Ten weeks in the life of an eDonkey server

Frédéric Aidouni, Matthieu Latapy and Clémence Magnien
LIP6 – CNRS and University Pierre & Marie Curie
104 avenue du president Kennedy, 75016 Paris, France
Email: Firstname.Lastname@lip6.fr

***Abstract*—This paper presents a capture of the queries managed by an *eDonkey* server during almost 10 weeks, leading to the observation of almost 9 billion messages involving almost 90 million users and more than 275 million distinct files. Acquisition and management of such data raises several challenges, which we discuss as well as the solutions we developed. We obtain a very rich dataset, orders of magnitude larger than previously avalaible ones, which we provide for public use. We finally present basic analysis of the obtained data, which already gives evidence of non-trivial features.**

I. Introduction

Collecting live data on running *peer-to-peer* networks is an important task to grasp their fundamental properties and design new protocols [1], [2], [3], [4], [5]. To this end, *eDonkey* is appealing: it is one of the currently largest and most popular *peer-to-peer* systems. Moreover, as it is based on servers in charge of file and source searches, it is possible to capture the traffic of such a server to observe the queries it manages and the answers it provides.

Contribution and context.

We describe here a continuous capture of UDP/IP level traffic on an important *eDonkey* server during almost ten weeks, from which we extract the application-level queries processed by the server and the answers it gave. This leads to the observation of 8 867 052 380 *eDonkey* messages, involving 89 884 526 distinct *clientID* and 275 461 212 distinct *fileID*. We carefully anonymise and preprocess this data, in order to release it for public use and make it easier to analyse. Its huge size raises unusual and sometimes striking challenges (like for instance counting the number of distinct *fileID* observed), which we address.

The obtained data surpasses previously available ones regarding several key features: its wide time scale, the number of observed users and files, its rigorous measurement, encoding, and description, and/or the fact that it is released for public use. It also has the distinctive feature of dealing with user behaviors, rather than protocols and algorithms, or traffic analysis, *e.g.* [6], [7], [8], [9], [10]. To this regard, it is more related to previous measurement-based studies of peer behaviors in various systems, *e.g.* [11], [12], [13], [14], [15], [16], and should lead to more results of this kind.

As a passive measurement on a server, it is complementary of passive traffic measurements in the network [9], [10], [8], and client-side passive or active measurements [13], [14], [15], [16] previously conducted on *eDonkey*. Up to our knowledge, it is the first significant dataset on *eDonkey* exchanges released so far (though [17], [5] use similar but much smaller data), and it is the largest *peer-to-peer* trace ever released. Of course, it also has its own limitations (for instance, it does not contain any information on direct exchanges between clients).

II. Measurement

Since our goal was to observe *real-world* exchanges processed by an *eDonkey* server, we had to capture the traffic on an existing server (with the authorization of its administrator and within legal limits). In this context, it was crucial to avoid any significant overload on neither the server itself nor its administrator. Likewise, installing dedicated material (*e.g.* a DAG card) was impossible.

Moreover, it is of prime importance to ensure a high level of anonymisation of this kind of data. This anonymisation must be done in real-time during the capture. As IP addresses appear at both UDP/IP and *eDonkey*/application levels, this implies that the network traffic must be decoded to application-level traffic in real-time.

Finally, we want the released data to be as useful for the community as possible, and so we want to format it in a way that makes analysis easier. This plays an important role in our encoding strategy described in Section II-D, with a strong impact on data usability which we illustrate in Section III.

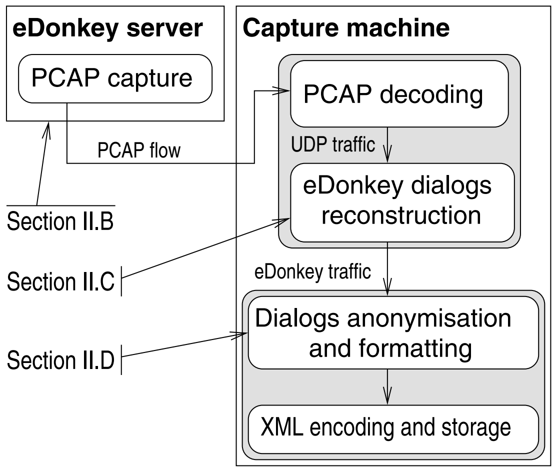

Fig. 1. **From PCAP raw traffic to XML representation**

In order to reach these goals, we set up a measurement procedure in three successive steps, as illustrated in Figure 1. First, we *capture* the network traffic of an *eDonkey* server using a dedicated program and send it to our capture machine

(Section II-B). Then this traffic is reconstructed at IP level and *decoded* into *eDonkey*-level traffic, *i.e.* queries and corresponding answers (Section II-C). Finally, these queries are *anonymised and formated* (Section II-D) before being stored as XML documents.

