INFORMATION SOCIETY TECHNOLOGIES (IST)
PROGRAMME

Large Scale Monitoring of BroadBand Internet Infrastructure
Contract No. 004336

D4.4: Web-based Interactive Monitoring Application

Abstract:

In this document we present *appmon*, a passive monitoring application for per-application network traffic classification. *Appmon* uses deep packet inspection to accurately attribute traffic flows to the applications that generate them, and reports in real time the network traffic breakdown through a Web-based GUI.

Contractual Date of Delivery	31 December 2006
Actual Date of Delivery	22 December 2006
Deliverable Security Class	Public

The LOBSTER Consortium consists of:

FORTH	Coordinator	Greece
VU	Principal Contractor	The Netherlands
CESNET	Principal Contractor	Czech Republic
UNINETT	Principal Contractor	Norway
ENDACE	Principal Contractor	United Kingdom
ALCATEL	Principal Contractor	France
FORTHnet	Principal Contractor	Greece
TNO	Principal Contractor	The Netherlands
TERENA	Principal Contractor	The Netherlands

Contents

1 Introduction **7**

2 Design and Implementation **9**
2.1 Traffic Classification 9
2.2 Implementation 11
2.3 Graphical User Interface 12

3 Performance Evaluation **17**
3.1 Local Testbed 17
3.2 Monitoring Real Network Traffic 18

4 Deployment **21**

5 Summary **27**

A Appendix **29**
A.1 Tracker library 29
A.1.1 TRACK_FTP 29
A.1.2 TRACK_DC 29
A.1.3 TRACK_GNUTELLA 30
A.1.4 TRACK_EDONKEY 30
A.1.5 TRACK_TORRENT 30
A.1.6 TRACK_GRID_FTP 30
A.2 Extra Functions library 31
A.2.1 COOKING 31
A.2.2 REGEXP 31
A.2.3 TOP 31
A.2.4 EXPIRED_FLOWS 31

List of Figures

2.1 Appmon Web Interface. 13
2.2 Per-application bandwidth usage. 14
2.3 Top 10 incoming traffic IP addresses as presented by appmon. . . 15

3.1 *appmon* measurement environment. 18
3.2 *appmon* CPU Load Vs. Traffic Load for both NIC and DAG Interfaces using the local testbed descriped in Figure 3.1. 19
3.3 *appmon* performance on a sensor deployed on University of Crete for serveral days. Each measurement was taken every 10 seconds. 19

4.1 Lobster Applications Deployment. 22
4.2 Application deployment in Checz Republic Monitoring the connection with GEANT2. 23
4.3 Google Earth Application showing results from FORTH-ICS sensor. 23
4.4 Google Earth Application showing results from HellasGRID sensor. 24
4.5 *appmon* running on a sensor at National Technical University of Athens. Using a DAG network interface *appmon* manages to process traffic in Gbit speeds. 25
4.6 *appmon* running on a sensor at Aristotle University of Thessalloniki. Using a NIC card *appmon* can process traffic load of 500 Mbps. 26

Chapter 1

Introduction

One of the most frequent requests of network administrators is to identify the applications and hosts that generate the largest amount of network traffic. The emergence of peer-to-peer file sharing, multimedia streaming, and conferencing applications has resulted to a substantial increase in the traffic volume, since they transfer a large amount of data. However, monitoring the traffic generated from such applications is becoming increasingly difficult.

Traditionally, traffic attribution to the corresponding applications is performed using the statically assigned port numbers. Widely used network services, like the Web, Telnet, SSH, and many others, are associated with well-known port numbers which can be used for identifying the traffic related with each application. However, many major new applications, including popular, bandwidth-hungry file sharing applications and widely used video and voice conferencing applications, do not use well-known port numbers. Instead, they allocate and use dynamically negotiated ports. Furthermore, some applications masquerade their traffic using pervasive, firewall-friendly protocols, like HTTP, in order to bypass firewall restrictions and make the identification of their traffic harder. Indeed, several widely used applications like BitTorrent [9] and Skype [8] can be configured to operate through port 80, which is usually left open even in environments with strict firewall configurations. Nowadays, the assumption that port 80 traffic is solely HTTP Web traffic is hardly true.

It is clear from the above that traditional network monitoring methods for determining per-applications network usage are not effective anymore for accurate traffic categorization [17]. Having identified this issue, several researchers have conducted significant work towards alternative ways for network traffic classification. Due to the popularity and high bandwidth demands of peer-to-peer file sharing applications, a significant body of work has focused on the identification and categorization of peer-to-peer application traffic. Initial approaches used deep packet inspection and application signatures for attributing traffic flows to the corresponding applications [13, 19]. Recent approaches identify the applications that

generate the traffic either by deriving statistical models for certain protocols [11] or by characterizing the behavior of the host generating this traffic [14, 15].

Motivated by the significance of traffic categorization for effective network management and traffic engineering and aiming at gaining a better understanding of Internet traffic, we have developed *appmon*, a passive network monitoring application for accurate per-application traffic identification and categorization. *appmon* uses three different approaches for attributing flows to the applications that generate them. First, it searches inside application messages for characteristic application protocol patterns. For certain applications that dynamically negotiate the ports that are going to be used, *appmon* fully decodes the applications protocol to identify the new, dynamically generated port number and then tracks further traffic flows through these ports. Finally, legacy applications that do not match above filters are categorized based on well-known port numbers and protocols using BPF filters.

Appmon has been implemented on top of Monitoring API (MAPI) [18, 20]. The core of the application is a new MAPI function library, called `trackflib`. This new library provides several different filters, called *trackers*, which can identify and track the traffic of different network applications. Trackers have been implemented using various traffic filtering techniques, such as *deep packet inspection*, *protocol decoding*, and *header filtering*.

The rest of the document is organized as follows. Section 2 gives an extensive description of the design and implementation of*appmon*. Section 3 presents the experimental evaluation of the application, and Section 4 presents its current deployment to several monitoring sensors. Finally, Section 5 summarizes the document.

Chapter 2

Design and Implementation

This chapter presents the overall design of *appmon*, including a detailed description of the traffic classification algorithm, implementation details, and the Web-based graphical user interface.

2.1 Traffic Classification

appmon passively monitors the traffic that passes trough the monitored link and categorizes the active network flows according to the application that generated them. A network flow is defined as a set of IP packets with the same protocol, source and destination IP address, and source and destination port (also known as a 5-tuple). Traffic categorization is performed using information from both the packet header and payload .

