Strumenti di analisi della rete

Federico De Meo, Davide Quaglia

Introduzione agli analizzatori di rete

Esistono strumenti SW che consentono di analizzare i pacchetti che arrivano alla propria interfaccia di rete:

- **tcpdump (**http://www.tcpdump.org/tcpdump_man.html) storico tool in linea di comando per Linux
- **wireshark** (http://www.wireshark.org/docs/) nuovo tool con interfaccia grafica disponibile per Linux, Windows e Mac
- **ethereal (**http://www.ethereal.com/docs/man-pages/tethereal.1.html) versione in linea di comando di Wireshark
- **analyzer** (**http://analyzer.polito.it/**) sviluppato al Politecnico di Torino per Windows
- **windump** (**http://www.winpcap.org/windump**) per Windows.

Questi tool di analisi si basano tutti sulla libreria C chiamata ***libpcap***, reperibile al sito http://www.tcpdump.org (tcpdump è lo sniffer per eccellenza che sfrutta la libpcap).
Le principali funzioni di questa libreria sono la possibilità di cercare e trovare interfacce di rete, gestire potenti filtri di cattura, analizzare ciascun pacchetto, gestire errori e statistiche di cattura.

2.1 Scaricamento e installazione

I tool si possono scaricare liberamente dalle rispettive pagine web o probabilmente sono presenti già nell'installazione della propria distribuzione Linux.

ATTENZIONE: per poter utilizzare le funzionalità di cattura di questi tool in ambiente Linux bisogna essere autenticati come utente *root* o aver installato il tool con *setuid* a root. In ogni caso questi tool possono essere utilizzati per analizzare catture precedentemente effettuate da utente root e salvate su file.
In questa esercitazione non cattureremo del traffico dalla rete ma analizzeremo del traffico precedentemente catturato.

2.2 Alcuni concetti sullo *sniffing*

Sniffing in **reti non-switched**: In questo tipo di reti ethernet il mezzo trasmissivo è condiviso e quindi tutte le schede di rete dei computer nella rete locale ricevono tutti i pacchetti, anche quelli destinati ad altri, selezionando i propri a seconda dell'indirizzo ***MAC*** (indirizzo hardware specifico della scheda di rete). Lo sniffing in questo caso consiste nell'impostare sull'interfaccia di rete la cosiddetta **modalità promiscua**, che disattiva il “filtro hardware” basato sul MAC permettendo al sistema l'ascolto di tutto il traffico passante sul cavo. Esempio di rete non switched è la **rete WiFi**.

Sniffing in **reti ethernet switched**: In questo caso l'apparato centrale della rete, definito switch, si preoccupa, dopo un breve transitorio, di inoltrare su ciascuna porta solo il traffico destinato ai dispositivi collegati a quella porta; ciascuna interfaccia di rete riceve, quindi solo i pacchetti

destinati al proprio indirizzo, i pacchetti multicast e quelli broadcast. L'impostazione della modalità promiscua è quindi insufficiente per poter intercettare il traffico in una rete gestita da switch. Un metodo per poter ricevere tutto il traffico dallo switch da una porta qualunque è il **MAC flooding**. Tale tecnica consiste nell'inviare ad uno switch pacchetti appositamente costruiti per riempire la *CAM table* dello switch di indirizzi MAC fittizi. Questo attacco costringe lo switch ad entrare in una condizione detta di *fail open* che lo fa comportare come un hub, inviando così gli stessi dati a tutti gli apparati ad esso collegati.

Utilizzo di *wireshark*

Alcune caratteristiche:

- i dati possono essere acquisiti direttamente dall'interfaccia di rete (“from the wire”) oppure possono essere letti su un file di cattura precedente
- i dati possono essere catturati dal vivo da reti Ethernet, WiFi, ADSL, ecc.
- i dati di rete catturati possono essere esplorati nelle loro parti tramite un'interfaccia grafica
- i filtri di visualizzazione possono essere usati per colorare o visualizzare selettivamente le informazioni sommarie sui pacchetti
- i protocolli di comunicazione possono essere scomposti, in quanto wireshark riesce a “comprendere” la struttura dei diversi protocolli di rete, quindi è in grado di visualizzare incapsulamenti e campi singoli, ed di interpretare il loro significato
- è possibile studiare le statistiche di una connessione TCP e di estrarne il contenuto.

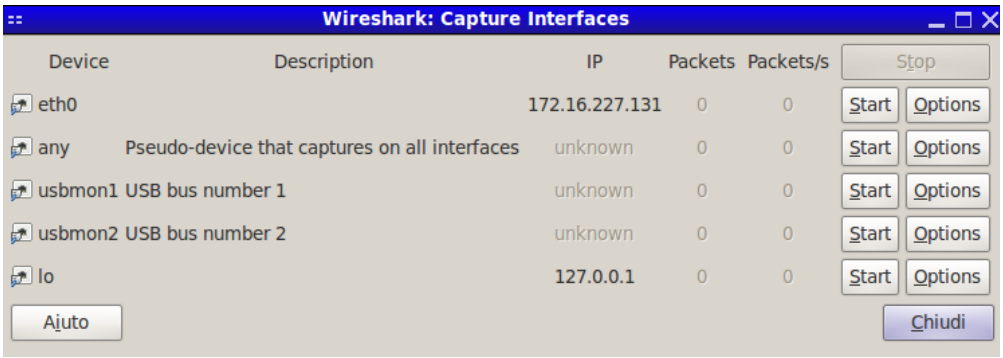
Per lanciare l'applicazione, digitare su riga di comando:

```
$ wireshark
```

oppure, essendo wireshark un'applicazione grafica, basta lanciarlo cliccando sull'apposita icona:

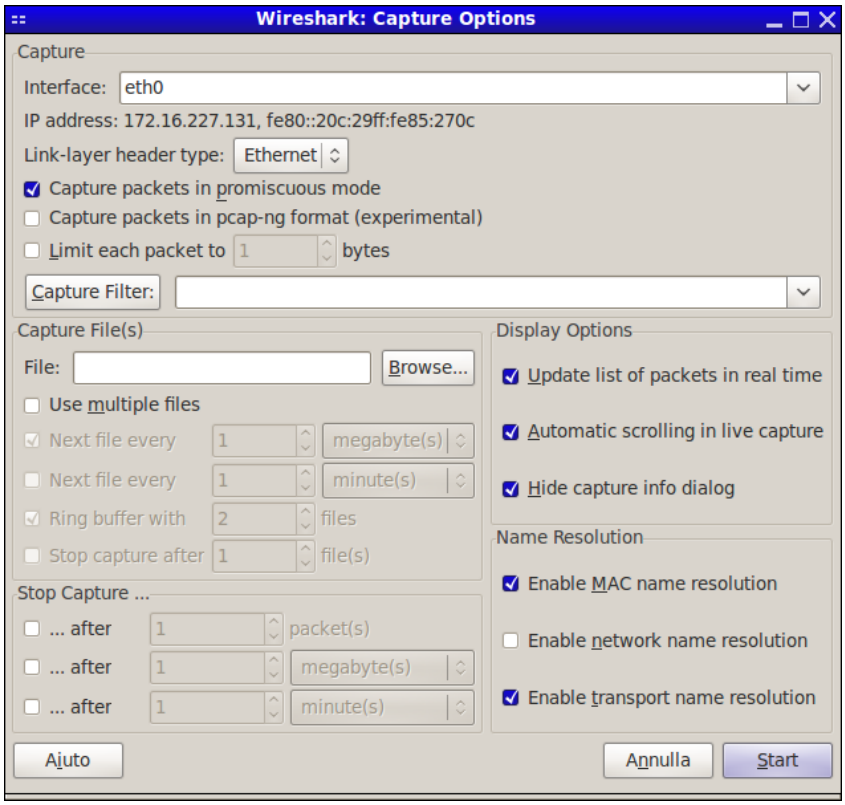
si ricordi che è necessario essere l'utente *root* se si intende fare una cattura diretta dalle interfacce di rete, ma in questa esercitazione useremo un file con delle catture già fatte, quindi non si necessitano i privilegi dell'utente root.

Per avviare la cattura dei pacchetti è necessario specificare da quale interfaccia si vuole effettuare la cattura. Per fare ciò aprire il menu `Capture/Interfaces`



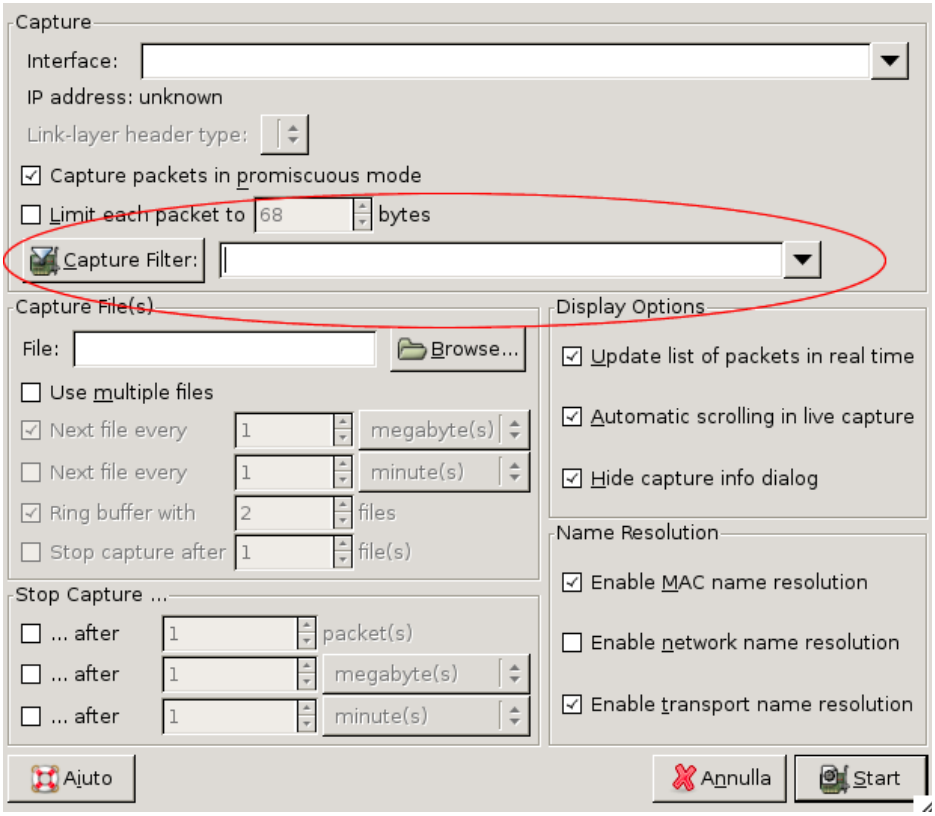
verrà chiesto quale interfaccia di rete utilizzare per la cattura.