A. The eDonkey protocol briefly

eDonkey is a semi-distributed *peer-to-peer* file exchange system based on directory servers. These servers index files and users, and their main role is to answer to searches for files (based on metadata like filename, size or filetype for instance), and searches for providers (called *sources*) of given files.

Files are indexed using a MD4 hash code, the *fileID*, and are characterised by at least two metadata: name and size. Sources are identified by a *clientID*, which is their IP address if they are directly reachable or a 24 bits number otherwise.

eDonkey messages basically fit into four families: management (for instance queries asking a server for the list of other servers it is aware of); file searches based on metadata, and the server's answers consisting of a list of *fileID* with the corresponding names, sizes and other metadata; source searches based on *fileID*, and the server's answers consisting of a list of sources (providers) for the corresponding files; and announcements from clients which give to the server the list of files they provide.

An unofficial documentation of the protocol is available [18], as well as source code of clients; we do not give more details here and refer to this document for further information.

B. Traffic capture

Before starting any traffic capture, one has to obtain the agreement of a server administrator. The following guarantees made it possible to reach such an agreement: negligible impact of the capture on the system; use of collected data for scientific research; and high level of anonymisation (higher than requested by law).

The ideal solution would be to patch the server source code to add a traffic recording layer. However, as this source code is *not* open-source, this was impossible. We thus had to design a traffic capture system at the IP level, then decode this traffic into *eDonkey* messages.

The server is located in a *datacenter* to which we have no access. A dedicated traffic interception hardware installation was therefore impossible, and we had to build a software solution. To this end, we used *libpcap*[1], a standard network traffic capture library. We sent a copy of the traffic to a capture machine, in charge of decoding (Section II-C), anonymising (Section II-D) and storing.

This approach leads to packet losses during the capture, due to the duration of the capture and the network's bandwidth. Indeed, *libpcap* uses a buffer where the kernel stores captured packets. In case of traffic peaks, this buffer may be unsufficient and get full of packets, while some others still arrive. The kernel cannot store these new packets in the buffer, and some

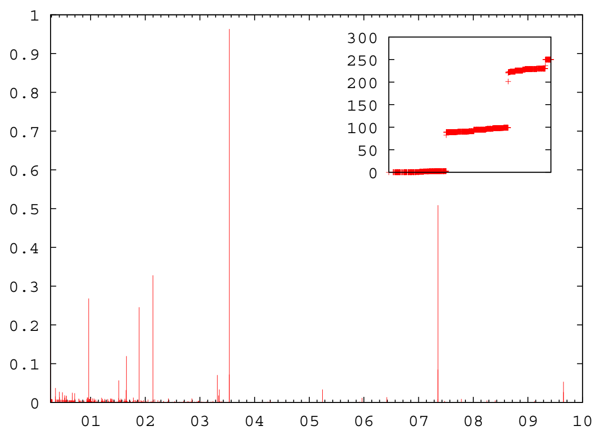

Fig. 2. Ethernet packet losses rate per second during the capture and cumulative losses in thousands of packets (inset). Horizontal axes are labelled by the number of weeks elapsed since the beginning of the measurement. By its end, 250 266 packets were lost and 31 555 295 781 were captured.

are thus lost. The number of lost packets is stored in a kernel structure, and thus we know the amount of losses that occured, see Figure 2. These losses, although very rare, make TCP flows reconstruction very difficult, as packets are missing inside flows[2]. In this paper, we therefore focus on UDP traffic only, which constitutes about half of the captured traffic. At the *eDonkey* level, the main difference between UDP and TCP is that only peers connected via TCP send the list of files they share; we will therefore not observe this information here.