The classification algorithm operates as follows: *appmon* processes each captured network packet sequentially. For each captured packet, it first checks if the packet belongs to an already categorized network flow. Information about the network flows seen so far is stored into a hash table, along with information about the matching application. *appmon* keeps the minimal state required in order to reduce the packet processing time. This allows for a "fast path" processing of subsequent packets of an already categorized flow, since they will only result to a look up in the hash table for finding the record of the network flow in which they belong, and, consequently, the matching application, without the need for any further packet processing.

Packets that do not have a matching entry in the hash table are passed down to the next processing level, where each packet is sequentially processed by a set of modules called application trackers. Each tracker is responsible for identifying the traffic of a particular application or protocol. There are three different types of application trackers, depending on method used for classifying traffic: *packet inspection* trackers, *protocol decoding* trackers, and *header filtering* trackers. Trackers have been implemented as separate MAPI functions, and have been collected into

a separate MAPI function library, called `trackflib`, for facilitating reuse of the tracking functionality from other applications.

Packet inspection trackers. Packet inspection trackers are used for tracking application level protocols, mainly used in peer-to-peer file sharing applications such as Gnutella [2] and BitTorrent. Packet inspection trackers search inside packet payloads for characteristic application messages or binary byte sequences that are used by application protocols. These application messages were selected by extensively reverse-engineering the network traffic of popular file sharing applications, as well as by studying the related work on signature-based traffic classification [6, 13, 19]. Although pattern matching inside packet payloads is a quite CPU intensive operation, in most cases the characteristic application patterns, usually protocol control messages, are present in the first 100 bytes of the packet payload, and thus the pattern matching is performed only to this portion of the payload, reducing significantly the processing overhead.

Protocol decoding trackers. Protocol decoding trackers are used for publicly documented application level protocols that operate through well known control ports, but use a dynamically assigned port for data exchange. For example, in passive FTP, control messages are exchanged through port 21, but actual data transfers are made through a dynamically negotiated port. Protocol decoding trackers operate by fully decoding the application-level messages exchanged through the well-known control port, trying to identify the messages related with the negotiation of port numbers that will be used for future data transfers. When such a message is identified, the number of the dynamic port is extracted and then the tracker will correctly classify the new network flow that is going to be used for the data transfer, since the flow will use this dynamically negotiated port.

Header filtering trackers. If none of the above groups of trackers succeeds in identifying a given packet, then the packet is passed to the header filtering trackers. Filtering trackers classify traffic based on packet header information such as identifying predefined registered ports [4] and other protocol information. Filtering trackers are implemented using BPF filters [16].

As we have already discussed, several applications masquerade their traffic using widely used, firewall-friendly protocols, like HTTP, in order to bypass firewall restrictions and make identification of their traffic harder. To avoid potential traffic misclassification due to such tricks, trackers are prioritized, with packet inspection trackers applied first, then the protocol decoding trackers, and finally header filtering trackers. When a packet is matched by a tracker, then it is not processed further by subsequent trackers. For example, the BitTorrent tracker has higher priority than the HTTP Web tracker. Thus, the flow of a BitTorrent packet through port 80 will be correctly attributed to the BitTorrent protocol, and not to Web traffic.

Layer 4 Protocols	Application Protocols	
TCP	BitTorrent	eDonkey
UDP	Direct Connect	Gnutella
ICMP	FTP	HTTP
IP-in-IP	SSH	SMTP
	DNS	NetBIOS
	RTSP	OpenVPN
	GRID FTP	BDII
	GRIS	

Table 2.1: Implemented Protocol Trackers

If none of the above methods manages to classify the flow in which the packet belongs, then the packet is temporarily considered as unknown, and the application waits for more packets of the same flow in order to classify it.

It is worth mentioning that since most of the application specific patterns are located at the beginning of a flow, the vast majority of the monitored packets will belong to an already active—and thus categorized—network flow. As a result, expensive deep packet inspection operations are performed only to a small subset of the traffic, and appmon manages to process traffic loads of several hundred Mbit/s

Table 2.1 presents the currently implemented protocol trackers in appmon. We split these protocols into two broad categories. The first contains the main layer 4 protocols, while the other contains application-level protocols, including those used by several popular traffic-dominating peer-to-peer applications.

2.2 Implementation

The core functionality of *Appmon* has been implemented as a new MAPI function library. The Tracker MAPI function library (`trackflib`) provides implementation of the *packet inspection* and *decoding* tracker functions described in section 2.1 inside the Monitoring API. A description of all available tracker functions implemented in `trackflib` is presented in Appendix A.1.

For each application protocol, *appmon* creates a new network flow and applies the appropriate function from the `trackflib` library. It then applies a `BYTE_COUNTER` function and retrieves the results needed for visualization to take place. *Appmon* also makes use of the `TOP` function of MAPI to find the TOP IP addresses for each protocol. The TOP function is located in `extraflib` library and is described in Appendix A.2.

The following pseudocode shows how a typical MAPI flow created by *appmon* looks like.

```
1  /* create the flow */
2  fd = mapi_create_flow("host:dev");
```

```
3
4  /* apply the tracker function */
5  mapi_apply_function(fd, "TRACK_GNUTELLA");
6
7  /* apply byte counter */
8  fid = mapi_apply_function(fd, "BYTE_COUNTER");
9
10 /* apply the TOP function */
11 top_fid  = mapi_apply_function(fd, "TOP",
12     10, TOPX_IP, TOPX_IP_SRC_IP, SORT_BY_BYTES, 0);
13
14 while(1) {
15     sleep(10);
16
17     /* read counters */
18     res = mapi_read_results(fd, fid);
19
20     /* read TOP IP address */
21     res = mapi_read_results(fd, top_fid);
22
23     /* report results */
24 }
```

Appmon uses the RRDtool suite [7] for storing measurement data and graphing the traffic distribution. The Round Robin Database provided by RRDtool efficiently stores time-series data for very long periods in very little space using data aggregation. The database used by *appmon* has a size of a few megabytes and can store measurements for a period as long as one year.

The installation of the Web interface requires a web server like Apache with no extra packages. The results are rendered using simple CGI scripts and plain html code.

