Traffic Characterization:

An Application for Monitoring Peer-To-Peer Systems

Undergraduate Thesis

Demetres Antoniades

AM: 1215

danton@csd.uoc.gr

Supervisor:

Prof. Evangelos Markatos

markatos@csd.uoc.gr

University of Crete – Computer Science Department

March, 2005

Abstract

As networks get bigger and faster the role of monitoring applications, becomes more complicated. Also application programmers become more clever and manage to camouflage the traffic of their application using dynamic ports. Motivated by the needs for better network monitoring, we have developed a tool for monitoring both static and dynamic port applications. The tool we present is developed using the newly released Monitoring API (MAPI), and aims at expanding our knowledge of what is going on into our network. Using our tool we managed to reduce the portion of unknown traffic, as reported by other –static– classification methods, by almost 57%.

Acknowledgments

The work presented in this undergraduate thesis was done while both the writer and his supervisor where at the Institute of Computer Science at the Foundation for Research and Technology Hellas (ICS - FORTH), and where both actively working on the IST project SCAMPI.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As networks get faster and as network-centric applications get more complex, our understanding of the underlying Internet traffic continues to diminish. Nowadays, we frequently discover, to our surprise, that there exist new aspects of Internet behavior that are either unknown or poorly understood.

Whether some incidents are malicious, like viruses and worms, or not we come against the lack of understanding of the Internet, that shows the need for better Internet traffic monitoring and ability to provide better traffic characterization, even for applications that want to elude characterization, such as peer-to-peer systems.

One of the most frequent requests of network administrators is to find out the applications that generate the largest amount of traffic that flows through the monitored networks. Most monitoring systems assume that each application is associated with only one static well-known port. Although this is true for several of the traditional applications, most of the emerging applications, such as peer-to-peer systems and video conferencing systems, use a variety of dynamically generated ports. Therefore, traditional ways of monitoring usually end up unable to monitor the network.

As an example Figure 1.1 shows the distribution of traffic among various applications for the University of Wisconsin during the year from February 2004 till February 2005. The legend reads that 2.0% of the outgoing traffic is attributed to eDonkey, 0.8% of the outgoing traffic is attributed to Gnutella, 0.1% of the outgoing traffic is attributed to KaZaa, 22.7% of the outgoing traffic is attributed to HTTP, etc. What is the most important to see, however is that the penultimate line of the figure legend states, that 64.5% of the outgoing traffic is attributed to ”Other” applications, that is, applications besides peer-to-peer systems, besides web, besides ftp, besides Usenet news (nntp),

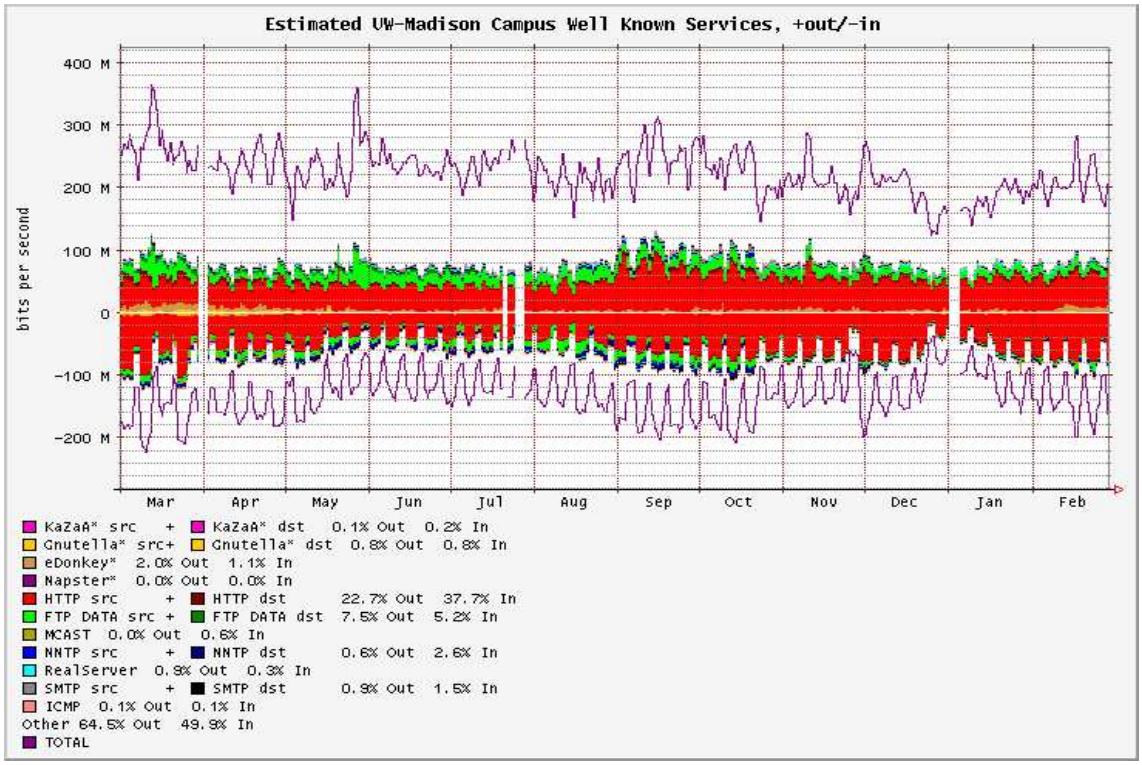

Figure 1.1: Traffic Distribution of the network of the University of Winsonsin from Feb. 2004 till Feb. 2005. The photograph is courtesy of wwwstats.net.wisc.edu