C. From UDP to eDonkey

At UDP level, our decoding software checks packets and re-assembles the traffic. Among 14 124 818 158 UDP packets captured, 8 933 745 734 were received by the *emule* server (a large part of the traffic consisted of *Kadmelia* protocol), 34 652 are fragments and 11 235 476 lacked the *eDonkey* protocol header. This corresponds to 8 922 475 606 potential *eDonkey* messages, which are then decoded.

The captured traffic is generated by many poorly reliable clients of different kinds (and versions), with their own interpretation of the protocol. Moreover, their source codes are intricate, and the protocol embeds complex encoding optimisations. Finally, decoding the server traffic is much harder than programming a client, and requires an important work of manual decoding of the messages.

Our decoder operates in two steps: a structural validation of messages (based on their expected length or protocol semantics) then, if successful, an attempt at effective decoding. Among the 8 922 475 606 potential *eDonkey* messages, only 0.62% were not decoded by our system (78% of these undecoded messages were structurally incorrect, and thus not decodable) leading finally to 8 867 052 380 well-formed *eDonkey* messages.

[1]http://tcpdump.org

[2]Even without packet losses, TCP conversation reconstruction is not an easy task, as the server receives about 5000 SYN packets per minute.

D. Anonymisation and formating

Anonymisation of internet traces is a subtle issue in itself [19]. Since we want to provide the obtained data for public use, we need a very strong anonymisation scheme: *clientID*, *fileID*, search strings, filenames and filesizes must all be anonymised (each with a dedicated method, described below). In addition, timestamps are replaced by the time elapsed since the beginning of the capture to further limit the desanonymisation risks.

Filesizes are stored in kilo-bytes (originally they were in bytes); this precision reduction seems enough to protect this information, which raises no important privacy issue.

Anonymising *clientID* with a hash code is not satisfactory: if one knows the hash function, it is easy to find the original *clientID* by applying the function to the 2^{32} possible *clientID*. Shuffling strategies are not strong enough either for this very sensitive data. We therefore chose to encode *clientID* according to their order of appearance in the captured data: the first one is anonymised with the value 0, the second with 1 and so on. Although computationaly expensive (see below), this technique has two advantages: it ensures a very strong anonymisation level and it makes further use of the dataset much easier, as anonymised *clientID* are integers between 0 and N-1 (if there are N distinct *clientID*).

To perform this encoding, we must be able to recognise previously encountered (and anonymised) *clientID*. We must thus store throughout the capture the set of *clientID* already seen, with their anonymisation. As each message contains at least one *clientID*, an overwhelming number of searches (several billions) must be performed in this set, as well as millions of insertions. Classical data structures (like hashtables or trees) are unsatisfactory in this context: they are too slow and/or too space consuming. Instead, we used the fact that at most 2^{32} dictinct *clientID* exist: we used an array of 2^{32} integers (hence of total size 16 giga-bytes), and stored the anonymisation of each *clientID* in the *clientID*-th cell of this array. This has a high cost in central memory, but allowed us to anonymise *clientID* with a direct memory access operation only, hence very efficiently.

We also chose to anonymise the *fileID* by their order of appearance. Here again, the number of insertions and searches in the corresponding set is huge. As a consequence, classical set structures were not relevant in this case either. Moreover, because of the size of *fileID* (128 bits), we could not use the same solution as for *clientID*.

A possible solution could be to use a sorted array containing *fileID*, with their anonymisation key. Arrays are compact structures, and when sorted a dichotomic search is very fast. However, insertion has a prohibitive cost, due to the reorganisation it implies to keep the array sorted.

One may avoid this problem in a simple way, as *fileID* are hash codes: they are supposed to be uniformally distributed in their coding space. As a consequence, dividing the main array in equally-sized smaller ones, indexed by any part of the *fileID*, should reduce their size uniformally and thus significantly speed up element insertions.