2.3 Graphical User Interface

Appmon reports the classification results through two different user interfaces, depending on the requirements of the user. For quick and easy network monitoring, there is a console-mode version which can report the results either through a batch text mode printout, or a more user-friendly ncurses [10] version. For long-term usage, *appmon* provides a powerful GUI accessible using any web browser. In this document we describe only the Web interface, since it provides a superset of the information provided by the console-mode versions.

Appmon reports the per-application traffic distribution through the web interface presented in Figure 2.1. The main page is split into three frames. The central frame presents a graph of the incoming and outgoing traffic distribution for the last hour. The graph presents the traffic portion of each categorized application with a

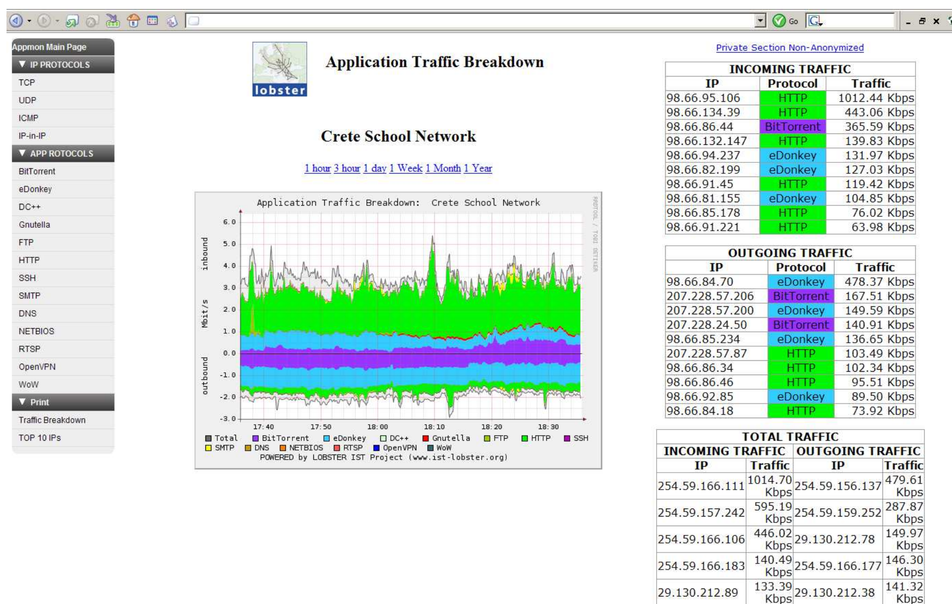

Figure 2.1: Appmon Web Interface.

different color, while any remaining non-categorized traffic is shown in grey. The topmost/bottommost line represents the total observed traffic load.

The information of this frame is separately presented in Figure 2.2, which presents the per-application distribution of the incoming and outgoing traffic at the University of Crete in Greece. The values are expressed in Mbit/s, and the graph is updated every 10 seconds. A detailed per-application breakdown of the traffic load is presented underneath the graph.

The application offers five different time period views of the traffic distribution. The main view presents the per-application traffic distribution of the last hour. Links also exist for the time period of the last three hours, last day, last week, last month and last year.

Besides traffic classification, *appmon* also reports the K top bandwidth consuming IP addresses. This is done by accumulating the traffic of each IP address after every packet is categorized at a specific application. In order to achieve this some extra state is needed. For every protocol we keep a hash table with all the IP addresses that belong to flows marked as belonging to this protocol. For every IP address we keep the number of bytes it transmitted, and the addresses are sorted in descending order according to the amount of traffic seen so far.

The top bandwidth consuming IP addresses are showed in three tables in the right frame of the Web interface. The first two tables contain the IP addresses of the

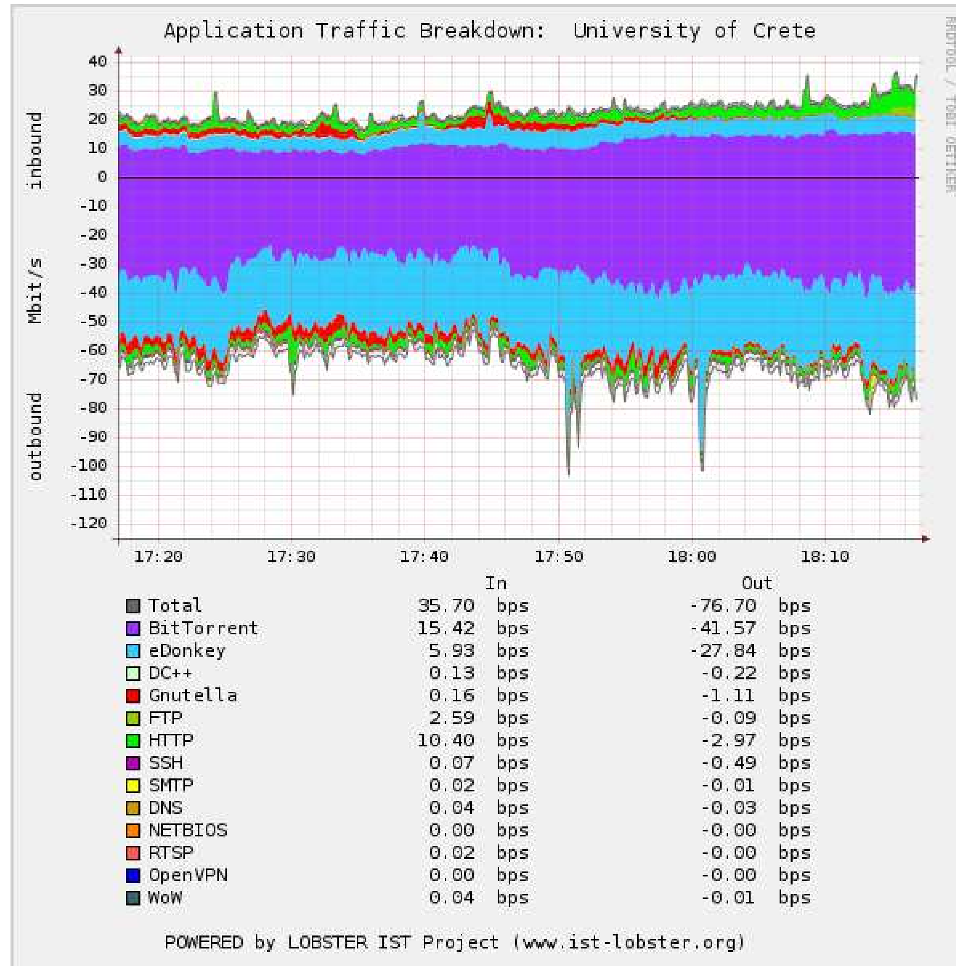

Figure 2.2: Per-application bandwidth usage.