besides email (smtp), besides real audio/video, etc. It is surprising to see that close to 65% of the outgoing traffic of a University with well-monitored networks is attributed to applications that we do not currently know. Actually this surprising effect is probably due to the fact that the methodology used to categorize traffic into applications is based on traditional monitoring methods that associate applications with static ports, while an emerging number of popular applications use dynamic ports. Thus, applications that use dynamic ports, such as KaZaA, seem to consume only 0.1% of the traffic while previous measurements, form the era when those applications used static ports reported that peer-to-peer applications consumed the largest portions of traffic (3). Indeed, analysis based on passive monitoring tools reveals that peer-to-peer applications consume three times more traffic than analysis based on static port numbers (16). As a result, in the above example, traditional monitoring methods cannot account for the 65% of the traffic, and in the end, network administrators are not able to see which applications consume which percentage of the traffic: in the above example they do not know who is using 65% of the network's traffic.

Motivated by this situation and feeling the need to give, to ourselves and others, better understanding to the Internet traffic, we developed an application, in order to identify and categorize

network traffic. In our application we deploy passive monitoring by applying string searching functions in network packets, in order to identify which packets are destined for which application. Continously we do packet inspection in order to identify the new, dynamically generated port, that is going to be used from the application.

Our application is built over the newly developed Monitoring API (MAPI), an expressive programming interface, that enables users to clearly communicate their monitoring needs to the underlying traffic monitoring platform.

This thesis presents our approach of achieving better monitoring of Internet traffic. In Chapter 2 we take a brief look at MAPI, and what we can achieve by using it. Chapter 3 presents the ways several applications use dynamic ports to transfer their data. In Chapter 4 we present our program implementation, and in Chapter 5 we take a look at the results we achieved. Finally Chapter 6 summarizes and concludes our work, while gives our thoughts for future work.

Chapter 2

The Monitoring Application Programming Interface

MAPI (13) was implemented during the IST project, SCAMPI (9), and is presented in (12; 21). It is an expressive programming interface, which enables users to clearly communicate their monitoring needs to the underlying traffic monitoring platform.

The Monitoring Application Programming Interface (MAPI) builds on a flow abstraction that allows users to tailor measurements to their own needs. The main novelty abstraction of MAPI is that it elevates flows to first class status, allowing programmers to perform a set of standard operation on flows similar to other system abstractions such as files and sockets. Where necessary and feasible, MAPI also allows the user to trigger custom processing routines not only on summarized data but also on the packets themselves, similar to programmable monitoring systems. The expressiveness of MAPI enables the underlying monitoring system to make informed decisions in choosing the most efficient implementation, while providing a coherent interface on top of different lower-level elements, including intelligent switches, high-performance network processors, and special-purpose network interface cards.

MAPI builds on a simple and powerful abstraction, the *network flow*, but in a flexible and generalized way. In MAPI, a network flow is generally defined as a sequence of packets that satisfy a given set of conditions. These conditions can be arbitrary, raging from simple header-based filters, to sophisticated protocol analysis and content inspection functions.

In order to give the ability to users to express a wide variety of new monitoring operations,

MAPI gives the ”network flow” a first-class status. That means, flows are named entities that can be manipulated in similar ways to other programming abstractions such as sockets, pipes, and files. In particular, users may create or destroy flows, sample or count packets of a flow, apply functions to flows, and retrieve other traffic statistics from a flow.

2.1 Creating and Terminating Network flows

Central to the operation of the MAPI is the action of creating a network flow:

fd = mapi_create_flow(char dev)*

*fd = mapi_create_offline_flow(char *path, int format, int speed);*

Both calls create a network flow, and return a flow descriptor *fd* that points to it. The first function's flow consists of all network packets which go through network device *dev*. In contrast the second function's flow consists of all the packets which are saved in the file denoted by *path* and have the *format*, format.

Besides creating a network flow, monitoring applications may also close the flow when they are no longer interested in monitoring, using:

int mapi_close_flow(int fd)

2.2 Applying Functions to Network Flows

Network flows allow users to treat packets that belong to different flows in different ways. For example, a user may be interested in *logging* all packets of a flow (e.g. to record an intrusion attempt), or in just *counting* the packets and their lengths (e.g. to count the bandwidth usage of an application), or in *sampling* the packets (e.g. to find the IP addresses that generate most of the traffic). The abstraction of the network flow allows the user to clearly communicate to the underlying monitoring system these monitoring needs. To enable users to communicate these different requirements, MAPI enables users to associate functions with flows, using:

fid = mapi_apply_function(int fd, char funct, ...)*

The above association applies function *funct* –with its parameters *(...)*– to every packet of flow *fd*, and returns an id –*fid*– for the function. Based on the header and the payload of the packet, the function will perform some computation, and may optionally discard the packet.

