

Konsep Jaringan

TCP/IP Programming

Oleh Politeknik Elektronika Negeri Surabaya
2017



Politeknik Elektronika Negeri Surabaya
Departemen Teknik Informatika dan Komputer

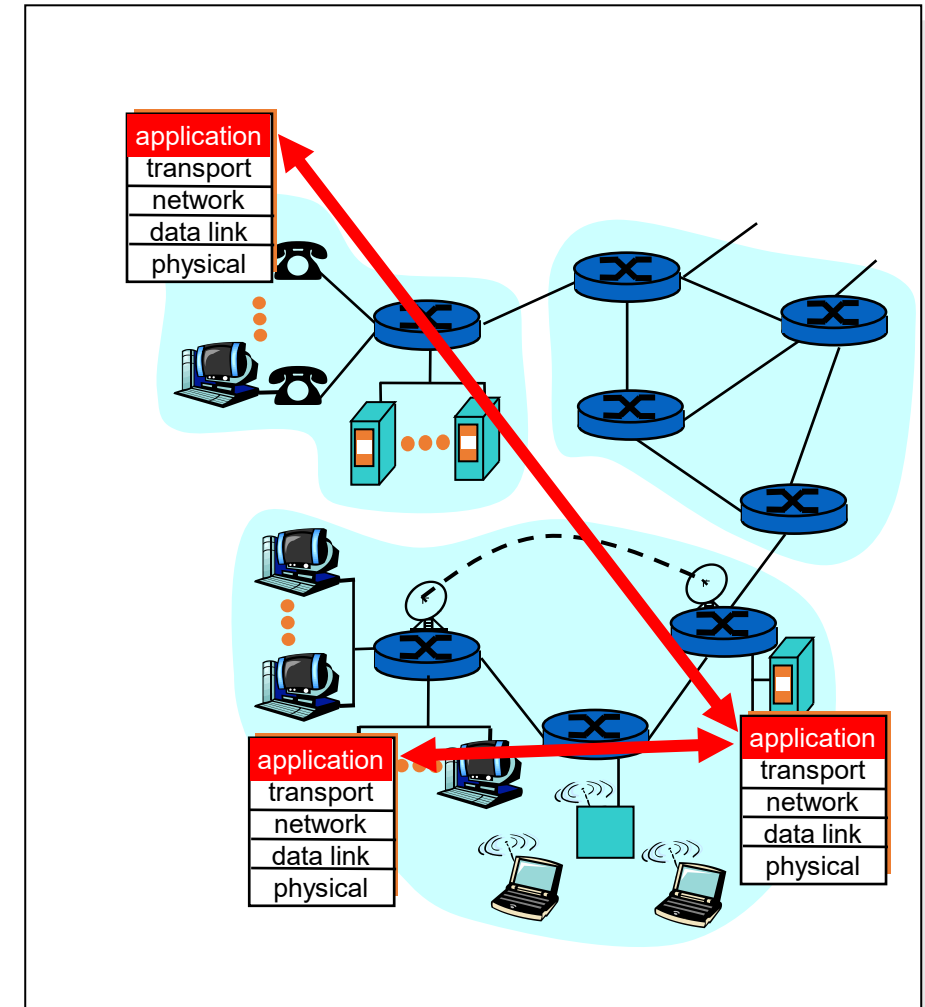
Lecture Overview

- Application layer
 - Client-server
 - Application requirements
- Background
 - TCP vs. UDP
 - Byte ordering
- Socket I/O
 - TCP/UDP server and client
 - I/O multiplexing



Applications and Application-Layer Protocols

- **Application: communicating, distributed processes**
 - Running in network hosts in “user space”
 - Exchange messages to implement app
 - e.g., email, file transfer, the Web
- **Application-layer protocols**
 - One “piece” of an app
 - Define messages exchanged by apps and actions taken
 - User services provided by lower layer protocols



Client-Server Paradigm

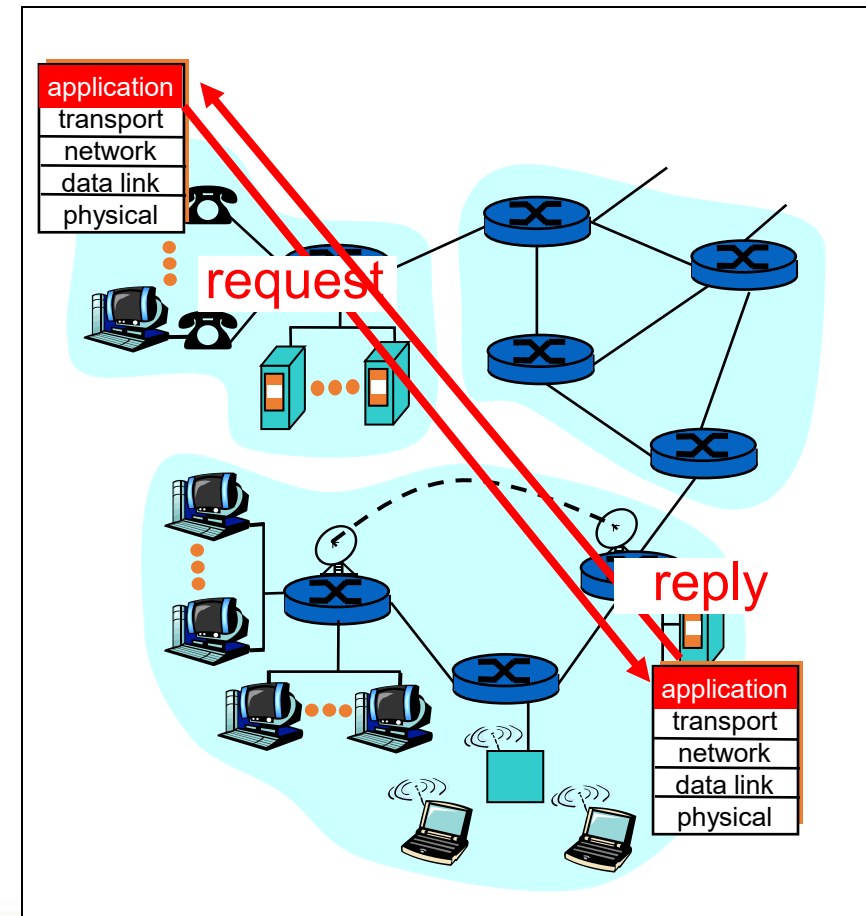
Typical network app has two pieces: *client* and *server*