Solitamente, in linux, la prima interfaccia di rete ha come nome `eth0` mentre in Windows ha il nome del produttore della scheda.
Una volta individuata l'interfaccia, premendo il tasto `Options` è possibile dare diverse impostazioni, tra le quali anche i filtri da utilizzare per la cattura.

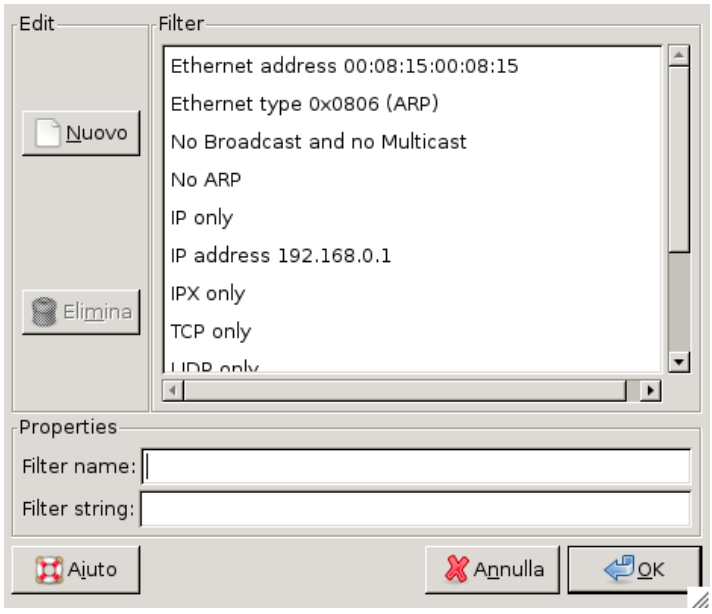


Premendo `Start` si avvia la cattura e viene visualizzata una finestra che mostra le statistiche di cattura. Per terminare la cattura (se non si è impostato un limite in pacchetti, secondi o byte) si usa il comando `Stop` dal menu `Capture`.

E' possibile limitare la cattura ai soli pacchetti che rispettano certi requisiti impostando un filtro di cattura nella finestra di `Capture Options` precedentemente vista, prendendo sul tasto `Capture Filter.`