Table 2.1: Short description of the predefined MAPI Apply Functions

Function Name	Description
BPF_FILTER	Filters the packets of a flow
PKT_COUNTER	Keeps the number of packets seen by a network flow
BYTE_COUNTER	Keeps the number of bytes seen by a network flow
STR_SEARCH	Searches for a string inside the packet payload
TO_BUFFER	Stores the packets of a flow for further reading
SAMPLE	Samples packets from a flow
HASHSAMP	Samples packets from a flow according to a hashing function
TO_FILE	Dumps the packets of a flow to disk
ETHEREAL	Filters packets using Ethereal display filters
HASH	Computes an additive hash over the packets of a flow
COOKING	TCP/IP packet de-fragmentation and stream reassembly
BUCKET	Divides packets into buckets based on their timestamps
THRESHOLD	Signals when a threshold is reached

MAPI provides several predefined functions that cover some standard monitoring needs. These functions are summarized in Table 2.1 Although this functions will enable users to process packets, and compute the network traffic metrics they are interested in without receiving the packets in their own address space, they must somehow communicate their results to the interested users. For example a user that will define that the function *PKT_COUNTER* will be applied to a flow, will be interested in reading what is the number of packets that have been counted so far. This can be achieved by using the function:

*void * mapi_read_results(int fd, int fid, int copy)*

This function will return a pointer to the results of function *fid*, which was applied in flow *fd*, if *copy* is set to FALSE, or a *copy* of the results, if copy is set to TRUE.

2.3 Reading packets from a flow

Once a flow is established, the user will probably want to read packets from the flow. Packets can be read one-at-a-time using the *blocking* call:

struct mapipkt mapi_get_next_pkt(int fd, int fid)*

which must be preceeded by an applied *TO_BUFFER* function, – with function id *fid* – to the flow *fd*.

MAPI also gives the functionality for declaring callback functions, which will be called when a packet to the specific flow is available. This is done using the functions:

mapi_loop(int fd, int fid, int cnt, mapi_handler)

The above call makes sure that the *mapi_handler* callback will be called for each of the next *cnt* packets that will arrive in the flow *fd*.

Chapter 3

Challenge of Dynamic Ports

In this chapter we discuss how dynamic ports are used in applications. We first take a look to the well known and documented approach of FTP and next we look at Fasttrack, a wide used peer-to-peer protocol.

3.1 The FTP Dynamic Approach

FTP (10) differs from most other TCP services because it communicates using two different server ports. The first port is port 21, which is well-known as the standard FTP command port. This port is used by FTP only as a control channel. The second port is used for data exchanged between the client and the server. When a file transfer is initiated, the FTP server informs the client about the dynamic port number to be used for the transfer.

Fortunately, FTP is well documented and the procedure followed by the server–client pair is known. The server sends a packet containing the following message:

227 Entering Passive Mode (139,91,70,70,146,226)

This message contains the IP address (139.91.70.70) and the port number to be used for the forthcoming transfer (37602), generated by the two last bytes (146 & 226).

Therefore in order to accurately account for all FTP traffic, a monitoring application needs to monitor port 21 to find new clients as well as the dynamic ports these new clients will use in order to transfer their data. This, kind of monitoring can be easily be done using MAPI, described in Chapter 2. What we need to do is apply a *BPF_FILTER* function to the flow requesting for packets

that come from port 21. Next we apply a *STR_SEARCH* function to that packets in order to find the pattern. We then request each of this packets, and we inspect the packet to extract the IP address and the port number.

The sample code for this procedure would be:

```
void FTP_Counter(void)
{
    struct mapipkt *pkt;
    int fd, pkt_cnt_fid, byte_cnt_fid, buf_fid;
    long byte_counter;
    long pkt_counter;
    int port;
    /* create a flow to monitor the traffic from device: eth0 */
    fd = mapi_create_flow("eth0");
    /* apply a bpf_filter to the flow to monitor only tcp traffic from port 21 */
    mapi_apply_function(fd, "BPF_FILTER", "tcp and port 21");
    /* apply functions to count packets and byte from default port */
    pkt_cnt_fid = mapi_apply_function(fd, "PKT_COUNTER");
    byte_cnt_fid = mapi_apply_function(fd, "BYTE_COUNTER");
    /* apply function to get only the packets that contain the signature */
    mapi_apply_function(fd, "STR_SEARCH", "227 Entering Passive Mode (", 0, 0, 1500);
    /* apply function TO_BUFFER in order to collect the packets and be able to read them */
    buf_fid = mapi_apply_function(fd, "TO_BUFFER");
    mapi_connect(fd);
    while(1)
    {
        pkt = mapi_get_next_pkt(fd, bufid);
        export_port(pkt, &port);
        /*Allocate struct for new flow data*/
        flow_struct = allocate_new_data_struct();
        /* Create flow to monitor the new port number */
        flow_struct.fd = mapi_create_flow("eth0");
        /*Apply filter to the flow to get only the packets to the to-from the new port*/
        mapi_apply_function(flow_struct.fd, "BPF_FILTER", "new port filter");
        /*Apply packet and byte counters*/
        flow_struct.pc = mapi_apply_function(flow_struct.fd, "PKT_COUNTER");
        flow_struct.bc = mapi_apply_function(flow_struct.fd, "BYTE_COUNTER");
        /* Connect flow*/
        mapi_connect(flow_struct.fd);
        /* Add flow struct to a list */
        append_flow(flow_list, flow_struct);
        while(list_not_empty(flow_list))
```

```
        {
            aggregate_counter_results();
        }
        report_results();
    }
}
```

