

Keywords: trafic, mesure, pair-à-pair, P2P, eDonkey, emule, eServer,

1 Introduction

Acquérir des informations sur les réseaux *pair-à-pair* déployés à grande échelle est une tâche complexe mais cruciale pour appréhender leur fonctionnement et mettre au point de nouveaux protocoles [HKF+06, SGG02, SG07]. De ce point de vue, le système *eDonkey* est particulièrement intéressant car c'est l'un des plus utilisé. De plus il est structuré autour de serveurs gérant les recherches de fichiers et de fournisseurs ; il est donc possible, en se positionnant sur un de ces serveurs, d'observer ces requêtes.

Nous présentons ici la capture du trafic d'un des serveurs *eDonkey* les plus actifs, réalisée en continu pendant une semaine en 2007. Les conditions de la capture, la masse de données collectées, leur décodage, anonymisation et prétraitement (pour en faciliter l'exploitation ultérieure) sont autant de défis. Nous décrivons les solutions que nous avons proposées et mises en œuvre pour y répondre au cours de cette capture, qui dépasse de plusieurs ordres de grandeur celles précédemment disponibles.

2 *eDonkey* en bref

eDonkey est un système *pair-à-pair* semi-distribué d'échange de fichiers basé sur un ensemble de serveurs qui assurent une fonction d'annuaire, référençant les fichiers et leurs diffuseurs. Ils répondent essentiellement à des requêtes de recherche de fichiers par métadonnées (nom, taille ou type de contenu par exemple) et à des requêtes de recherche de diffuseurs (appelés sources).

Les fichiers sont indexés au moyen d'une clé de hachage MD4, le *fileID*, et ont au moins deux métadonnées : nom et taille. Les diffuseurs de fichiers sont réferencés par un *clientID* qui est leur adresse IP lorsqu'ils sont joignables par une connexion directe, et un nombre de 24 bits sinon.

Il existe une documentation officieuse du protocole [KB05], et les codes sources de plusieurs clients sont disponibles ; nous renvoyons à ces références pour plus de détails.

Les messages se repartissent en quatre familles : administratifs, comme une demande au serveur de la liste des serveurs qu'il connaît ; recherche de fichiers par métadonnées, la réponse étant alors une liste de *fileID*, avec leur nom, leur taille et diverses métadonnées ; recherches de sources par *fileID*, auxquelles le serveur répond par une liste de sources pour le fichier ayant ce *fileID* ; alimentation de l'annuaire *i.e.* l'annonce par un client de la liste des fichiers qu'il partage.

3 Capture du trafic

Avant même de pouvoir capturer le trafic d'un serveur *eDonkey* il est nécessaire d'obtenir l'accord de son administrateur. La garantie d'un impact négligeable sur le système, de l'utilisation des données collectée à des fins de recherche, de leur mise à disposition de la communauté scientifique et de leur anonymisation conformément à la loi française, nous ont permis d'obtenir un tel accord.

Le code source du serveur *eDonkey* n'étant pas disponible, la solution idéale de le modifier pour y inclure l'enregistrement du trafic qu'il gère est impossible. Il faut donc déployer un système de capture du trafic brut au niveau IP, qu'il s'agira ensuite de décoder en messages *eDonkey*.

Le serveur étant localisé dans un *datacenter* sans possibilité d'accès physique pour y installer une carte d'acquisition, nous avons dû concevoir une solution logicielle. Nous avons utilisé *libpcap* (`http://tcpdump.org`), une librairie classique de capture du capture trafic *ethernet*. Le flux produit était ensuite redirigé vers une machine de décodage (Section 4), anonymisation (Section 5) et stockage.