Client:

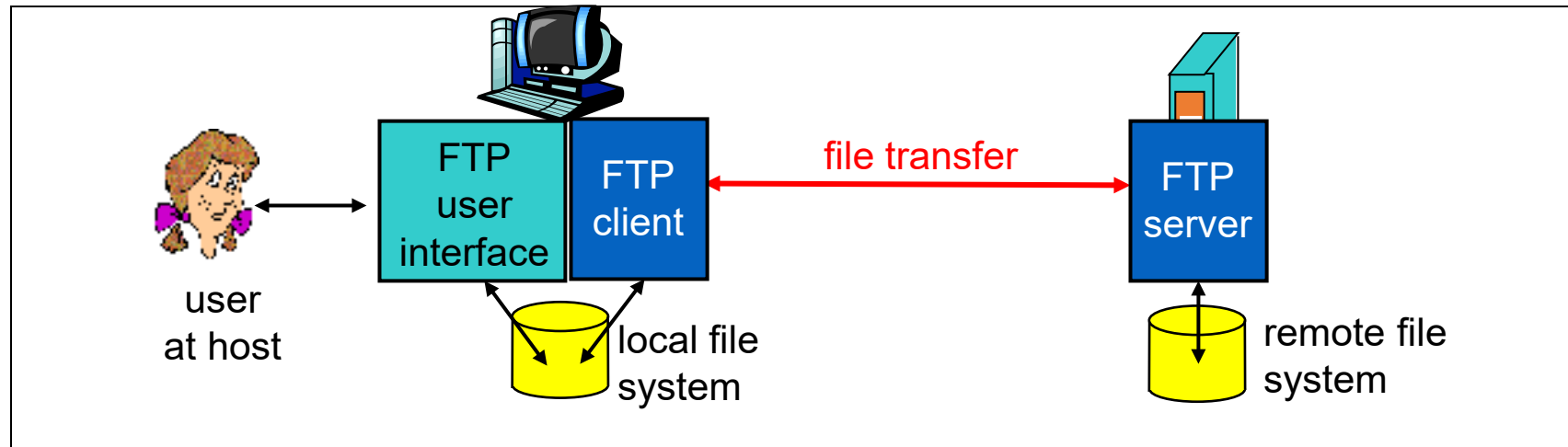
- Initiates contact with server (“speaks first”)
- Typically requests service from server,
- For Web, client is implemented in browser; for e-mail, in mail reader

Server:

- Provides requested service to client
- e.g., Web server sends requested Web page, mail server delivers e-mail