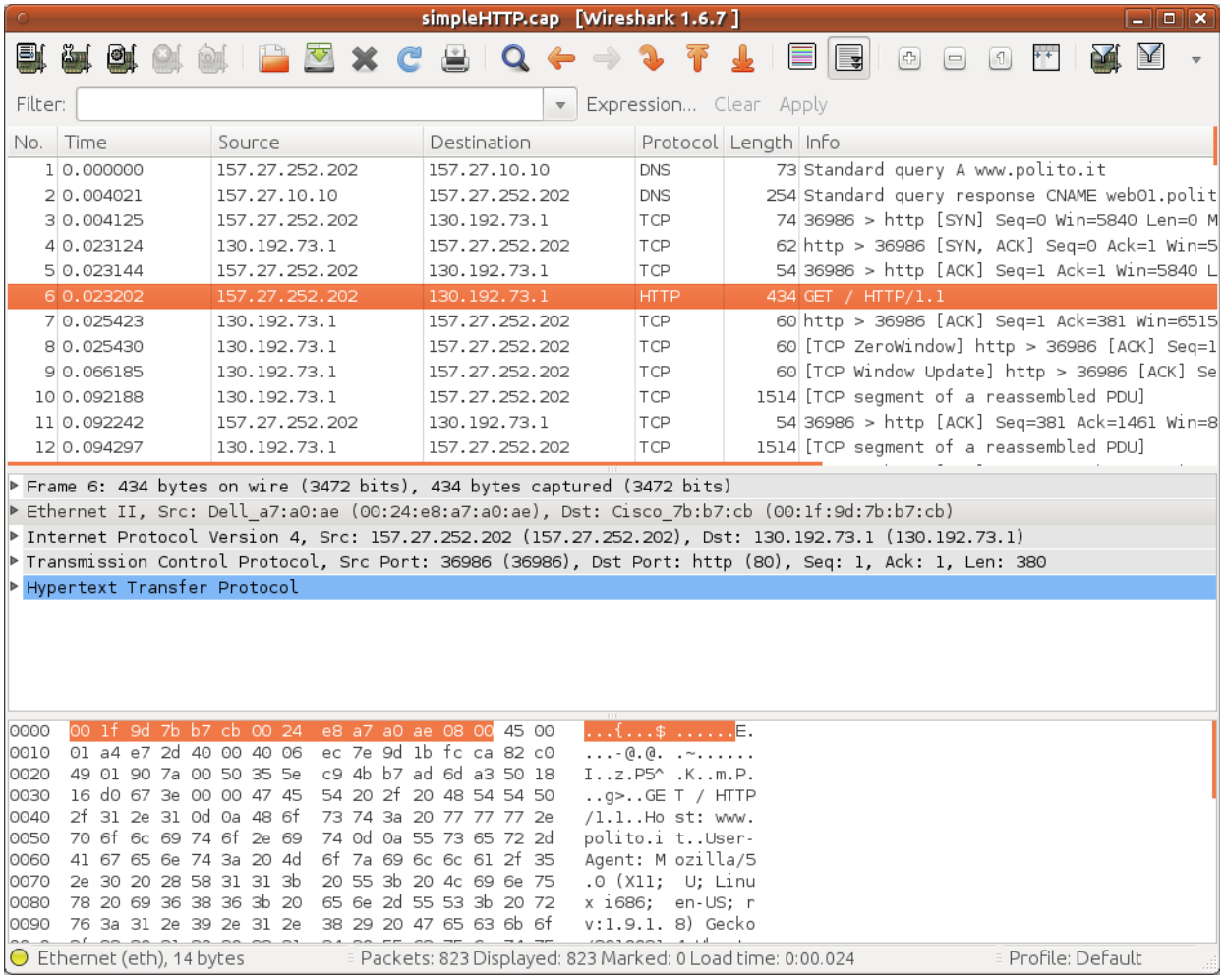


Nella sezione apposita è possibile scrivere l'espressione del filtro (trovate la sintassi nell'appendice A) oppure selezionare un filtro pre-esistente cliccando sull'apposito bottone e al limite modificarlo. I filtri di cattura programmano la scheda di rete al fine di catturare solo determinati pacchetti; tali filtri si usano quando la quantità di pacchetti che passano sul tratto di rete osservato è tale che se tutti i pacchetti venissero passati alla CPU le sue prestazioni sarebbero compromesse.



Al termine della cattura la finestra principale di wireshark mostra i dati catturati. Tale finestra è divisa in tre parti, nell'ordine:

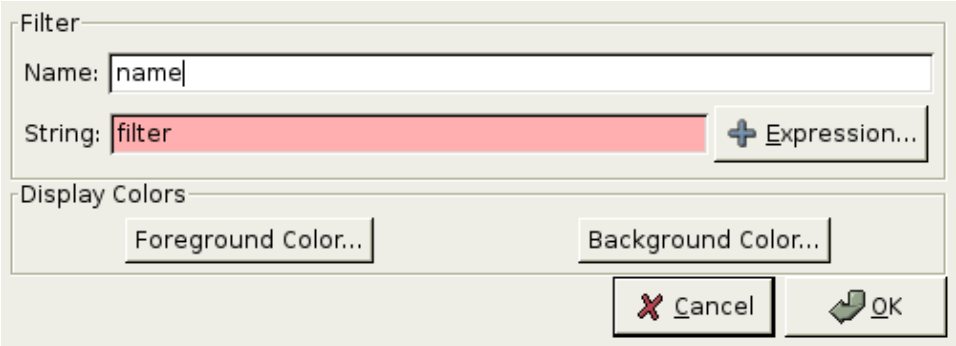
- Tabella dei pacchetti catturati;
- Vista sulla incapsulazione dei protocolli del pacchetto selezionato nella tabella;
- Vista in versione binaria dei dati del pacchetto selezionato nella tabella.



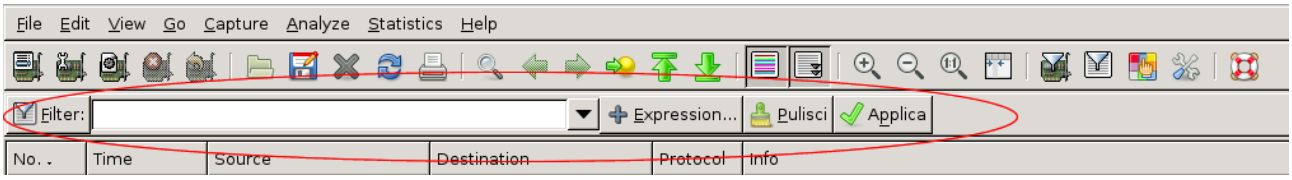
E' anche possibile vedere un sommario sulle statistiche scegliendo il menu `Statistics/Summary`.

E' possibile migliorare la visualizzazione dei vari pacchetti nella tabella principale colorando le righe in base al tipo di protocolli o indirizzi coinvolti. I filtri di coloramento si possono impostare nel menu `View/Colouring rules.`

E' possibile creare nuovi filtri di colori con il bottone `New` che apre una finestra in cui si può impostare nome, regole e colori. Le regole si impostano con un linguaggio diverso da quello dei filtri di cattura (si noti il pulsante Expression che semplifica la scrittura delle regole combinando campi dei pacchetti, valori e operatori logici).



E' possibile creare un filtro per limitare il numero di pacchetti visualizzati in una cattura già avvenuta, per fare questo, si utilizza la barra dei filtri presente nella schermata principale.