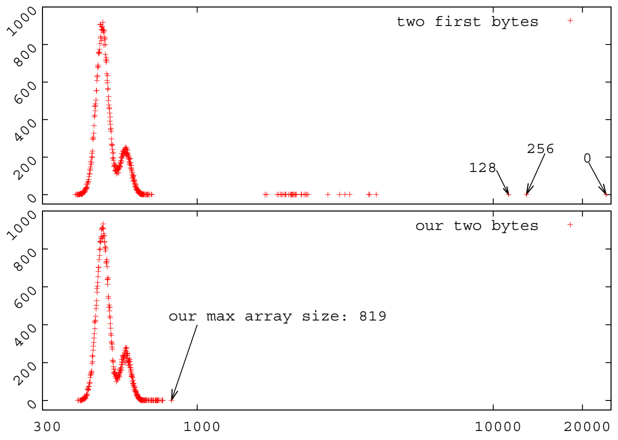

Fig. 3. **Size distribution of *fileID* anonymisation arrays after one week of capture.** One can observe abnormally large arrays when the arrays are indexed by the first two bytes (array 0 contains 24 024 elements in this case); using other bytes reduces this significantly.

In our particular situation, dividing the array size by a factor of 65 536 by using the two first bytes to index 65 536 arrays seems a good solution: as we encounter 88 million distinct *fileID* in our capture, each array length should be around 1500; sorted insertion in such arrays is reasonable.

However, implementing this strategy led to surprising results: anonymisation arrays 0 and 256 had very large sizes, see Figure 3. This shows that, in practice, a majority of *fileID* start with 0 or 256, and thus reveals the massive presence of forged *fileID* [20]. They induce the unbalanced sizes of our anonymisation arrays, which strongly hampers our computations.

We solved this problem by selecting two different bytes in the *fileID* to index our 65 536 arrays. Figure 3 shows that this approach does not perfectly remove the heterogeneity of array sizes, but it was sufficient for our application.

Search strings, filenames, and server descriptions are similarly encoded by their order of appearance. We re-used the *fileID* anonymisation scheme by first encoding these strings as MD5 hashes, and then replacing these hash codes by the wanted anonymisation.

Finally, the processing method we have described is rather space consuming as we fully used a *bi-dual Opteron* server loaded with 24Gb of RAM to compute the stream, but it is able to decode UDP traffic in real-time, while anonymizing, which is crucial in our context.

E. Final dataset

The final dataset we obtain consists in a series of 8 867 052 380 *eDonkey* messages (queries from clients and answers to these queries from the server) in XML format[3]. It contains very rich information on users at 89 884 526 distinct

[3]We chose XML as output format because it leads to easy-to-read and rigorously specified text files, and, once compressed, does not have a prohibitive space cost.

IP addresses dealing with 275 461 212 distinct *fileID*, while preserving the privacy of users.

This dataset is publicly available with its formal specification [4].

III. Basic analysis

We present in this section a few basic analysis. Notice however that these statistics are subject to measurement bias [21], and only reflect the content of our data; more careful analysis should be conducted to derive accurate conclusions on the underlying objects.

Their purpose is to highlight the fact that the grade of our anonymisation process does not prevents analysis as stated also in [14] or [5], and the richness of our dataset. As for instance, studies of community structures and evolution or content diffusion may be conducted as well. That dataset could also be used as a source of statistical insights for in-development *peer-to-peer* protocols.

Notice that our formating greatly simplifies such analysis: having *clientID* and *fileID* represented by contiguous intergers starting from 0 is central in making most computations tractable. Easier anonymisation schemes, like hashing of the *fileIDs* and *clientIDs*, would have produced a slightly less compact dataset, and, more importantly, would make analysis much harder.

A. File size

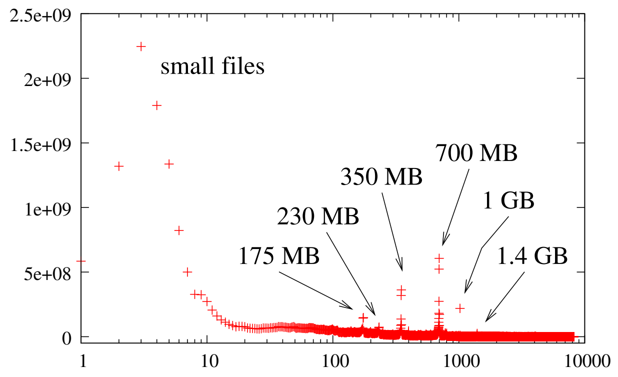

Fig. 4. File size distribution, *i.e.* for each encountered file size (horizontal axis) the number of files having this size.

We display in Figure 4 the distribution of the size of exchanged files, obtained from the answers of the server to some queries which indicate the size of found files. One observes many small files (probably music files), and clear peaks at 700 MB (typical size of a CD-ROM ISO images or *SVCD movie*), and at fractions (1/2, 1/3, 1/4) or multiples (2 $\times$) of this value. The peak at 1 GB may indicate that users split very large files (DVD images for instance) into 1 GB pieces.