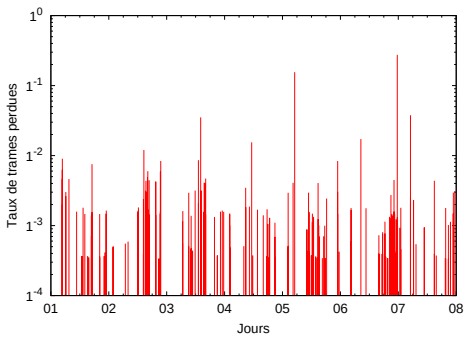

FIG. 1: Taux de paquets *ethernet* perdus par seconde sur la durée de la capture, soit 1501 trames perdues sur 949 876 855 acquises effectivement (0.016%)

Cette configuration induit néanmoins des pertes de paquets qui sont provoquées par la durée de la capture et le débit. En effet, dans notre contexte, *libpcap* fonctionne grâce à un *buffer* de trames *ethernet* alimenté par le noyau. En cas de pointe de trafic, le noyau, qui est prioritaire sur les applications, peut remplir ce *buffer* plus vite qu'il n'est vidé. Ainsi, ponctuellement, le noyau ne l'alimente plus car il est plein. Dans cette situation le noyau compte le nombre de trames non stockées chaque seconde, comme représenté Figure 1. Ces pertes rendent la reconstruction des flux TCP excessivement difficile[†]. Dans la suite nous ignorons par conséquent le trafic TCP et considérons uniquement UDP (qui représente la moitié du trafic global).

4 De UDP à *eDonkey*

Au niveau UDP/IP, le logiciel de décodage vérifie les trames, défragmente le trafic et anonymise les adresses IP de la couche *ethernet* (*cf* Section 5). Sur 949 876 855 trames *ethernet* capturées, 2981 sont des fragments et 169 sont mal formées.

Le décodage manuel des messages tient une place importante durant la phase de conception d'un décodeur tel que celui dont nous avons besoin. En effet ce trafic est issu d'une multitude de logiciels clients différents (eux mêmes déclinés en plusieurs versions), peu fiables, ne respectant pas complètement le protocole, etc. De plus, les codes sources des clients récents sont particulièrement opaques, et le protocole recèle des optimisations de codage qui complexifient la compréhension du protocole. Au final, la réalisation d'un système de décodage se révèle beaucoup plus complexe que l'écriture d'un client.

Ainsi, notre décodeur fonctionne en deux passes : une validation structurelle des messages (fondée sur leur longueur prévue par exemple), puis, en cas de succès, une tentative de décodage effectif. Ainsi, sur les 949 873 704 messages traités, seulement 0.68% n'ont pas été décodé, dont 78% étaient structurellement incorrects et ne pouvaient donc pas être décodés.

Soulignons également que nous avons normalisé la représentation des nombres, éliminé les astuces de codage mises en œuvre par le protocole et encapsulé les parties les plus complexes pour que les utilisateurs de nos données puissent aisément les ignorer. Les messages *eDonkey* finalement obtenus sont donc dans un format plus uniforme que les originaux.

5 Anonymisation et formatage

L'anonymisation de traces internet est en soi un sujet délicat [AP07]. Nous avons choisi ici une approche extrêmement prudente puisque nous anonymisons les *clientID*, les *fileID*, les chaînes de caractères utilisées dans les recherches et les noms de fichiers, et les tailles de fichiers. De plus, l'horodatage des données commence à 0 afin de ne pas fournir la date exacte de capture. Chacune de ces données à anonymiser a ses spécificités dont nous avons dû tenir compte.

[†] Le serveur reçoit en moyenne environ 5000 paquets SYN par minute, ce qui représente plusieurs centaines de milliers de flux à suivre en temps réel.

Les tailles de fichiers sont stockées en kilo-octets (alors qu'elle sont originalement en octets) ; cette réduction de la précision nous a paru suffisante pour cette information peu sensible. Les chaînes de caractères des recherches par mots-clés et les noms de fichiers sont codés par leur MD5 qui garantit une anonymisation suffisante tout en gardant les données cohérentes.

Coder les *clientID* par une clé de hachage est trop hasardeux. Il serait en effet facile, connaissant le type de la fonction de hachage, de retrouver le *clientID* d'origine (il suffirait d'appliquer la fonction aux 2^{32} *clientID* possibles). De même, les diverses technique de brouillage (inversion de bits) n'apportent pas une anonymisation suffisante pour cette information extrêmement sensible. Nous avons donc opté pour un codage des *clientID* par leur ordre d'apparition : le premier *clientID* rencontré est anonymisé en 0, le second en 1, etc. Cette technique offre deux avantages : elle fournit une anonymisation extrêmement sûre, et simplifie l'exploitation ultérieure des données puisque les *clientID* sont anonymisés par des entier de 0 à N-1 (si N est le nombre de *clientID* distincts).