Tramite il tasto `Filter` è possibile specificare un filtro esistente o crearne di nuovi e per la stesura dei filtri ci aiuta il tasto `Expression` che fornisce un tool automatico di stesura filtri simile a quello per la colorazione delle righe.

Per quanto riguarda TCP, selezionando un pacchetto TCP nella finestra principale, è possibile seguire l'intero flusso di dati di quella "conversazione" mediante la voce `Analyze/Follow TCP Stream` e studiare l'andamento di alcuni parametri del protocollo mediante la voce `Statistics/TCP stream graph.`

Esercitazione

Per motivi di sicurezza non è possibile analizzare il traffico reale presente nella rete del laboratorio e quindi, per questa esercitazione, useremo del traffico precedentemente catturato.

Esercizio 1

Occorre avviare wireshark, aprire il menu `File/Open` e selezionare il file `capture.cap`.

Si prenda in considerazione il pacchetto numero 9 e si risponda alle seguenti domande/esercizi:

1. che tipo di protocollo di livello datalink è usato? Come fa Wireshark a capirlo?
2. disegnare la PDU di livello datalink indicando il valore dei vari campi
3. qual è il MAC sorgente ? Di che tipo è: unicast o broadcast ?
4. qual è il MAC destinazione ? Di che tipo è: unicast o broadcast ?
5. che tipo di protocollo di livello network è usato? Come fa Wireshark a capirlo?
6. qual è la lunghezza dell'header IP?
7. quali sono gli indirizzi IP sorgente e destinazione?
8. che tipo di protocollo di livello trasporto è contenuto in IP? Come fa Wireshark a capirlo?

9. quali sono le porte sorgente e destinazione a livello trasporto?
10. creare un filtro per visualizzare solo i pacchetti che hanno ARP come protocollo (suggerimento: basta scrivere “arp” nella barra Filter sotto la toolbar; si ricordi di premere su APPLY dopo aver scritto “arp”)
11. dopo aver applicato il filtro precedente qual è la percentuale di pacchetti che rimangono visualizzati rispetto al totale ? (suggerimento: vedere entrambi i valori nella barra di stato in basso)
12. creare un filtro per visualizzare solo i pacchetti che hanno sorgente MAC 00:22:19:c7:2b:ee (suggerimento: usare l'editor di espressioni; la categoria da selezionare è Ethernet; per l'indirizzo MAC usare la notazione esadecimale con i due punti come separatori; si ricordi di premere su APPLY dopo aver creato l'espressione)
13. dopo aver applicato il filtro precedente qual è la percentuale di pacchetti che rimangono visualizzati rispetto al totale ? (suggerimento: vedere entrambi i valori nella barra di stato in basso)
14. creare un filtro per visualizzare solo i pacchetti che hanno destinazione MAC broadcast (suggerimento: nell'editor di espressioni la categoria da usare è Ethernet; per l'indirizzo MAC usare la notazione esadecimale con i due punti come separatori; si ricordi di premere su APPLY dopo aver creato l'espressione)
15. dopo aver applicato il filtro precedente qual è la percentuale di pacchetti che rimangono visualizzati rispetto al totale ? Sono molti ? Perché ?

Esercizio 2

Occorre aprire il menu `File/Open` e selezionare il file `simpleHTTP.cap`.

1. Colorare di rosso tutti i pacchetti che contengono UDP e di verde tutti i pacchetti che contengono TCP (suggerimento: nell'editor delle regole di colorazione è sufficiente portare in alto due regole già esistenti e modificarle per cambiarne i colori di sfondo)
2. Cosa contengono i primi due pacchetti della sessione di cattura?
   - IP sorgente, IP destinazione
   - tipo di protocollo di trasporto
   - tipo di protocollo di livello applicazione. Come fa Wireshark a capirlo?
   - messaggio contenuto nel payload di livello applicazione
3. Prendere in considerazione il pacchetto 3
   - IP sorgente, IP destinazione
   - tipo di protocollo di trasporto
   - IP sorgente e destinazione sono in qualche modo collegati con i messaggi scambiati a livello applicazione nei primi due pacchetti ? E' possibile fare delle ipotesi su cosa serve il protocollo di livello applicazione dei primi due pacchetti ?
4. Descrivere la struttura del pacchetto 6
   - IP sorgente, IP destinazione
   - tipo di protocollo di trasporto

- tipo di protocollo di livello applicazione
- Perché prima della trasmissione del primo messaggio HTTP c'è lo scambio di tre pacchetti puramente TCP? Quali sono i flag settati nell'header TCP di questi tre pacchetti?

5. creare un filtro per visualizzare solo i pacchetti TCP (compresi i pacchetti HTTP) e determinarne il numero.
6. creare un filtro per visualizzare solo i pacchetti TCP (esclusi i pacchetti HTTP) e determinarne il numero.
   - Qual è la percentuale sul totale dei pacchetti TCP trovata al punto 5?
   - A cosa servono tali pacchetti?
   - Se il protocollo DNS dei pacchetti 1 e 2 avesse usato il protocollo TCP, quanti pacchetti IP sarebbero stati generati? Sarebbe stato utile?
7. Selezionare il pacchetto 3 e seguire lo stream TCP col comando da menu `Analyze/Follow TCP Stream`
   - cosa si può leggere?
   - qual è il messaggio contenuto nel payload della PDU di livello applicazione?