K (10 by default) flows that consumed the largest portion of bandwidth during the last measurement period. Each record contains information about the application in which the flow belongs to and the exact amount of bandwidth that it consumed. The third table presents the same information at the IP level, which corresponds to the top K IP addresses that consumed the largest portion of bandwidth irrespective of application. Figure 2.3 shows an example of how the top 10 IP addresses are presented through the web interface.

Since information about IP addresses is sensitive and in some cases it may not be desirable to be exposed, *appmon* can anonymize all the IP addresses presented by the Web interface. Address anonymization is performed using prefix-preserving anonymization [12, 22, 23], which preserves subnet information. A non-anonymized version of the TOP IP address information is also available for view only by authorized personnel using a login procedure.

INCOMING TRAFFIC		
IP	Protocol	Traffic
98.66.95.106	HTTP	1012.44 Kbps
98.66.134.39	HTTP	443.06 Kbps
98.66.86.44	BitTorrent	365.59 Kbps
98.66.132.147	HTTP	139.83 Kbps
98.66.94.237	eDonkey	131.97 Kbps
98.66.82.199	eDonkey	127.03 Kbps
98.66.91.45	HTTP	119.42 Kbps
98.66.81.155	eDonkey	104.85 Kbps
98.66.85.178	HTTP	76.02 Kbps
98.66.91.221	HTTP	63.98 Kbps

Figure 2.3: Top 10 incoming traffic IP addresses as presented by appmon.

Finally, the left frame of the Web interface gives the user the ability to view the traffic of only selected protocols through a menu with all available protocols. Using the menu, the user can select only a few application protocols, in order to get information about both the traffic and the TOP IP addresses of these specific protocols.

Chapter 3

Performance Evaluation

In this section, we present a performance evaluation of *appmon*, through a series of experiments performed in a local testbed, as well as using *appmon* installations that monitor real production traffic.

3.1 Local Testbed

Our first experiment aims at exploring the performance of our application. We used a local testbed consisting of three PCs connected to a gigabit switch, as shown in Figure 3.1. The ”Sender” PC generates traffic destined to the ”Receiver” PC using the nttcp [5] tool. The traffic from both hosts is mirrored to the third monitoring machine which is running *appmon*.

The configuration of the measurement machine is as follows. We used an Intel Xeon 2.4 MHz, with 512 KB cache and 512 MB memory. The Operating System was Debian Linux with 2.6.15 kernel version. Two kinds of network interfaces were used. A regular Gigabit Ethernet interface (NIC), and a specialized DAG 4.3GE packet capturing card [1].

It is important to mention that nttcp produces artificial traffic by filling the packet payload with random bytes. This is a worst-case traffic load for *appmon* since none of the packets matches any of the monitored protocols. Thus, every packet passes through the slow processing path, going through all tracker functions, since none of the packets has a matching entry in the hash table, and none of the trackers is able to find a matching packet.

We stressed *appmon* by sending traffic in various speeds. Figure 3.2 shows the results for both NIC and DAG interfaces. As we can see *appmon* can process up to 500 Mbit/s without any packet loss when running on a regular NIC interface (blue line), while it is able to process all 900 Mbit/s when running on top of the DAG card (green line). The results imply that the application can fully monitor a Gigabit link using a DAG card.

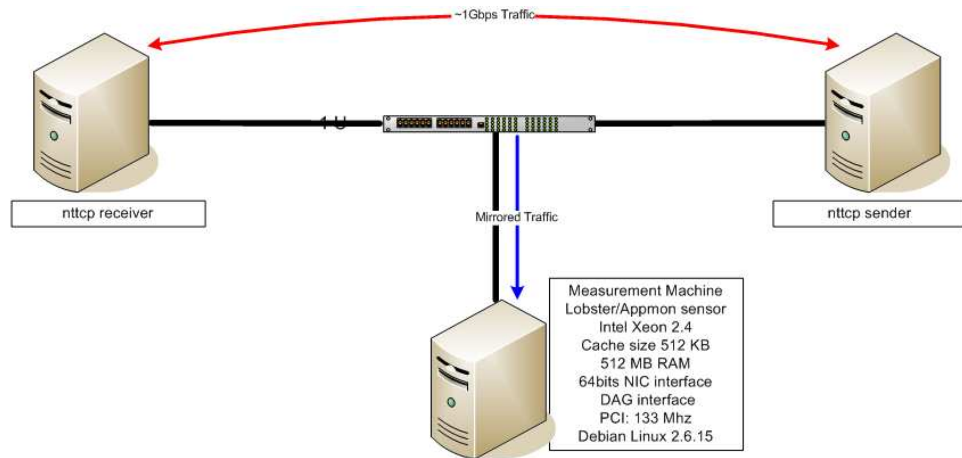

Figure 3.1: *appmon* measurement environment.

3.2 Monitoring Real Network Traffic

For our second experiment we deployed *appmon* in a real network environment, aiming at verifying the performance results of the first experiment. *appmon* was installed on a sensor at University of Crete, monitoring the incoming and outgoing traffic from the campus to the Internet. The monitoring machine was an Intel Xeon 3.2GHz, with 2MB cache memory and 1GB main memory, running a Debian Linux, kernel version 2.6.15. The traffic is captured using a DAG 4.2GE passive monitoring card. Along with the traffic load reported by the application, we measured the CPU load of the machine. A new measurement result was produced every 10 seconds for a measurement period of four days.

Figure 3.3 presents the CPU load (y-axis) of the monitoring sensor as a function of the monitored traffic load (x-axis). Each point corresponds to a five minute interval, computed as the average of the measurements performed every 10 seconds in that interval. *appmon* has a steady behavior, since the CPU load increases as the traffic load increases. Some corner cases in which the load is increased significantly while the traffic load is low are probably caused due to the almost simultaneous arrival of many new traffic flows that have not yet been categorized.