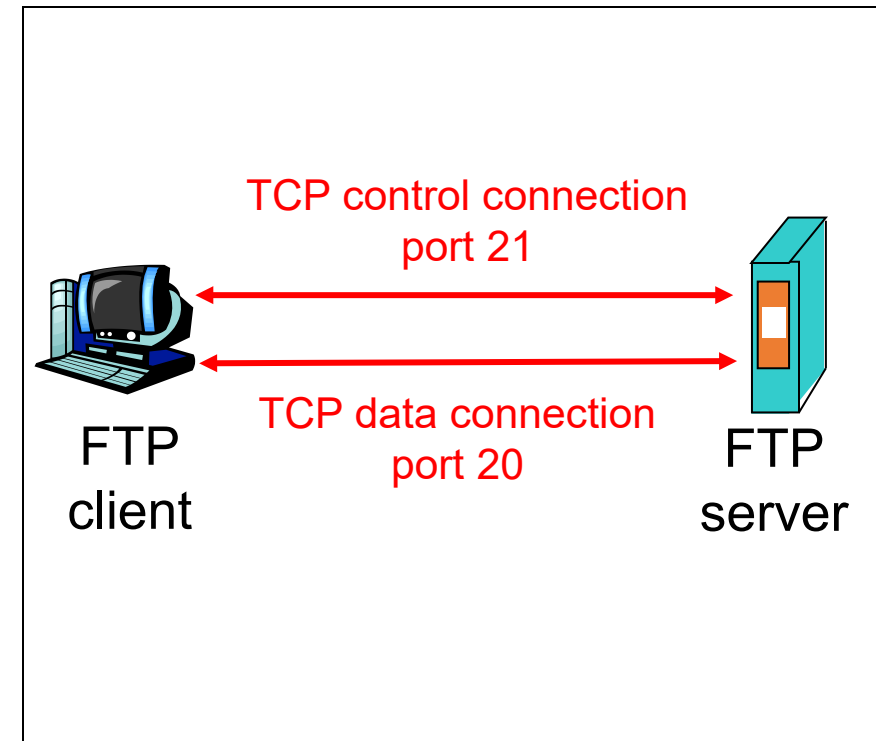
Ftp: The File Transfer Protocol



- Transfer file to/from remote host
- Client/server model
 - *Client*: side that initiates transfer (either to/from remote)
 - *Server*: remote host
- ftp: RFC 959
- ftp server: port 21

Ftp: Separate Control, Data Connections

- Ftp client contacts ftp server at port 21, specifying TCP as transport protocol
- Two parallel TCP connections opened:
 - **Control:** exchange commands, responses between client, server.
“out of band control”
 - **Data:** file data to/from server
- Ftp server maintains “state”: current directory, earlier authentication



Ftp Commands, Responses

Sample Commands:

- sent as ASCII text over control channel
- **USER *username***
- **PASS *password***
- **LIST** return list of files in current directory
- **RETR *filename*** retrieves (gets) file
- **STOR *filename*** stores (puts) file onto remote host

Sample Return Codes

- status code and phrase
- **331 Username OK, password required**
- **125 data connection already open; transfer starting**
- **425 Can't open data connection**
- **452 Error writing file**

What Transport Service Does an Application Need?

Data loss

- Some apps (e.g., audio) can tolerate some loss
- Other apps (e.g., file transfer, telnet) require 100% reliable data transfer

Timing

- Some apps (e.g., Internet telephony, interactive games) require low delay to be “effective”

Bandwidth

- Some apps (e.g., multimedia) require minimum amount of bandwidth to be “effective”
- Other apps (“elastic apps”) make use of whatever bandwidth they get

Transport Service Requirements of Common Apps

Application	Data loss	Bandwidth	Time Sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
web documents	no loss	elastic	no
real-time audio/ video	loss-tolerant	audio: 5Kb-1Mb video:10Kb-5Mb	yes, 100's msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few Kbps	yes, 100's msec
financial apps	no loss	elastic	yes and no