Esercizio 3

Occorre aprire il file `busyNetwork.cap` con wireshark.

1. Elencare i protocolli di livello applicazione entrano in azione in questa cattura classificandoli in base al livello trasporto utilizzato.
2. Provare ad analizzare diversi stream TCP con sopra diversi protocolli di livello applicazione.
3. Che differenza c'e' tra il contenuto trasmesso in una connessione TCP per il protocollo FTP e quello trasmesso per il protocollo SSH?

APPENDICE A. Sintassi dei filtri di cattura

La descrizione della sintassi è presa dalla man page di tcpdump.

An expression selects which packets will be dumped. If no *expression* is given, all packets on the net will be dumped. Otherwise, only packets for which *expression* is `true' will be dumped.

The expression consists of one or more primitives. Primitives usually consist of an id (name or number) preceded by one or more qualifiers. There are three different kinds of qualifier:

type qualifiers say what kind of thing the id name or number refers to. Possible types are **host**, **net** and **port**. E.g., `host foo', `net 128.3', `port 20'. If there is no type qualifier, **host** is assumed.

dir qualifiers specify a particular transfer direction to and/or from *id*. Possible directions are **src**, **dst**, **src or dst** and **src and dst**. E.g., `src foo', `dst net 128.3', `src or dst port ftp-data'. If there is no dir qualifier, **src or dst** is assumed. For `null' link layers (i.e. point to point protocols such as slip) the **inbound** and **outbound** qualifiers can be used to specify a desired direction.

proto qualifiers restrict the match to a particular protocol. Possible protos are: **ether**, **fddi**, **tr**, **ip**, **ip6**, **arp**, **rarp**, **decnet**, **tcp** and **udp**. E.g., `ether src foo', `arp net 128.3', `tcp port 21'. If there is no proto

qualifier, all protocols consistent with the type are assumed. E.g., `src foo' means `(ip or arp or rarp) src foo' (except the latter is not legal syntax), `net bar' means `(ip or arp or rarp) net bar' and `port 53' means `(tcp or udp) port 53'.

[`fddi' is actually an alias for `ether'; the parser treats them identically as meaning ``the data link level used on the specified network interface." FDDI headers contain Ethernet-like source and destination addresses, and often contain Ethernet-like packet types, so you can filter on these FDDI fields just as with the analogous Ethernet fields. FDDI headers also contain other fields, but you cannot name them explicitly in a filter expression.

Similarly, `tr' is an alias for `ether'; the previous paragraph's statements about FDDI headers also apply to Token Ring headers.]

In addition to the above, there are some special `primitive' keywords that don't follow the pattern: **gateway**, **broadcast**, **less**, **greater** and arithmetic expressions. All of these are described below.

tives. E.g., `host foo and not port ftp and not port ftp-data'. To save typing, identical qualifier lists can be omitted. E.g., `tcp dst port ftp or ftp-data or domain' is exactly the same as `tcp dst port ftp or tcp dst port ftp-data or tcp dst port domain'.

Allowable primitives are:

dst host *host*

True if the IPv4/v6 destination field of the packet is *host*, which may be either an address or a name.

src host *host*

True if the IPv4/v6 source field of the packet is *host*.

host *host*

True if either the IPv4/v6 source or destination of the packet is *host*. Any of the above host expressions can be prepended with the keywords, **ip**, **arp**, **rarp**, or **ip6** as in:

ip host *host*

which is equivalent to:

ether proto *ip* **and host** *host*

If *host* is a name with multiple IP addresses, each address will be checked for a match.

ether dst *ehost*

True if the ethernet destination address is *ehost*. *Ehost* may be either a name from /etc/ethers or a number (see **ethers(3N)** for numeric format).

ether src *ehost*

True if the ethernet source address is *ehost*.

ether host *ehost*

True if either the ethernet source or destination address is *ehost*.

gateway *host*

True if the packet used *host* as a gateway. I.e., the ethernet source or destination address was *host* but neither the IP source nor the IP destination was *host*. *Host* must be a name and must be found both by the machine's host-name-to-IP-address resolution mechanisms (host name file, DNS, NIS, etc.) etc.). (An equivalent expression is

ether host *ehost* **and not host** *host*

which can be used with either names or numbers for *host / ehost*.) This syntax does not work in IPv6-enabled configuration at this moment.

dst net *net*

True if the IPv4/v6 destination address of the packet has a network number of *net*. *Net* may be either a name from /etc/networks or a network number (see *networks(4)* for details).

src net *net*

True if the IPv4/v6 source address of the packet has a network number of *net*.

net *net*

True if either the IPv4/v6 source or destination address of the packet has a network number of *net*.

net *net* **mask** *netmask*

True if the IP address matches *net* with the specific *netmask*. May be qualified with **src**

or **dst**. Note that this syntax is not valid for IPv6 *net*.

net *net*/*len*

True if the IPv4/v6 address matches *net* with a netmask *len* bits wide. May be qualified with **src** or **dst**.

dst port *port*

True if the packet is ip/tcp, ip/udp, ip6/tcp or ip6/udp and has a destination port value of *port*. The *port* can be a number or a name used in /etc/services (see **tcp(4P)** and **udp(4P)**). If a name is used, both the port number and protocol are checked. If a number or ambiguous name is used, only the port number is checked (e.g., **dst port 513** will print both tcp/login traffic and udp/who traffic, and **port domain** will print both tcp/domain and udp/domain traffic).

src port *port*

True if the packet has a source port value of *port*.