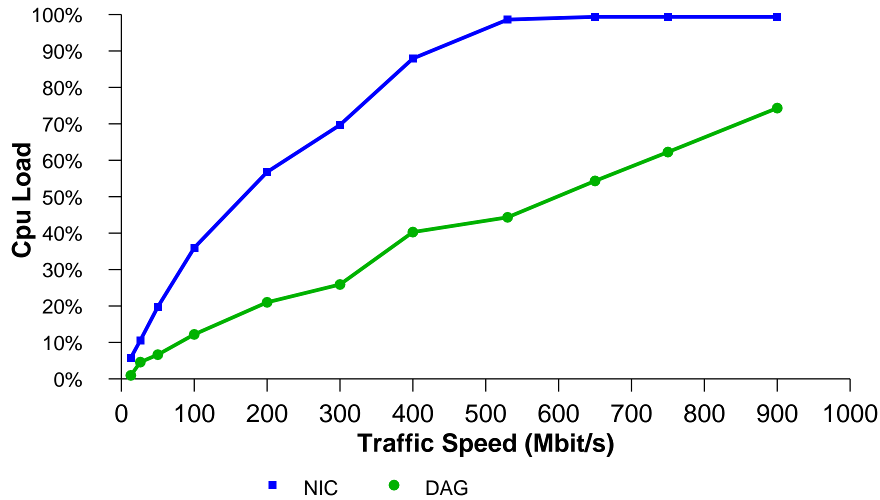

Figure 3.2: *appmon* CPU Load Vs. Traffic Load for both NIC and DAG Interfaces using the local testbed descriped in Figure 3.1.

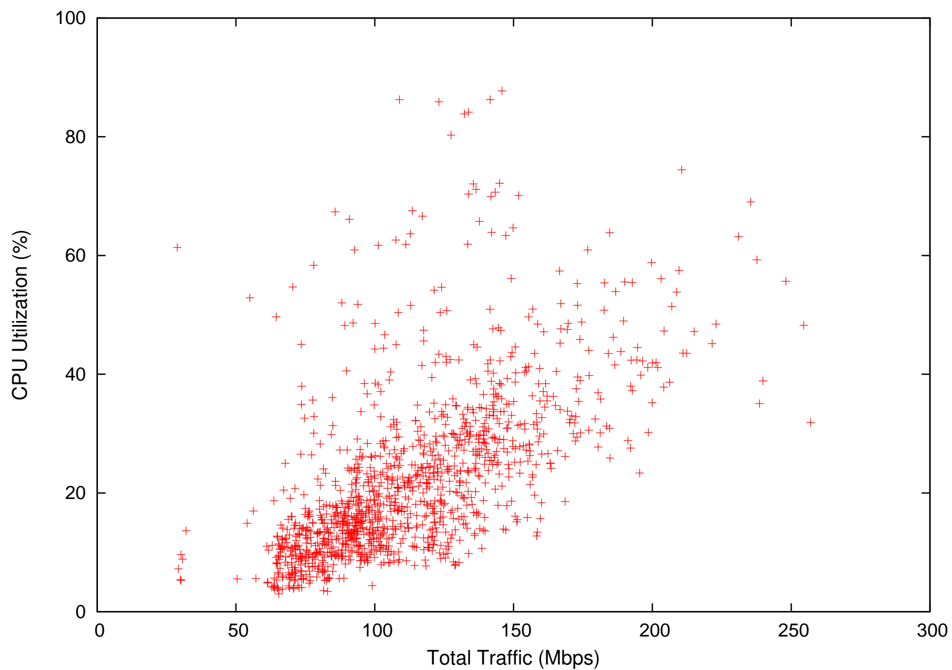

Figure 3.3: *appmon* performance on a sensor deployed on University of Crete for serveral days. Each measurement was taken every 10 seconds.

Chapter 4

Deployment

Taking advantage of the distributed monitoring functionality of MAPI, we have deployed *appmon* in several monitoring points around the world. Figure 4.1 presents a map with deployed Lobster applications in several places around the world. Currently we have deployed *appmon* sensors in six institutions in Greece; the Foundation of Research and Technology Hellas, the University of Crete, the Greek School Network, the Node of HellasGRID in Crete, the National Technical University of Athens and Aristotle University of Thessaloniki. We have also deployed sensors in Czech Republic, as shown in Figure 4.2, and several sensors in Norway.

Figures 4.3 and 4.4 present a graphical representation of LOBSTER sensors using the Google EARTH [3] tool. The first figure show the current network contitions in the sensor located at ICS-FORTH, while the second figure shows the current network contitions of the sensor at the HellasGRID node.

Figure 4.5 presents *appmon* in action monitoring the Gbit network of the National Technical University of Athens. This sensor is deployed using a DAG Network Interface. Figure 4.6 presents *appmon* on a sensor at Aristotle University of Thessaloniki where, while deployed on a regular NIC interface, it manages to process 500 Mbps of traffic.