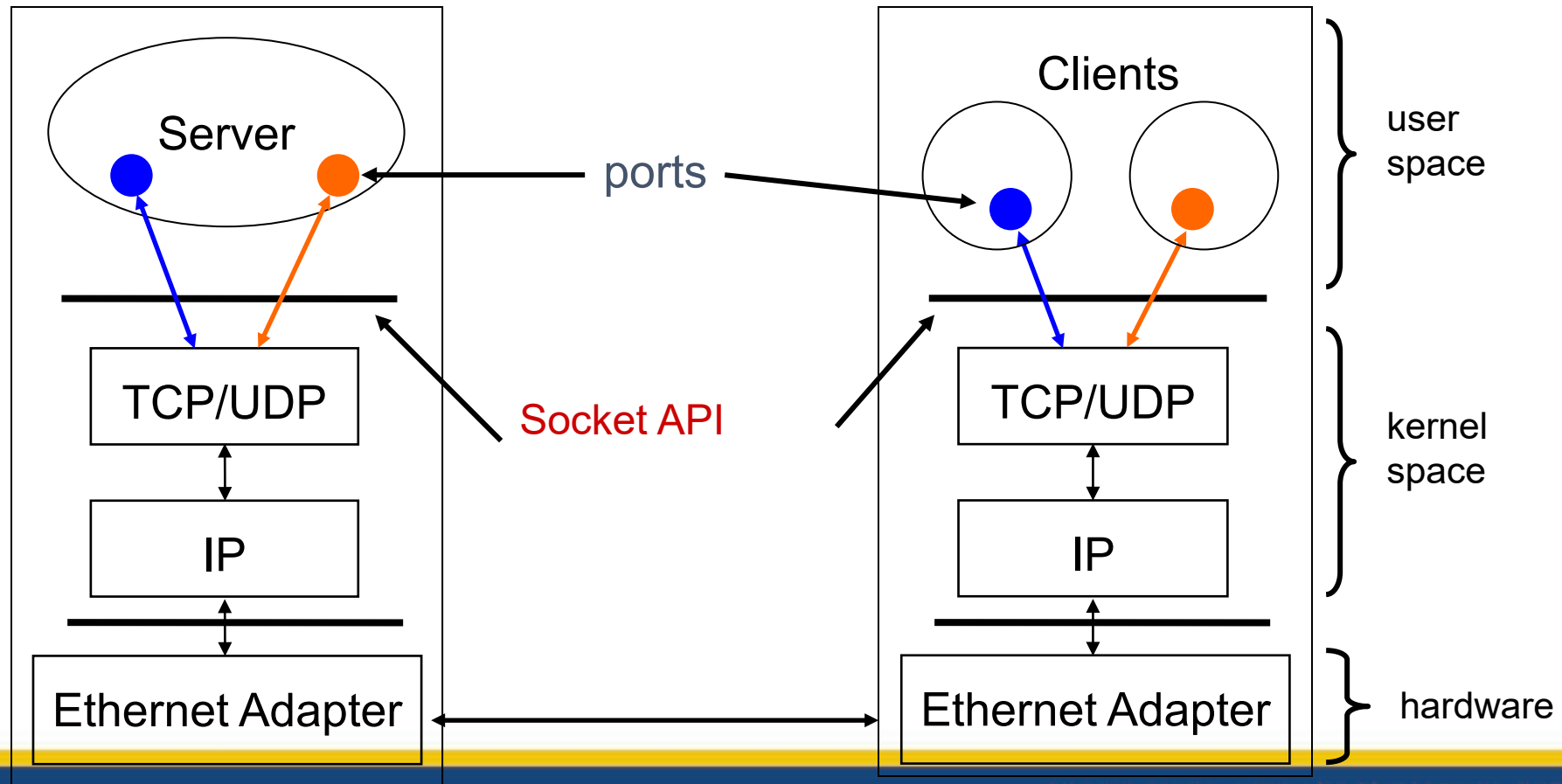
Lecture Overview

- Application layer
 - Client-server
 - Application requirements
- Background
 - TCP vs. UDP
 - Byte ordering
- Socket I/O
 - TCP/UDP server and client
 - I/O multiplexing



Server and Client

Server and Client exchange messages over the network through a common **Socket API**



User Datagram Protocol(UDP): An Analogy

UDP

- Single socket to receive messages
- No guarantee of delivery
- Not necessarily in-order delivery
- Datagram – independent packets
- Must address each packet

Postal Mail

- Single mailbox to receive letters
- Unreliable 😊
- Not necessarily in-order delivery
- Letters sent independently
- Must address each reply

Example UDP applications
Multimedia, voice over IP

Transmission Control Protocol (TCP): An Analogy

TCP

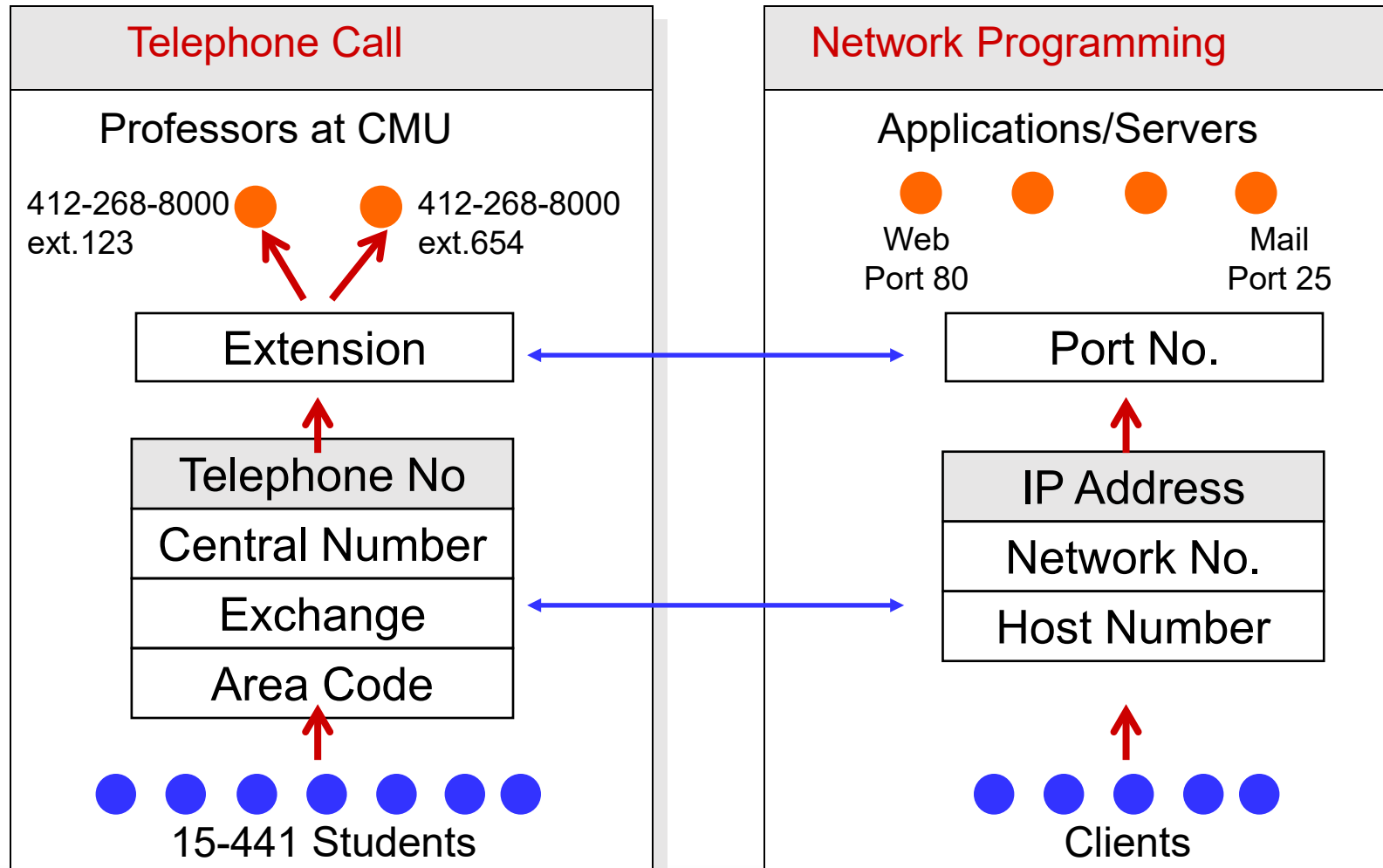
- Reliable – guarantee delivery
- Byte stream – in-order delivery
- Connection-oriented – single socket per connection
- Setup connection followed by data transfer