This plot reveals the fact that, even though in principle files exchanged in P2P systems may have any size, their actual sizes are strongly related to the space capacity of classical exchange and storage supports.

[4]http://www-rp.lip6.fr/~latapy/tenweeks/

B. File popularity

We define the *popularity* of a file (identified by its *fileID*) as the number of distinct peers asking or providing this file at any time in our dataset (we do not consider files which are never provided). The distribution of the 11 760 816 popularities obtained this way (not represented here) is perfectly well fitted by a power-law on three decades, and has a heavy tail. It means that the popularities are very hererogeneous: although a very large number (more than 5 millions) of files have a popularity lower than or equal to 10, some (46) have a popularity larger than 50 000.

Let us first focus on these extremely popular files, the 46 ones with popularity larger than 50 000. We plot the evolution of their popularity during time in Figure 5. These plots clearly show that the popularity of these files evolves smoothly as soon as the file is first seen, and does not stop to grow. However, some files appear rather late during our measurement (up to more than three weeks after its beginning). This shows that new popular files appeared after the beginning of the measurement. Therefore, our dataset may be used to study how a new file becomes popular, which is a fundamental question.

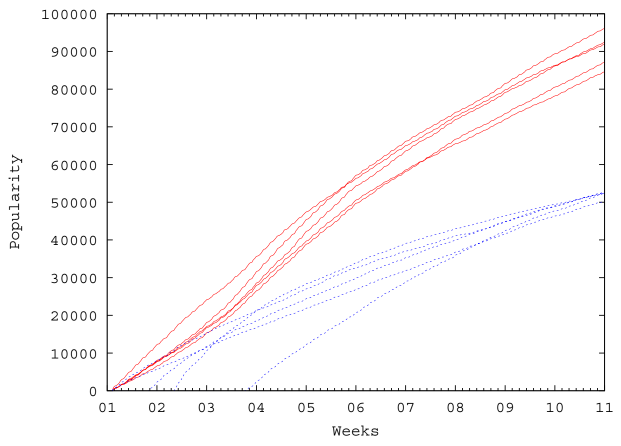

Fig. 5. Time-evolution of the popularity of some of the 46 extremely popular files: red solid lines for the five most popular; blue dashed lines for the five least popular.

Figure 5 also shows that very popular files are popular for a rather long time. As a consequence, and even though we may observe the appearance of very popular files in our dataset, we cannot observe the disappearence of such files.

In order to explore this, we selected the most popular files which we encountered neither during the first week of measurement nor during the last one. Only 26 such files have a popularity larger than 2 000, much lower than the maximal. As expected, the time evolution of their popularity strongly differs from the one of the extremely popular files, see Figure 6 for an illustration. In these cases, the appearance of the file is followed by a relatively short period during which many peers download it, and then the file is never encountered again. Notice that this does not mean that no user is interested in this file anymore, but maybe that no provider is present, which may be investigated further using our dataset.

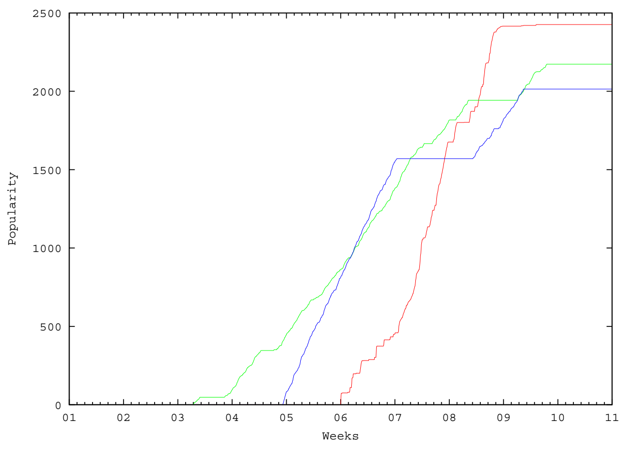

Fig. 6. Time-evolution of the popularity of some typical files among the most popular ones for which we can observe appearance and disappearance.