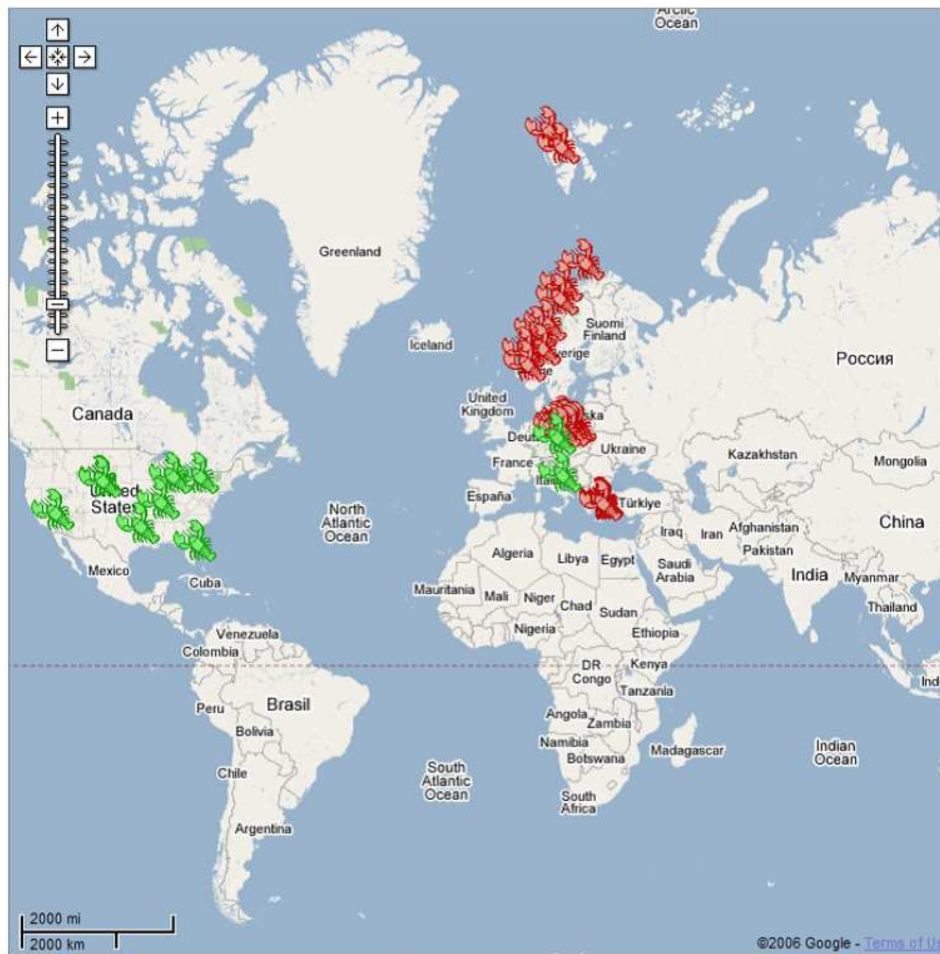

Figure 4.1: Lobster Applications Deployment.

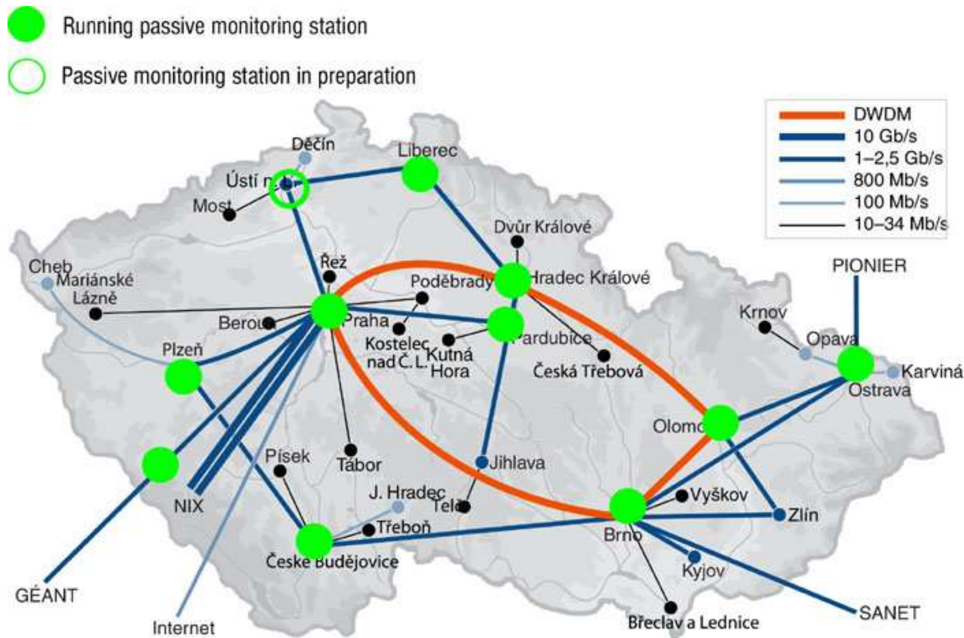

Figure 4.2: Application deployment in Checz Republic Monitoring the connection with GEANT2.

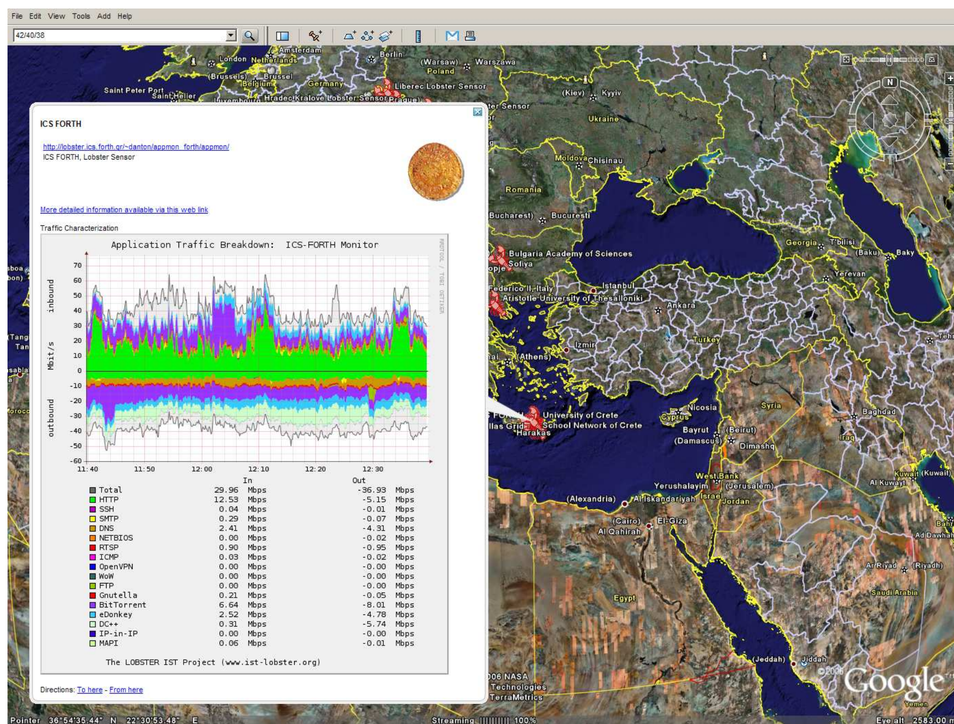

Figure 4.3: Google Earth Application showing results from FORTH-ICS sensor.