Telephone Call

- Guaranteed delivery
- In-order delivery
- Connection-oriented
- Setup connection followed by conversation

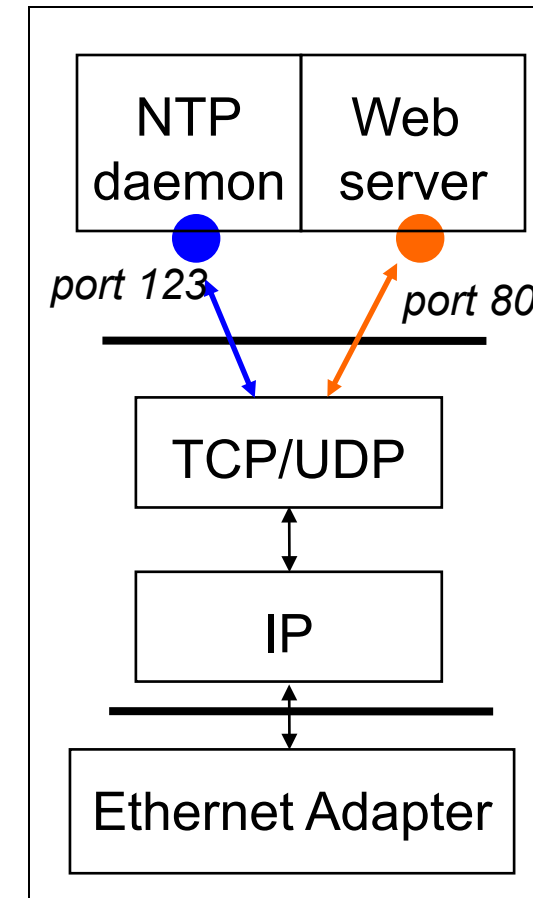
Example TCP applications
Web, Email, Telnet

Network Addressing Analogy



Concept of Port Numbers

- Port numbers are used to identify “entities” on a host
- Port numbers can be
 - Well-known (port 0-1023)
 - Dynamic or private (port 1024-65535)
- Servers/daemons usually use well-known ports
 - Any client can identify the server/service
 - HTTP = 80, FTP = 21, Telnet = 23, ...
 - */etc/service* defines well-known ports
- Clients usually use dynamic ports
 - Assigned by the kernel at run time



Names and Addresses

- Each attachment point on Internet is given unique address
 - Based on location within network – like phone numbers
- Humans prefer to deal with names not addresses
 - DNS provides mapping of name to address
 - Name based on administrative ownership of host

Internet Addressing Data Structure

```
#include <netinet/in.h>

/* Internet address structure */
struct in_addr {
    u_long s_addr;          /* 32-bit IPv4 address */
};                          /* network byte ordered */

/* Socket address, Internet style. */
struct sockaddr_in {
    u_char sin_family;      /* Address Family */
    u_short sin_port;       /* UDP or TCP Port# */
                          /* network byte ordered */
    struct in_addr sin_addr; /* Internet Address */
    char sin_zero[8];       /* unused */
};
```

- sin_family = AF_INET selects Internet address family

Byte Ordering

```
union {
    u_int32_t addr; /* 4 bytes address */
    char c[4];
} un;
/* 128.2.194.95 */
un.addr = 0x8002c25f;
/* c[0] = ? */
```

c[0] c[1] c[2] c[3]

- **Big Endian**

- Sun Solaris, PowerPC, ...

128	2	194	95
-----	---	-----	----

- **Little Endian**

- i386, alpha, ...

95	194	2	128
----	-----	---	-----

- **Network byte order = Big Endian**

Byte Ordering Functions

- Converts between **host byte order** and **network byte order**
 - 'h' = host byte order
 - 'n' = network byte order
 - 'l' = long (4 bytes), converts IP addresses
 - 's' = short (2 bytes), converts port numbers