IV. Conclusion

This paper presents a capture of the queries managed by a live *eDonkey* server at a scale significantly larger than before, both in terms of duration, number of peers observed, and number of files observed. This dataset is available for public use with its formal specification [5] in an easy-to-use and rigorous format which significantly reduces the computational cost of its analysis. We present a few simple analysis which give evidences of the fact that our dataset contains much information on various phenomena of interest. It may also be used for simulation (trace replay) and modeling purposes.

This work may be extended by conducting measurements of TCP *eDonkey* traffic, and more generally by measuring the *eDonkey* activity using complementary methods (active measurements from clients, for instance). The measurement duration may also be extended even more, and likewise the traffic losses may be reduced.

From an analysis point of view, this work opens many directions for further research. For instance, it makes it possible to study and model user behaviors, communities of interests, how files spread among users, etc. Most of these directions were out of reach with previously available data, and they are crucial from both fundamental and applied points of view.

Acknowledgements. This work is supported in part by the European MAPAP SIP-2006-PP-221003 and French ANR MAPE projects.

References

[1] S. B. Handurukande, A.-M. Kermarrec, F. L. 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*. New York, NY, USA: ACM, 2006, pp. 359–371.

[2] S. Saroiu, P. Gummadi, and S. Gribble, "A measurement study of peer-to-peer file sharing systems," 2002. [Online]. Available: citeseer.ist.psu.edu/saroiu02measurement.html

[3] W. Saddi and F. Guillemin, "Measurement based modeling of edonkey peer-to-peer file sharing system," in *International Teletraffic Congress*, 2007, pp. 974–985.

[4] P. Gauron, P. Fraigniaud, and M. Latapy, "Combining the use of clustering and scale-free nature of user exchanges into a simple and efficient P2P system," in *Euro-Par*, 2005.

[5] J.-L. Guillaume, S. Le-Blond, and M. Latapy, "Clustering in P2P exchanges and consequences on performances," in *IPTPS*, 2005.

[6] W. Acosta and S. Chandra, "Trace driven analysis of the long term evolution of gnutella peer-to-peer traffic," in *PAM*, 2007.

[7] A. Legout, G. Urvoy-Keller, and P. Michiardi, "Rarest first and choke algorithms are enough," in *IMC*, 2006.

[8] W. Saddi and F. Guillemin, "Measurement based modeling of edonkey peer-to-peer file sharing system," in *ITC*, 2007.

[9] T. Karagiannis, A. Broido, M. Faloutsos, and K. Claffy, "Transport layer identification of p2p traffic," in *IMC*, 2004.

[10] K. Tutschku, "A measurement-based traffic profile of the edonkey filesharing service," in *PAM*, 2004.

[11] D. Hughes, J. Walkerdine, G. Coulson, and S. Gibson, "Peer-to-peer: Is deviant behavior the norm on p2p file-sharing networks?" *IEEE Distributed Systems Online*, vol. 7, no. 2, 2006.

[12] G. Neglia, G. Reina, H. Zhang, D. Towsley, A. Venkataramani, and J. Danaher, "Availability in bittorrent systems," in *INFOCOM*, 2007.

[13] M. Zghaibeh and K. Anagnostakis, "On the impact of p2p incentive mechanisms on user behavior," in *NetEcon+IBC*, 2007.

[14] S. Handurukande, A.-M. Kermarrec, F. L. 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*, 2006.

[15] F. L. Fessant, S. Handurukande, A.-M. Kermarrec, and L. Massoulié, "Clustering in peer-to-peer file sharing workloads," in *IPTPS*, 2004.

[16] O. Allali, M. Latapy, and C. Magnien, "Measurement of edonkey activity with distributed honeypots," in *Proceedings of HotP2P'09*, 2009.

[17] S. Le-Blond, M. Latapy, and J.-L. Guillaume, "Statistical analysis of a P2P query graph based on degrees and their time evolution," in *IWDC*, 2004.

[18] Y. Kulbak and D. Bickson, "The emule protocol specification," 2005. [Online]. Available: citeseer.ist.psu.edu/kulbak05emule.html

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

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

[21] D. Stutzbach, R. Rejaie, N. Duffield, S. Sen, and W. Willinger, "On unbiased sampling for unstructured peer-to-peer networks," in *IMC*, 2006.

[5] http://www-rp.lip6.fr/~latapy/tenweeks/