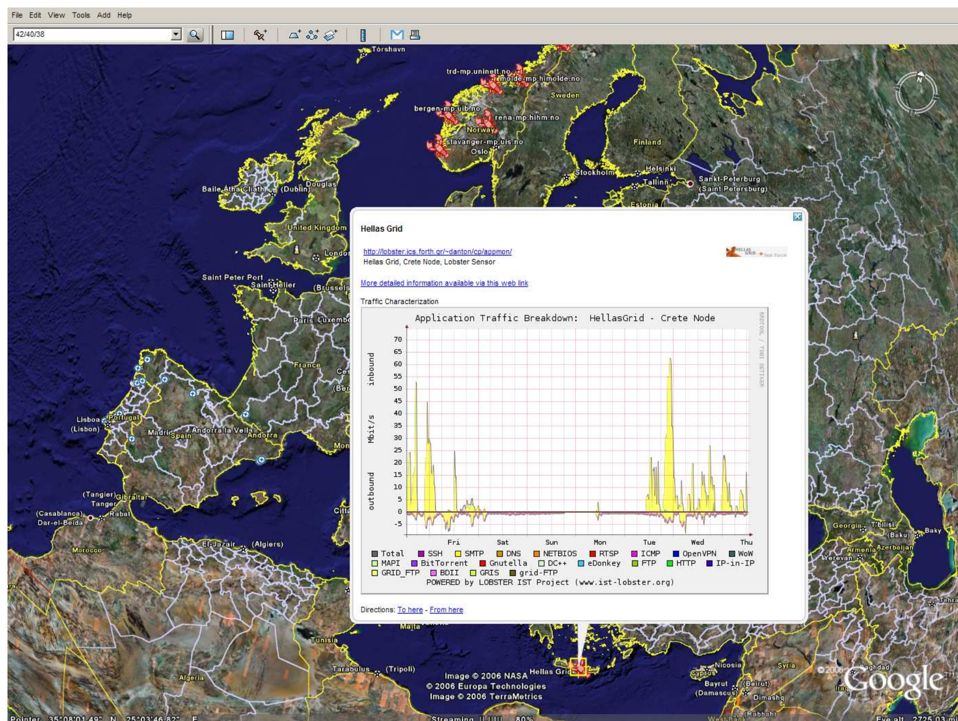

Figure 4.4: Google Earth Application showing results from HellasGRID sensor.

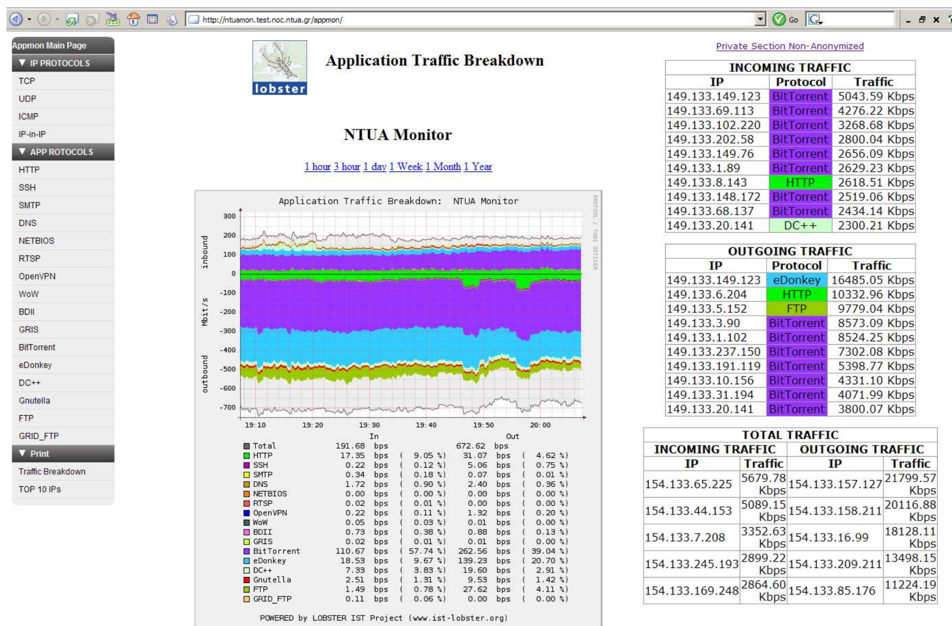

Figure 4.5: *appmon* running on a sensor at National Technical University of Athens. Using a DAG network interface *appmon* manages to process traffic in Gbit speeds.

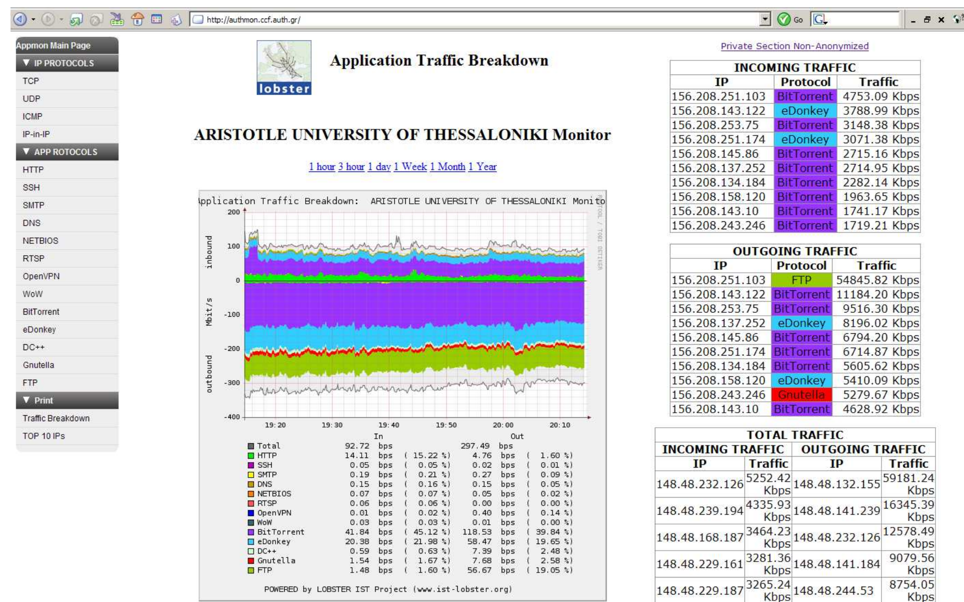

Figure 4.6: *appmon* running on a sensor at Aristotle University of Thessalloniki. Using a NIC card *appmon* can process traffic load of 500 Mbps.

Chapter 5

Summary

In this document we have presented *appmon*, an application for real time per-application network traffic categorization. The main goal of the application is to visualize the network traffic usage in order to help in effectively monitoring the network traffic usage. As we have shown, *appmon* is able to categorize traffic in speeds that reach the one Gbit/s. *appmon* uses a large set of protocol trackers for the classification of traffic from many emerging applications, while its module design allows for the easy addition of more protocol trackers in the future.