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int
hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int
netshort);
```

Lecture Overview

- Application layer
 - Client-server
 - Application requirements
- Background
 - TCP vs. UDP
 - Byte ordering
- Socket I/O
 - TCP/UDP server and client
 - I/O multiplexing



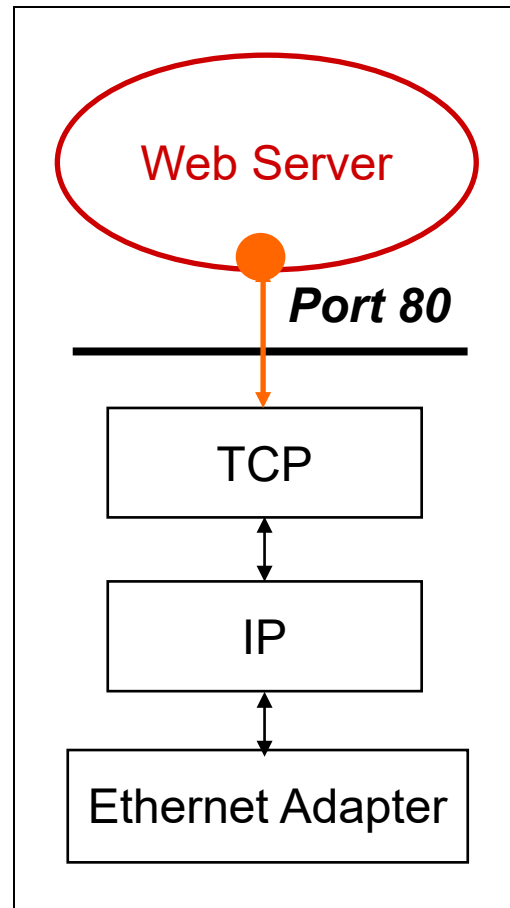
What is a Socket?

- A socket is a file descriptor that lets an application read/write data from/to the network

```
int fd;          /* socket descriptor */
if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket");
    exit(1);
}
```

- **socket** returns an integer (socket descriptor)
 - $fd < 0$ indicates that an error occurred
 - socket descriptors are similar to file descriptors
- **AF_INET**: associates a socket with the Internet protocol family
- **SOCK_STREAM**: selects the TCP protocol
- **SOCK_DGRAM**: selects the UDP protocol

TCP Server



- For example: web server
- **What does a *web server* need to do so that a *web client* can connect to it?**

Socket I/O: socket()

- Since web traffic uses TCP, the web server must create a socket of type `SOCK_STREAM`

```
int fd;           /* socket descriptor */

if((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket");
    exit(1);
}
```

- **socket** returns an integer (**socket descriptor**)
 - **fd** < 0 indicates that an error occurred
- **AF_INET** associates a socket with the Internet protocol family
- **SOCK_STREAM** selects the TCP protocol

Socket I/O: bind()

- A **socket** can be bound to a **port**

```
int fd;                                /* socket descriptor */
struct sockaddr_in srv;                /* used by bind() */

/* create the socket */

srv.sin_family = AF_INET; /* use the Internet addr family */
srv.sin_port = htons(80); /* bind socket 'fd' to port 80*/

/* bind: a client may connect to any of my addresses */
srv.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
    perror("bind"); exit(1);
}
```

- **Still not quite ready to communicate with a client...**

Socket I/O: listen()

- ***listen*** indicates that the server will accept a connection

```
int fd;                /* socket descriptor */
struct sockaddr_in srv; /* used by bind() */

/* 1) create the socket */
/* 2) bind the socket to a port */

if(listen(fd, 5) < 0) {
    perror("listen");
    exit(1);
}
```

- **Still not quite ready to communicate with a client...**



Socket I/O: accept()

- **accept** blocks waiting for a connection

```
int fd; /* socket descriptor */
struct sockaddr_in srv; /* used by bind() */
struct sockaddr_in cli; /* used by accept() */
int newfd; /* returned by accept() */
int cli_len = sizeof(cli); /* used by accept() */

/* 1) create the socket */
/* 2) bind the socket to a port */
/* 3) listen on the socket */

newfd = accept(fd, (struct sockaddr*) &cli, &cli_len);
if(newfd < 0) {
    perror("accept");    exit(1);
}
```

- **accept** returns a new socket (**newfd**) with the same properties as the original socket (**fd**)
 - **newfd** < 0 indicates that an error occurred

Socket I/O: accept() continued...

```
struct sockaddr_in cli;          /* used by accept() */
int newfd;                      /* returned by accept() */
int cli_len = sizeof(cli);      /* used by accept() */

newfd = accept(fd, (struct sockaddr*) &cli, &cli_len);
if(newfd < 0) {
    perror("accept");
    exit(1);
}
```

- How does the server know which client it is?
 - **cli.sin_addr.s_addr** contains the client's *IP address*
 - **cli.sin_port** contains the client's *port number*
- Now the server can exchange data with the client by using **read** and **write** on the descriptor **newfd**.
- Why does **accept** need to return a new descriptor?

Socket I/O: read()

- ***read*** can be used with a socket
- ***read* blocks waiting for data from the client but does not guarantee that sizeof(buf) is read**

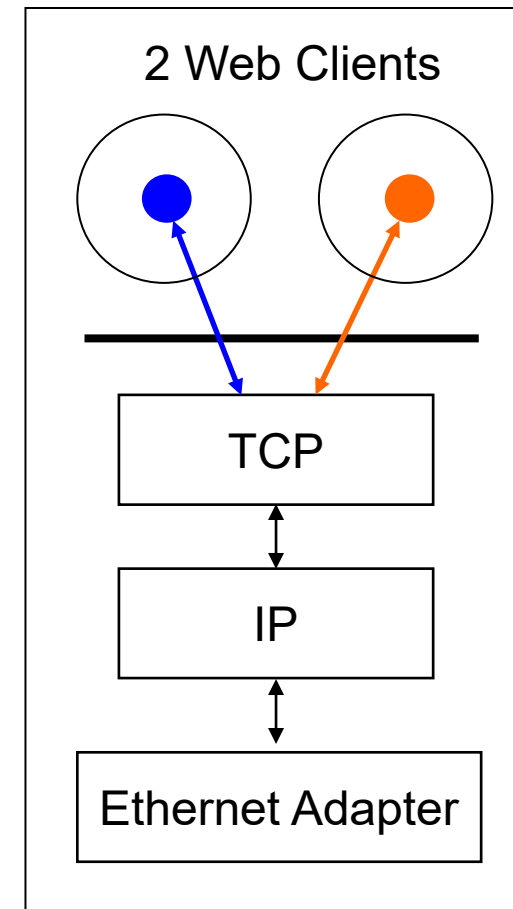
```
int fd;                /* socket descriptor */
char buf[512];         /* used by read() */
int nbytes;            /* used by read() */