3.2 Peer-to-Peer Systems: The dynamic side of File sharing

During the last years the composition of Internet usage shifted dramatically from mainly Web and e-mail, to file sharing, and the widely used peer-to-peer systems (P2P).

By the term peers, we actually refer to applications that act both like a server and a client, and are able to form an overlay network. This kind of networks are mostly used for file sharing between users, mostly music and video files.

In the first era of P2P networks, applications used well-defined port numbers specific to each network. In this way characterization of P2P traffic was straight forward.

After facing several legal threads from music and movie distribution companies (14; 7), but also facing the need of network administrators to control and minimize bandwidth used by this kind of applications, P2P developers turned to camouflage –even over HTTP well-known port (80)– their generated traffic (15), to circumvent both threads. Current P2P protocols offer the ability to the user to define a port range to be used by the application, Under this circumstances, recognizing P2P traffic has become a challenging action since it demands protocol knowledge and packet inspection in network speeds –which are also increasing dramatically.

Added to this difficulty of distinguishing P2P traffic comes the fact that many peer-to-peer protocols are commercial and so not well documented for their functionality and way of operation. During our study we tried to reverse engineer some of these protocols, in order to understand the way they communicate and reveal some control and data-transfer patterns used. We used two well known tools, tcpdump (20) and Ethereal (4), in order to have a view of the packets exchanged between several applications, and reveal patterns that we can use to monitor traffic. A very helpful study about P2P traffic characterization, for several protocols, was presented in (19).

There are two categories for classifying peer-to-peer traffic. First the user queries for a filename and the peers exchange query information, which are protocol depended. After a query succeeds, the

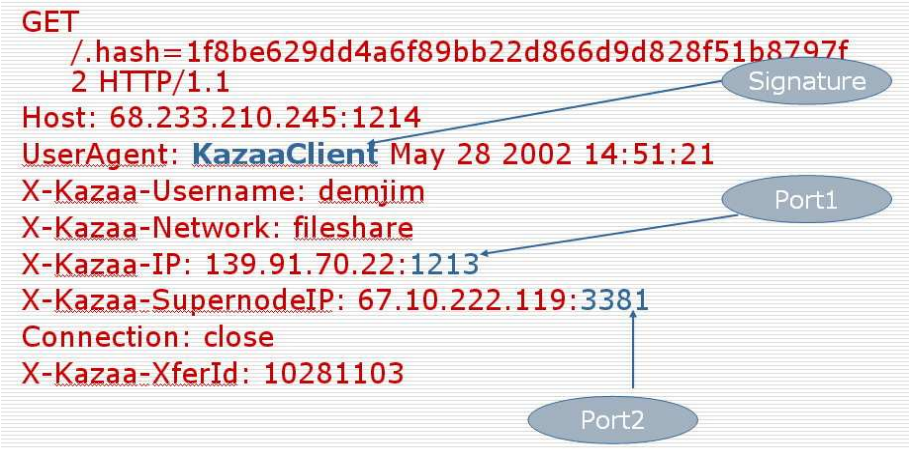

Figure 3.1: Kazza Traffic Header

querying peer contacts the peer who has the file and asks for the file to be transfered. The provider peer may result to several peers when using swarming download, where a file is downloaded in chunks from multiple peers.

3.2.1 Case Study: Fasttrack–The KaZaA Protocol

In this section we present how Fasttrack peer-to-peer protocol works and how we manage to recognize file transfers.

Fasttrack is a very popular protocol. Its clients include the well-known KaZaA Media Desktop (11), Grokster (6) and iMesh (8). By default Fasttrack uses port 1214 for both TCP and UDP traffic, but it is a simple user option to change to another port, even port 80.

To make things a bit more complicated, Fasttrack uses HTTP protocol to transfer the files between the content provider and the requester node. Studies made by (19) revealed that the protocol uses *"GET /.hash="* to request a file with its hash value and also fastrack packets make use of a custom method: *"GIVE"*.