True if either the source or destination port of the packet is *port*. Any of the above port expressions can be prepended with the keywords, **tcp** or **udp**, as in:

tcp src port *port*

which matches only tcp packets whose source port is *port*.

less *length*

True if the packet has a length less than or equal to *length*. This is equivalent to:

len <= *length***.**

greater *length*

True if the packet has a length greater than or equal to *length*. This is equivalent to:

len >= *length***.**

ip proto *protocol*

True if the packet is an IP packet (see **ip(4P)**) of protocol type *protocol*. *Protocol* can be a number or one of the names *icmp*, *icmp6*, *igmp*, *igrp*, *pim*, *ah*, *esp*, *vrrp*, *udp*, or *tcp*. Note that the identifiers *tcp*, *udp*, and *icmp* are also keywords and must be escaped via backslash (\), which is \\ in the C-shell. Note that this primitive does not chase the protocol header chain.

ip6 proto *protocol*

True if the packet is an IPv6 packet of protocol type *protocol*. Note that this primitive does not chase the protocol header chain.

ip6 protochain *protocol*

True if the packet is IPv6 packet, and contains protocol header with type *protocol* in its protocol header chain. For example,

ip6 protochain 6

matches any IPv6 packet with TCP protocol header in the protocol header chain. The packet may contain, for example, authentication header, routing header, or hop-by-hop option header, between IPv6 header and TCP header. The BPF code emitted by this primi-

tive is complex and cannot be optimized by BPF optimizer code in *tcpdump*, so this can be somewhat slow.

ip protochain *protocol*

Equivalent to **ip6 protochain** *protocol*, but True if the packet is an ethernet broadcast packet. The *ether* keyword is optional.

ip broadcast

True if the packet is an IP broadcast packet. It checks for both the all-zeroes and all-ones broadcast conventions, and looks up the local subnet mask.

ether multicast

True if the packet is an ethernet multicast packet. The *ether* keyword is optional. This is shorthand for `**ether[0] & 1 != 0**'.

ip multicast

True if the packet is an IP multicast packet.

ip6 multicast

True if the packet is an IPv6 multicast packet.

ether proto *protocol*

True if the packet is of ether type *protocol*. *Protocol* can be a number or one of the names *ip, ip6, arp, rarp, atalk, aarp, decnet, sca, lat, mopdl, moprc, iso, stp, ipx,* or *netbeui*. Note these identifiers are also keywords and must be escaped via backslash

(\).

[In the case of FDDI (e.g., `**fddi protocol arp**') and Token Ring (e.g., `**tr protocol arp**'), for most of those protocols, the protocol identification comes from the 802.2 Logical Link Control (LLC) header, which is usually layered on top of the FDDI or Token Ring header.

When filtering for most protocol identifiers on FDDI or Token Ring, *tcpdump* checks only the protocol ID field of an LLC header in so-called SNAP format with an Organizational Unit Identifier (OUI) of 0x000000, for encapsulated Ethernet; it doesn't check whether the packet is in SNAP format with an OUI of 0x000000.

The exceptions are *iso*, for which it checks the DSAP (Destination Service Access Point) and SSAP (Source Service Access Point) fields of the LLC header, *stp* and *netbeui*, packet with an OUI of 0x080007 and the Appletalk etype.

In the case of Ethernet, *tcpdump* checks the Ethernet type field for most of those protocols; the exceptions are *iso*, *sap*, and *netbeui*, for which it checks for an 802.3 frame and then checks the LLC header as it does for FDDI and Token Ring, *atalk*, where it checks both for the Appletalk etype in an Ethernet frame and for a SNAP-format packet as it does for FDDI and Token Ring, *aarp*,

where it checks for the Appletalk ARP etype in either an Ethernet frame or an 802.2 SNAP frame with an OUI of 0x000000, and *ipx*, where it checks for the IPX etype in an Ethernet frame, the IPX DSAP in the LLC header, the 802.3 with no LLC header encapsulation of IPX, and the IPX etype in a SNAP frame.]

decnet src *host*

True if the DECNET source address is *host*, which may be an address of the form ``10.123", or a DECNET host name. [DECNET host name support is only available on Ultrix systems that are configured to run DECNET.]

decnet dst *host*

True if the DECNET destination address is *host*.

decnet host *host*

True if either the DECNET source or destination address is *host*.

ip, **ip6**, **arp**, **rarp**, **atalk**, **aarp**, **decnet**, **iso**, **stp**, **ipx**, *netbeui*

Abbreviations for:

ether proto *p*

where *p* is one of the above protocols.