/* 1) create the socket */
/* 2) bind the socket to a port */
/* 3) listen on the socket */
/* 4) accept the incoming connection */

if((nbytes = read(newfd, buf, sizeof(buf))) < 0) {
    perror("read"); exit(1);
}
```

TCP Client

- For example: web client
- **How does a *web client* connect to a *web server*?**



Dealing with IP Addresses

- IP Addresses are commonly written as strings ("128.2.35.50"), but programs deal with IP addresses as integers.

Converting strings to numerical address:

```
struct sockaddr_in srv;  
  
srv.sin_addr.s_addr = inet_addr("128.2.35.50") ;  
if(srv.sin_addr.s_addr == (in_addr_t) -1) {  
    fprintf(stderr, "inet_addr failed!\n"); exit(1);  
}
```

Converting a numerical address to a string:

```
struct sockaddr_in srv;  
char *t = inet_ntoa(srv.sin_addr) ;  
if(t == 0) {  
    fprintf(stderr, "inet_ntoa failed!\n"); exit(1);  
}
```

Translating Names to Addresses

- Gethostbyname provides interface to DNS
- Additional useful calls
 - Gethostbyaddr – returns `hostent` given `sockaddr_in`
 - Getservbyname
 - Used to get service description (typically port number)
 - Returns `servent` based on name

```
#include <netdb.h>

struct hostent *hp; /*ptr to host info for remote*/
struct sockaddr_in peeraddr;
char *name = "www.cs.cmu.edu";

peeraddr.sin_family = AF_INET;
hp = gethostbyname(name)
peeraddr.sin_addr.s_addr = ((struct in_addr*)(hp->h_addr))->s_addr;
```



Socket I/O: connect()

- **connect** allows a client to connect to a server...

```
int fd;                                /* socket descriptor */
struct sockaddr_in srv;                 /* used by connect() */

/* create the socket */

/* connect: use the Internet address family */
srv.sin_family = AF_INET;

/* connect: socket 'fd' to port 80 */
srv.sin_port = htons(80);

/* connect: connect to IP Address "128.2.35.50" */
srv.sin_addr.s_addr = inet_addr("128.2.35.50");

if(connect(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
    perror("connect"); exit(1);
}
```


Socket I/O: write()

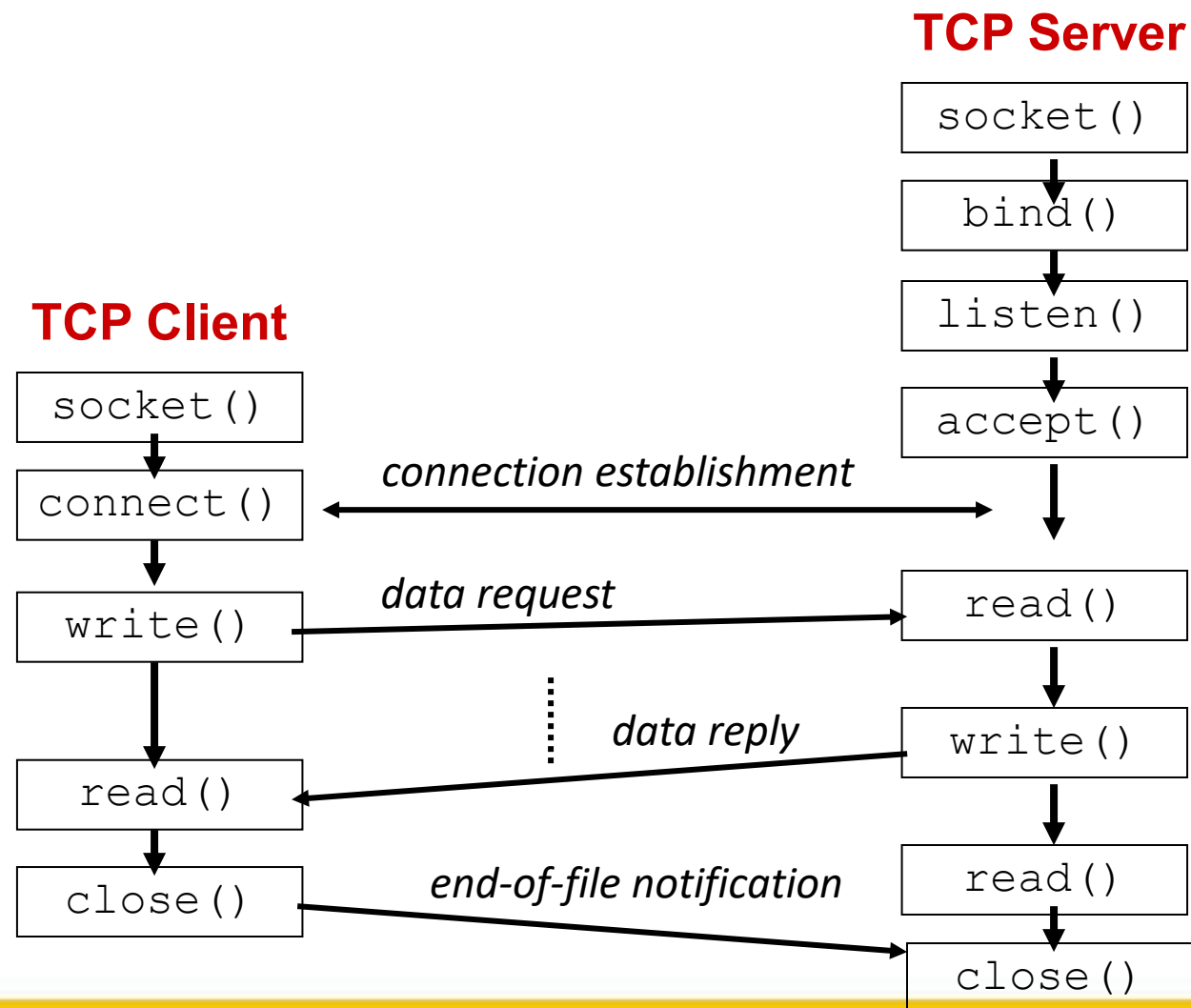
- **write** can be used with a socket