Figure 3.1 shows the HTTP header for a file transfer through the Fasttrack network. The header starts with the HTTP method *"GET"* followed with the hash value for the requested file. The field *"UserAgent"* is filled by every Fasttrack client by the keyword *"KazaaClient"*. This keyword is used as a signature to recognize fastrack traffic in the network and next push it for more inspection for finding the ports used. The two port numbers denoted in figure 3.1 are used by the protocol to transfer control and data packets. The first port, *1213*, is a UDP port used for communication

between the peers and the second port, *3381*, is a TCP port used for transferring the data files for the requested file.

After these observations, we use MAPI to count Fasttrack traffic passing by the network interface. We first apply a *STR_SEARCH* function to the flow, in order to find the packets that contain the keyword *"KazzaClient"*. Continously, we apply a *"TO_BUFFER"* function to collect all the packets containing the keyword. After getting each packet, using *"mapi_get_next_pkt"*, we inspect it to get the ports used. For each port we get, we apply the counting function –*"PKT_COUNTER"* and *"BYTE_COUNTER"*– to measure the traffic consumed by these transfers. By aggregating the traffic for each port, we get the complete portion of traffic used by the protocol.

Following is the sample code for this procedure:

```
void fastrack_counter(void)
{
    struct mapipkt *pkt;
    int fd, pkt_cnt_fid, byte_cnt_fid, buf_fid;
    long byte_counter;
    long pkt_counter;
    int port1, port2;
    /* create a flow to monitor the traffic from device: eth0 */
    fd = mapi_create_flow("eth0");
    /* apply function to get only the packets that contain the signature */
    mapi_apply_function(fd, "STR_SEARCH", "KazaaClient", 0, 0, 1500);
    /* apply function TO_BUFFER in order to collect the packets and be able to read them */
    buf_fid = mapi_apply_function(fd, "TO_BUFFER");
    mapi_connect(fd);
    while(1)
    {
        pkt = mapi_get_next_pkt(fd, bufid);
        export_port(pkt, &port1, &port2);
        /*Allocate struct for new flow data –port1–*/
        flow_struct = allocate_new_data_struct();
        /* Create flow to monitor the new port number */
        flow_struct.fd = mapi_create_flow("eth0");
        /*Apply filter to the flow to get only the packets to the to-from the new port*/
        mapi_apply_function(flow_struct.fd, "BPF_FILTER", "new port1 filter");
        /*Apply packet and byte counters*/
        flow_struct.pc = mapi_apply_function(flow_struct.fd, "PKT_COUNTER");
        flow_struct.bc = mapi_apply_function(flow_struct.fd, "BYTE_COUNTER");
        /* Connect flow*/
        mapi_connect(flow_struct.fd);
```

```
        /* Add flow struct to a list */
        append_flow(flow_list, flow_struct);
        /*Allocate struct for new flow data –port1–*/
        flow_struct = allocate_new_data_struct();
        /* Create flow to monitor the new port number */
        flow_struct.fd = mapi_create_flow("eth0");
        /*Apply filter to the flow to get only the packets to the to-from the new port*/
        mapi_apply_function(flow_struct.fd, "BPF_FILTER", "new port2 filter");
        /*Apply packet and byte counters*/
        flow_struct.pc = mapi_apply_function(flow_struct.fd, "PKT_COUNTER");
        flow_struct.bc = mapi_apply_function(flow_struct.fd, "BYTE_COUNTER");
        /* Connect flow*/
        mapi_connect(flow_struct.fd);
        /* Add flow struct to a list */
        append_flow(flow_list, flow_struct);
        while(list_not_empty(flow_list))
        {
            aggregate_counter_results();
        }
        report_results();
    }
}
```

Chapter 4

Implementation

In this chapter we describe the architecture decided for our application, the considerations made, and the problems we came across.

In order to fully characterize the traffic passing by the network adapter, we split the traffic into three main categories. The first category contains the main Ethernet protocols that are used by the network. The second category represents the main Internet protocols used by applications, and finally the third category reports Internet Application categorization of the traffic. Table 4.1 lists the contents of the three categories we used.

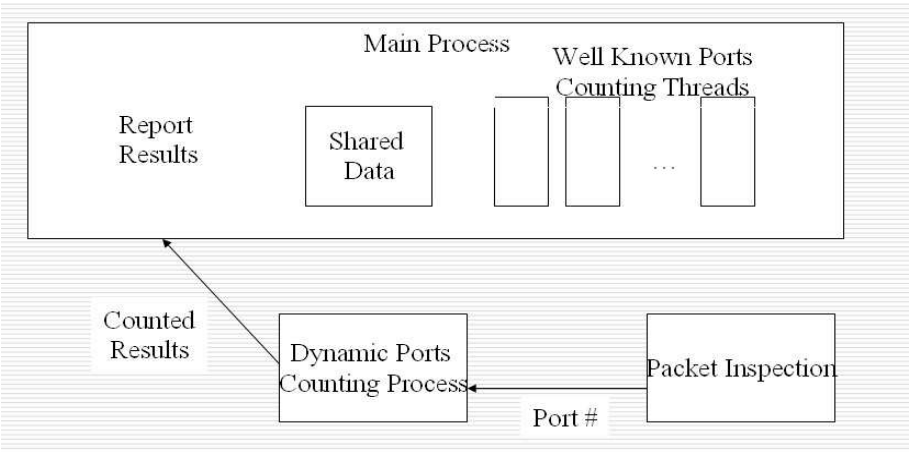

Figure 4.1: System Architecture. The figure presents the basic building blocks of our Application

4.1 Architecture

Figure 4.1 shows the architecture of our application. The application consists of a main process which has the duty of collecting the results from child threads and process and display the results to the user.