With several ”bandwidth-hungry” applications increasingly trying to make their traffic difficult to detect, we expect that the use of encrypted traffic is on the way. In order to address this problem, we plan to explore whether non payload traffic classification methods can be used to identify and classify network traffic in real time.

Appendix A

Appendix

A.1 Tracker library

The Tracker MAPI function library (`trackflib`) provides a set of predefined functions for per-application traffic monitoring, even for hard-to-track applications that use dynamically negotiated ports or use existing protocols and port numbers (e.g., non-web traffic using HTTP through port 80) to masquerade their traffic.

Each tracker function scans the network traffic for packets that belong to some specific application-level protocols. If such a packet is found, then the traffic of the flow in which the packet belongs to is attributed to the identified application. All tracker functions operate as network filters, for example in a similar way to the `BPF_FILTER` function. Thus, applying a tracker function to a network flow will result to capturing only the traffic of the particular application. For example, a network flow on which the `TRACK_GNUTELLA` function has been applied, will receive only traffic that belongs to Gnutella P2P file sharing applications.

A detailed description and evaluation of the Tracker MAPI function library (`trackflib`) is presented in [21].

A.1.1 TRACK_FTP

The `TRACK_FTP` function, when applied to a network flow, filters the network traffic by keeping only packets that belong to the FTP protocol. The function supports both active and passive FTP modes.

A.1.2 TRACK_DC

The `TRACK_DC` function, when applied to a network flow, filters the network traffic by keeping only packets that belong to the DC++ file sharing application/protocol.

A.1.3 TRACK_GNUTELLA

The TRACK_GNUTELLA function, when applied to a network flow, filters the network traffic by keeping only packets that belong to the Gnutella file sharing application/protocol.

A.1.4 TRACK_EDONKEY

The TRACK_EDONKEY function, when applied to a network flow, filters the network traffic by keeping only packets that belong to the eDonkey file sharing application/protocol.

A.1.5 TRACK_TORRENT

The TRACK_TORRENT function, when applied to a network flow, filters the network traffic by keeping only packets that belong to the BitTorrent file sharing protocol.

A.1.6 TRACK_GRID_FTP

The TRACK_GRID_FTP function, when applied to a network flow, filters the network traffic by keeping only packets that belong to the File Transfer Protocol used by GRID systems.

A.2 Extra Functions library

The extra MAPI function library (`extraflib`) provides a set of functions that cover more advanced monitoring needs. A detailed description and evaluation of the Extra Functions MAPI function library (`extraflib`) is presented in [21].

A.2.1 COOKING

The `COOKING` function processes the packets of a flow according to the TCP/IP protocol stack, by performing IP defragmentation and TCP stream reassembly. The received packets are stripped from their TCP/IP headers and assembled into a single "cooked" packet. The cooked packet is prepended with a pseudo TCP/IP header containing the size of the cooked packet and the source and destination IP addresses and port numbers.

A.2.2 REGEXP

The `REGEXP` function filters the traffic of a flow by keeping only the packets that match the given PCRE regular expression.

A.2.3 TOP

The `TOP` function returns the top `x` (first argument) values of a given packet header field. For example, it can be used for identifying the top ten IP addresses that generate the most traffic.

A.2.4 EXPIRED_FLOWS

The `EXPIRED_FLOWS` function facilitates passive end-to-end packet loss measurement.

Bibliography

[1] ENDACE. DAG Network Monitoring Interface Card. http://endace.com/networkMCards.htm.

[2] Gnutella. http://www.gnutella.com.

[3] Google Earth. http://earth.google.com.

[4] Internet Assigned Numbers Authority. http://iana.org.

[5] nttcp. http://sd.wareonearth.com/~phil/net/ttcp/.

[6] Protocol info. http://protocolinfo.org.

[7] rrdtool. http://oss.oetiker.ch/rrdtool/.

[8] Skype. http://www.skype.com.

[9] The BitTorrent Protocol. http://www.bittorent.org.

[10] The Ncurses Library. http://www.gnu.org/software/ncurses/ncurses.html.

[11] Laurent Bernaille, Renata Teixeira, Ismael Akodkenou, Augustin Soule, and Kave Salamatian. Traffic classification on the fly. *SIGCOMM Comput. Commun. Rev.*, 36(2):23–26, 2006.

[12] Crypto-Pan To Cisco. Network log anonymization: Application of.

[13] T. Karagiannis, A.Boido, N. Broenlee, kc claffy, and M. Galoutros. Is p2p dying or just hiding? In *Proceedings of the IEEE Globecom*, 2004.

[14] Thomas Karagiannis, Andre Broido, Michalis Faloutsos, and Kc claffy. Transport layer identification of p2p traffic. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 121–134, New York, NY, USA, 2004. ACM Press.

[15] Thomas Karagiannis, Konstantina Papagiannaki, and Michalis Faloutsos. Blinc: multilevel traffic classification in the dark. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures,*

and protocols for computer communications, pages 229–240, New York, NY, USA, 2005. ACM Press.

[16] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, pages 259–270, 1993.

[17] A. Moore and K. Papagiannaki. Toward the accurate identification of network applications. In *Passive and Active Measurement Workshop*, March 2005.

[18] Michalis Polychronakis, Kostas G. Anagnostakis, Evangelos P. Markatos, and Arne Øslebø. Design of an Application Programming Interface for IP Network Monitoring. In *Proceedings of the 9th IFIP/IEEE Network Operations and Management Symposium (NOMS'04)*, pages 483–496, April 2004.

[19] S. Sen, O. Spatscheck, and D. Wang. Accurate, scalable in-network identification of p2p traffic using applications signatures. In *Proceedings of the 13th International World Wide Web Conference*, 2004.

[20] The LOBSTER Consortium. MAPI Public Release. `http://mapi.uninett.no/download/mapi-2.0-beta1.tar.gz`.

[21] The LOBSTER Consortium. Deliverable D3.2: Deployment - first phase, November 2006. `http://www.ist-lobster.org/publication/deliverables/D3.3.pdf`.

[22] Bangnan Xu and Bernhard Walke. Design issues of self-organizing broadband wireless networks. *Computer Networks (Amsterdam, Netherlands: 1999)*, 37(1):73–81, 2001.

[23] J. Xu, J. Fan, M. Ammar, and S. Moon. Prefixpreserving ip address anonymization: Measurement-based security evaluation and a new cryptography-based scheme, 2002.