Pour réaliser ce codage, il faut être capable de savoir si un *clientID* a déja été anonymisé, et donc de gérer un ensemble contenant ceux déja rencontrés. Chaque message contenant *au moins* un *clientID*, un nombre astronomique de tests d'appartenance est nécessaire (plusieurs milliards), ainsi qu'un grand nombre d'ajouts (des millions). Les structures de données classiques (tables de hachage ou arbres, typiquement) sont inutilisables dans ce contexte, car trop lentes oú trop coûteuses en espace. Nous avons donc tiré parti du fait qu'au plus 2^{32} *clientID* distincts peuvent être rencontrés, et avons représenté l'ensemble par un tableau d'autant d'entiers d'anonymisation, indexé par *clientID*. Cette structure nous permet de tester la présence d'un élément et d'en ajouter un par un accès mémoire direct, au prix des 16 giga-octets de mémoire occupés par le tableau.

Dans le souci de simplifier l'exploitation ultérieure des données, nous avons choisi d'anonymiser également les *fileID* par leur ordre d'apparition. Ici aussi, le nombre de tests d'appartenance est énorme, ainsi que, dans une moindre mesure, le nombre d'ajouts. A nouveau, les structures ensemblistes classiques ne peuvent répondre à nos besoins. De plus, étant donnée la taille des *fileID* (128 bits), la solution mise en œuvre pour les *clientID* est exclue.

Une solution pourrait être l'utilisation d'un tableau dans lequel les *fileID* rencontrés seraient stockés de façon triée, avec leur anonymisation. La structure de tableau a l'avantage d'être relativement compacte. Le fait que le tableau soit trié permet de faire les tests d'appartenance très rapidement (recherche dichotomique). Par contre, les ajouts (triés) sont coûteux puisqu'ils induisent d'importantes réorganisations du tableau. Celui-ci étant amené à contenir des dizaines de millions d'éléments, leur coût est prohibitif.

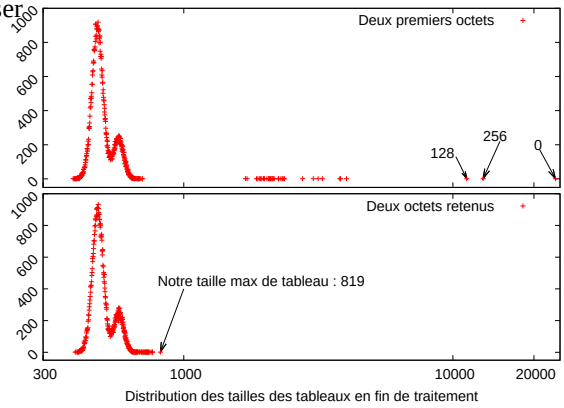

FIG. 2: Distribution des tailles de tableaux d'anonymisation des *fileID*, on observe la présence de tableaux anormalement grands, surtout lorsqu'on indexe par les deux premiers octets (le tableau d'index 0 a dans ce cas une taille de 24024 éléments)

Nous pouvons toutefois contourner ce problème en remarquant que les éléments de l'ensemble sont en principe des clés de hachage, et sont donc *a priori* répartis uniformément dans l'ensemble des possibles. Ainsi, si on divise le tableau en un certain nombre de tableaux plus petits indépendants, chacun responsable d'une portion égale de l'espace des possibles, ils auront en principe des tailles proches et plus petites d'autant. L'ajout d'éléments se fera alors beaucoup plus rapidement, au prix d'un coût en espace plus grand.

Dans notre cas, diviser l'espace des possibles en 65536 tableaux indexés par les deux premiers octets des *fileID* peut sembler un compromis raisonnable : puisqu'on rencontre dans notre trace 88 millions de *fileID* distincts, chacun de ces tableaux devrait contenir en fin de traitement environ 1500 éléments ; le coût d'une insertion triée dans un tel tableau est raisonnable.