lat, **moprc**, **mopdl**

Abbreviations for:

ether proto *p*

where *p* is one of the above protocols. Note that *tcpdump* does not currently know how to

parse these protocols.

vlan *[vlan_id]*

True if the packet is an IEEE 802.1Q VLAN packet. If *[vlan_id]* is specified, only encountered in *expression* changes the decoding offsets for the remainder of *expression* on the assumption that the packet is a VLAN packet.

tcp, **udp**, **icmp**

Abbreviations for:

ip proto *p* **or ip6 proto** *p*

where *p* is one of the above protocols.

iso proto *protocol*

True if the packet is an OSI packet of protocol type *protocol*. *Protocol* can be a number or one of the names *clnp*, *esis*, or *isis*.

clnp, **esis**, **isis**

Abbreviations for:

iso proto *p*

where *p* is one of the above protocols. Note that *tcpdump* does an incomplete job of parsing these protocols.

expr relop expr

True if the relation holds, where *relop* is one of >, <, >=, <=, =, !=, and *expr* is an arithmetic expression composed of integer constants (expressed in standard C syntax), the normal binary operators [+, -, *, /, &, |], a length operator, and special packet data accessors. To access data inside the

packet, use the following syntax:

proto **[** *expr* **:** *size* **]**

Proto is one of **ether, fddi, tr, ip, arp, rarp, tcp, udp, icmp** or **ip6**, and indicates the protocol layer for the index operation. Note that *tcp*, *udp* and other upper-layer protocol types only apply to IPv4, not IPv6 (this will be fixed in the future). The byte offset, relative to the indicated protocol layer, is given by *expr*. *Size* is optional and indicates the number of bytes in the field of interest; it can be either one, two, or four, and defaults to one. The length operator, indicated by the keyword **len**, gives the length of the packet.

For example, `**ether[0] & 1 != 0**' catches all multicast traffic. The expression `**ip[0] & 0xf != 5**' catches all IP packets with options. The expression `**ip[6:2] & 0x1fff = 0**' catches only unfragmented datagrams and frag zero of fragmented datagrams. This always means the first byte of the TCP *header*, and never means the first byte of an intervening fragment.

Some offsets and field values may be expressed as names rather than as numeric values. The following protocol header field offsets are available: **icmptype** (ICMP type field), **icmpcode** (ICMP code field), and **tcpflags** (TCP flags field).

The following ICMP type field values are available: **icmp-echoreply**, **icmp-unreach**,

icmp-sourcequench, **icmp-redirect**, **icmp-echo**, **icmp-routeradvert**, **icmp-routersolicit**, **icmp-timxceed**, **icmp-paramprob**, **icmp-tstamp**, **icmp-tstampreply**, **icmp-ireq**, **icmp-ireqreply**, **icmp-maskreq**, **icmp-maskreply**.

The following TCP flags field values are available: **tcp-fin**, **tcp-syn**, **tcp-rst**, **tcp-push**, **tcp-push**, **tcp-ack**, **tcp-urg**.

Primitives may be combined using:

A parenthesized group of primitives and operators (parentheses are special to the Shell and must be escaped).

Negation (`**!**' or `**not**').

Concatenation (`**&&**' or `**and**').

Alternation (`||' or `**or**').

Negation has highest precedence. Alternation and concatenation have equal precedence and associate left to right. Note that explicit **and** tokens, not juxtaposition, are now required for concatenation.

If an identifier is given without a keyword, the most recent keyword is assumed. For example,

not host vs and ace

is short for

not host vs and host ace

which should not be confused with

not (host vs or ace)

Expression arguments can be passed to *tcpdump* as either a single argument or as multiple arguments, whichever is more convenient. Generally, if the expression contains Shell metacharacters, it is before being parsed.

EXAMPLES

To print all packets arriving at or departing from *sundown*:

tcpdump host sundown

To print traffic between *helios* and either *hot* or *ace*:

tcpdump host helios and \(hot or ace \)

To print all IP packets between *ace* and any host except *helios*:

tcpdump ip host ace and not helios

To print all traffic between local hosts and hosts at Berkeley:

tcpdump net ucb-ether

To print all ftp traffic through internet gateway *snup*: (note that the expression is quoted to prevent the shell from (mis-)interpreting the parentheses):

tcpdump 'gateway snup and (port ftp or ftp-data)'

To print traffic neither sourced from nor destined for local hosts (if you gateway to one other net, this stuff should never make it onto your local net).

tcpdump ip and not net *localnet*

To print the start and end packets (the SYN and FIN packets) of each TCP conversation that involves a non-local host.

tcpdump 'tcp[tcpflags] & (tcp-syn|tcp-fin) != 0 and not src and dst net *localnet*'

To print IP packets longer than 576 bytes sent through gateway *snup*:

tcpdump 'gateway snup and ip[2:2] > 576'

To print IP broadcast or multicast packets that were *not* sent via ethernet broadcast or multicast:

tcpdump 'ether[0] & 1 = 0 and ip[16] >= 224'

To print all ICMP packets that are not echo requests/replies (i.e., not ping packets):

```
tcpdump 'icmp[icmptype] != icmp-echo and
icmp[icmptype] != icmp-echoreply'
```