In order to collect results from protocols using well-known ports, we create a thread for each protocol. This thread opens a network flow and applies the corresponding *BPF_FILTER*, (e.g. "tcp and port 80" for web traffic) to it in order to get only those packets that represent the selected protocol. Next the thread counts –through MAPI functions *PKT_COUNTER* and *BYTE_COUNTER*– the number of packets and bytes that passed the filter and reports the results to the main process using the share data structures.

On the other-hand, as we explained in chapter 3, collecting monitoring data from applications using dynamic port is a more difficult task. Due to the blocking functionality of the MAPI function, mapi_get_next_pkt() [chapter 2, section 2.3], and because every application uses one socket –Unix Sockets (12)– to communicate with the MAPI daemon, we could came up with an application which blocks every time a thread calls mapi_get_next_pkt().

To face this problem we create two process for each dynamic protocol we measure. The inspection procedure is done in the *"Packet Inspection"* process. This process lookups the packets containing the predefined signature and exports the ports used by that application, –as we explained in section 3.2–. When the ports are decoded, the *inspection* process passes the port numbers to the *counting* process –named *Dynamic Ports Counting Process* in figure 4.1 –. The *counting* process is responsible for reading the counters values from the daemon and passing them, aggregated, to the *main process*, in order to be displayed to the user.

Table 4.1: Table presents the three main categories we used to characterize traffic and the protocols used in each one of the categories

Ethernet Protocols	Internet Protocols	Internet Applications
IP	TCP	KAZAA
IPv6	UDP	DC++
ARP	ICMP	GNUTELLA
RARP	IGMP	BITTORENT
ATALK	IGRP	EDONKEY
AARP	PIM	NAPSTER
DECNET	UDP	FTP
ICO	IGRP	MAIL
STP	PIM	HTTP
IPX	AH	DNS
SCA	ESP	MSN
LAT	VRRP	NETBIOS
MOPDL	EIGRP	REMOTE SHELL
MOPRc		PRINTING(IPP)
NETBEUI		ROUTER PROTOCOL(HSRP)
		RTSP
		GPRS
		KERBEROS
		SUNRPC
		SSDP
		UCP
		WHO
		XDMCP

Chapter 5

Results

To test our application we collected some traces from a PC connected to the Internet. We tried the traffic in our traces to be as closer as the daily traffic of a personal computer, for a home user, which main concerns are Web, e-mail and file sharing. We also modified our tool in order to categorize the traffic using only the default ports reported for each protocol.

The results are presented in Figures 5.1 and 5.2. Both figures present the portion of traffic for several applications.

Figure 5.1 presents the whole identification of traffic when using only the default ports for the applications. We can see from this measurement that about 58.5% of the traffic is categorized as unknown traffic. KaZaA traffic –default port 1214– reports for about 12.5%. HTTP traffic –port 80– counts for about 16.5% of the whole traffic, while netbios traffic –ports 137, 138, 139– aggregates a about 11.75%. Other traffic is reported by the black color which counts for about 0.9%, and aggregates the rest of the applications reported in table 4.1. This provides a proof to the argument, we referred, that well-known port categorization is not enough for monitoring traffic.

In Figure 5.2 we can see the classification of the Internet Application traffic we managed to accomplish with our application. Unknown traffic is reduced to 11.6%, where we can view a reduction by almost 57%. The portion of HTTP and netbios traffic remains the same, since this applications use only standard port numbers. On the other hand the portion reported for KaZaA traffic reaches 59.3%.

Figures 5.3 through 5.8 show results using traces for some other known P2P protocols. Figures 5.3 and 5.4 show results from the gnutella (5) protocol. As we can see unknown traffic has reduced

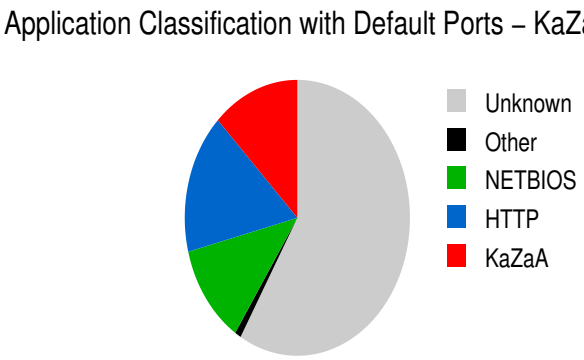

Figure 5.1: Traffic categorization using the default port reported for every application. The portion of unknown traffic is close to 70% of the whole traffic