La mise en œuvre de cette approche révèle toutefois un fait surprenant : les *tableaux d'anonymisation* d'index 0 ou 256 ont une taille disproportionnée, *i.e.* une majorité de *fileID* commencent par 0 ou 256, comme illustré Figure 2. Ceci indique la présence massive de *fileID* générés artificielement [UMJ+06], qui pénalisent fortement nos calculs. Pour contourner cette difficulté, nous avons retenu deux octets dispersés dans le *fileID* pour indexer notre tableau. La Figure 2 montre que cette méthode, sans éliminer totalement l'hétérogénéité des taille des tableaux, la gomme fortement, suffisament pour notre utilisation.

Finalement la chaîne de traitements que nous venons de décrire, quoique coûteuse en espace, nous permet de passer du trafic UDP au stockage anonymisé sur disque à raison d'une journée de trafic traitée à l'heure. Ceci garantit la nécessaire qualité temps-réel.

6 Conclusion

Nous avons décrit les solutions que nous avons apportées aux défis soulevés par la capture du trafic d'un serveur *eDonkey* de grande taille : réalisation d'une sonde réseau, décodage du protocole et anonymisation en temps réel. Nos choix durant l'anonymisation fournissent une aide précieuse à l'exploitation ultérieure des données. Par exemple, calculer le nombre d'occurences de chaque *fileID* vus au cours de la mesure devient trivial avec notre codage alors que sans ce prétraitement, même cette tâche *a priori* simple est difficile.

Nous produisons finalement des fichiers XML décrivant la quasi-totalité des échanges UDP entre un serveur *eDonkey* et ses clients‡, et les fournissons à la communauté§.

Ces données, qui surpassent largement en taille, durée et qualité celles précédemment disponibles, ouvrent la voie à de nombreuses analyses, comme illustré sur un exemple simple Figure 3. Notre perspective esentielle est de travailler en ce sens, notamment : étude des communautés, profilage d'utilisateurs, étude des phénomènes de diffusion, étude des échanges et diverses analyses statistiques.

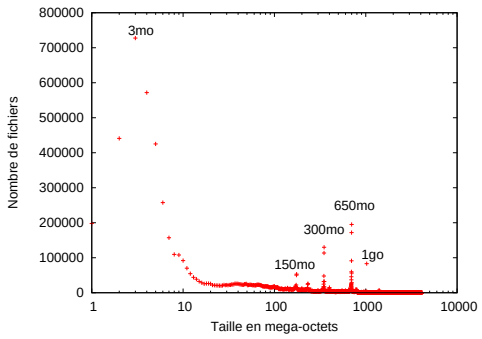

FIG. 3: Distribution des tailles de fichiers proposés par le serveur en méga-octets. On peut observer les tailles multiples de la taille d'un CD, ainsi que le pic sur les fichiers probablement musicaux de quelques méga-octets

Références

[AP07] M. Allman and V. Paxson. Issues and etiquette concerning use of shared measurement data. In *IMC*, 2007.

[HKF+06] S. B. Handurukande, A.-M. Kermarrec, F. Le Fessant, L. Massoulié, and S. Patarin. Peer sharing behaviour in the edonkey network, and implications for the design of server-less file sharing systems. In *EuroSys '06*, pages 359–371, New York, NY, USA, 2006. ACM.

[KB05] Y. Kulbak and D. Bickson. The emule protocol specification, 2005.

[SG07] Walid Saddi and Fabrice Guillemin. Measurement based modeling of edonkey peer-to-peer file sharing system. In *International Teletraffic Congress*, pages 974–985, 2007.

[SGG02] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems, 2002.

[UMJ+06] Lee U., Choi M., Cho J., Sanadidi M. Y., and Gerla M. Understanding pollution dynamics in p2p file sharing. In *In Proceedings of the 5th International Workshop on Peer-to-Peer Systems (IPTPS'06)*, 2006.

‡ Nous planifions une nouvelle campagne de captures qui incluera cette fois TCP.

§ `http://www-rp.lip6.fr/~latapy/p2pdata/`