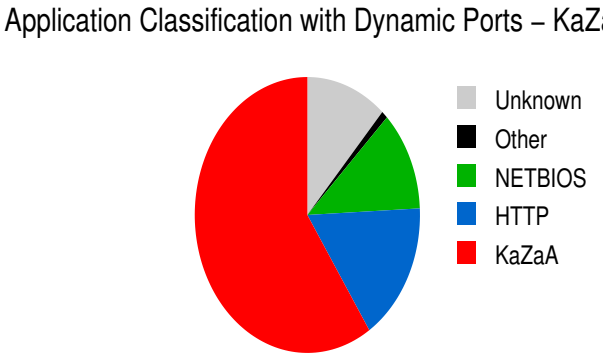

Figure 5.2: Traffic categorization using the tool we developed for dynamic port recognition. As we can see, the portion of unknown traffic decreases close to 11.6%

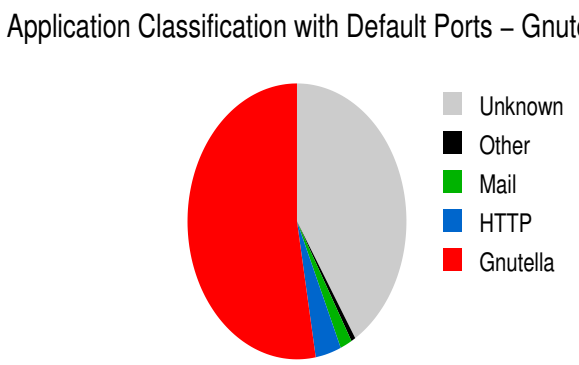

Figure 5.3: Traffic categorization using the default port reported for every application. The portion of unknown traffic is close to 40% of the whole traffic

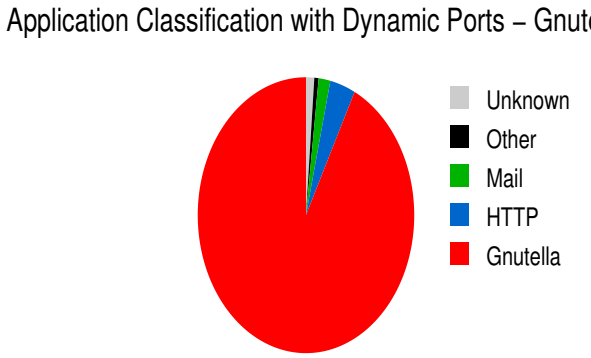

Figure 5.4: Traffic categorization using the tool we developed for dynamic port recognition. The portion of unknown traffic decreases close to 0.6%

by 40% using our categorization tool. Figures 5.5 and 5.6 present results using a trace that contains DC++ (1) traffic. Again the unknown traffic is reduced by 44%. The last two figures. 5.7 and 5.8, show the results taken from a trace that contains traffic from the edonkey protocol (2), witch is reported as the most popular protocol (18). In this case our tool manages to reduce the portion of unknown traffic almost to 0%.

It is important to mention that unknown traffic for Ethernet and Internet Protocols reported in table 4.1, is limited to 0%.

As we can see from the results, our application manages to count traffic that was consider unknown and remained unaccounted for a long period of time. We do not claim that we manage to count all the kinds of traffic that could be passing through any network and provide the catego-

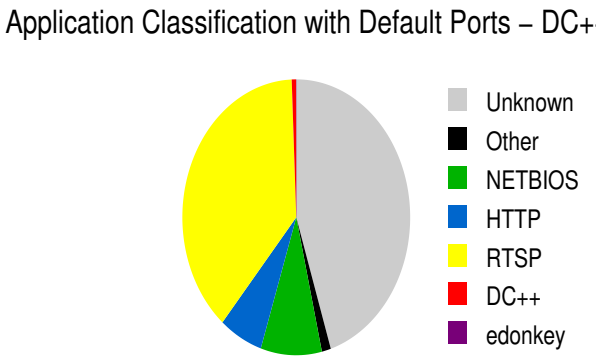

Figure 5.5: Traffic categorization using the default port reported for every application. The portion of unknown traffic is close to 45% of the whole traffic

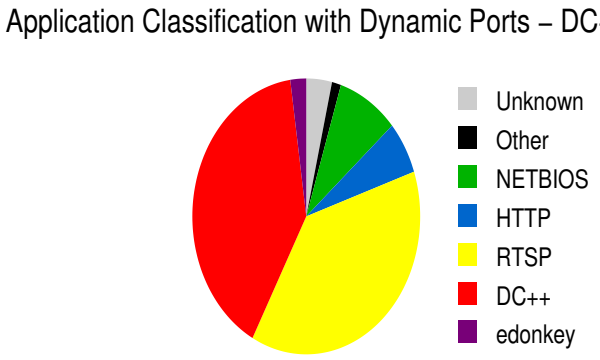

Figure 5.6: Traffic categorization using the tool we developed for dynamic port recognition. The portion of unknown traffic decreases close to 1.2%

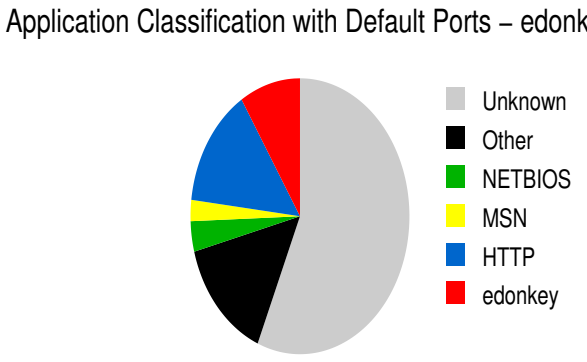

Figure 5.7: Traffic categorization using the default port reported for every application. The portion of unknown traffic is close to 55% of the whole traffic

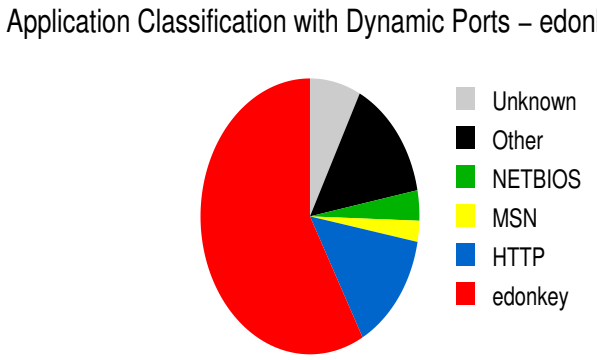

Figure 5.8: Traffic categorization using the tool we developed for dynamic port recognition. The portion of unknown traffic decreases close to 7.5%

rization knowledge for every single packet but we can claim that we took a step forward to better understanding of the Internet.

Chapter 6

Conclusions and Future Work

In the previous chapters we described the need for advanced network monitoring, because of the recent traffic masking trends implemented by application developers. These trends are mostly applied by the use of dynamic port generation and communication through dynamic port numbers. A great number of applications using this trend comes from the field of file-sharing application, mostly known as P2P systems.

We studied the way these dynamic ports are communicated between users and we developed a tool for monitoring these kind of applications. Implementing our tool using the Monitoring API (MAPI), developed within the IST project SCAMPI, we manage to reduce the portion of unknown traffic close to 10% of the whole traffic passing through out network interface.

Work is still to be done, in order to add more protocols in our inspection methods, and to study applications beyond file sharing like, audio and video traffic, which also produce a great portion of traffic.

Bibliography

[1] DC++. http://dcplusplus.sourceforge.net

[2] Edonkey. http://www.edonkey2000.com

[3] E.P. Markatos. Tracing a large-scale peer to peer system: an hour in the life of gnutella. In Proceedings of the CCGrid 2002: the second IEEE International Symposium on Cluster Computing and the Grid, May 2002.

[4] Ethereal. www.ethereal.org

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

[6] Grokster. http://www.grokster.com

[7] Hollywood Vs. P2P. http://www.theregister.co.uk/2002/08/13/hollywoods_private_war_for_social

[8] iMesh. http://www.imesh.com

[9] IST-SCAMPI project. www.ist-scampi.org

[10] J. Postel and J. Reynolds. File Transfer Protocol (FTP). RFC-959 (ftp://ftp.rfc-editor.org/in-notes/rfc959.txt), 1985

[11] Kazaa Media Desktop. http://www.kazaa.com

[12] M. Polychronakis, K. G. Anagnostakis, E. P. Markatos, Arne Øslebø. Design of an Application Programming Interface for IP Network Monitoring. Proceedings of the 9th IEEE/IFIP Network Operations and Management Symposium (NOMS2004), 19-23 April 2004, Seoul, Korea.

[13] MAPI official homepage. http://mapi.uninett.no/

[14] Metallica Vs. Napster. http://archives.cnn.com/2001/LAW/02/12/napster.decision.05

[15] N. Leibowitz, M. Ripeanu, and A. Wierzbicki. Deconstructing the Kazaa Network. In 3rd IEEE Workshop on Internet Applications (WIAPP'03), 2003.

[16] Oliver Spatscheck. How to monitor network traffic 5 gbit/sec at a time. In Workshop on Management and Processing of Data Streams, 2003.

[17] S. Sen and J. Wang. Analyzing Peer-to-Peer Traffic Across Large Networks. In ACM SIGCOMM Internet Measurement Workshop, 2002

[18] Slyck - File Sharing News and Info. http://www.slyck.com

[19] T. Karagiannis, A. Broido, N. Brownlee, k. claffy, M. Faloutsos. File-sharing in the Internet: A characterization of P2P traffic in the backbone. Technical report. November, 2003.

[20] tcpdump. www.tcpdump.org

[21] The SCAMPI Consortium. SCAMPI Architecture and Components: SCAMPI Deliverable D1.2, 2002. Available from http://www.ist-scampi.com