```
int fd;                                /* socket descriptor */
struct sockaddr_in srv;                 /* used by connect() */
char buf[512];                         /* used by write() */
int nbytes;                            /* used by write() */

/* 1) create the socket */
/* 2) connect() to the server */

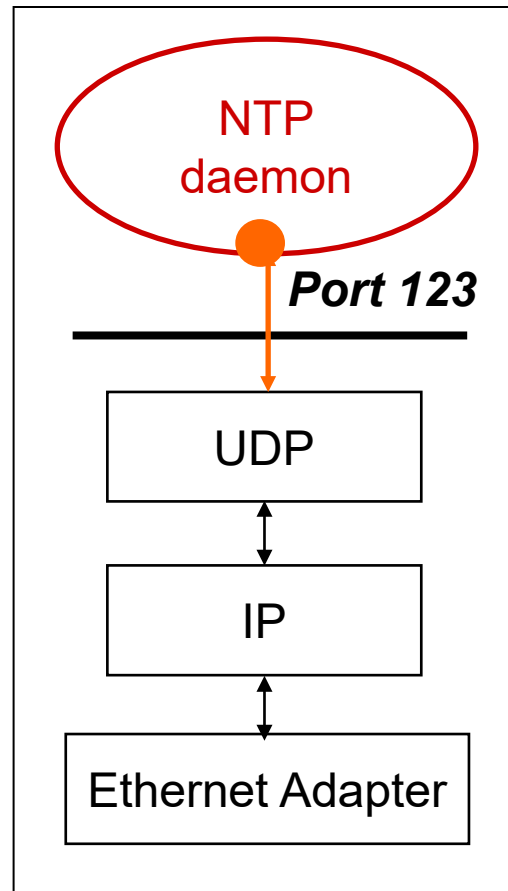
/* Example: A client could "write" a request to a server
*/
if((nbytes = write(fd, buf, sizeof(buf))) < 0) {
    perror("write");
    exit(1);
}
```

Review: TCP Client-Server Interaction



UDP Server Example

- For example: NTP daemon
- **What does a *UDP* server need to do so that a *UDP* client can connect to it?**



Socket I/O: socket()

- The UDP server must create a **datagram** socket...

```
int fd;                /* socket descriptor */

if((fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
    perror("socket");
    exit(1);
}
```

- **socket** returns an integer (**socket descriptor**)
 - **fd** < 0 indicates that an error occurred
- AF_INET: associates a socket with the Internet protocol family
- **SOCK_DGRAM**: selects the UDP protocol

Socket I/O: bind()

- A *socket* can be bound to a *port*

```
int fd;                                /* socket descriptor */
struct sockaddr_in srv;                /* used by bind() */

/* create the socket */

/* bind: use the Internet address family */
srv.sin_family = AF_INET;

/* bind: socket 'fd' to port 80*/
srv.sin_port = htons(80);

/* bind: a client may connect to any of my addresses */
srv.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
    perror("bind"); exit(1);
}
```

- Now the UDP server is ready to accept packets...



Socket I/O: recvfrom()

- **read** does not provide the client's address to the UDP server

```
int fd;                /* socket descriptor */
struct sockaddr_in srv; /* used by bind() */
struct sockaddr_in cli; /* used by recvfrom() */
char buf[512];         /* used by recvfrom() */
int cli_len = sizeof(cli); /* used by recvfrom() */
int nbytes;            /* used by recvfrom() */

/* 1) create the socket */
/* 2) bind to the socket */

nbytes = recvfrom(fd, buf, sizeof(buf), 0 /* flags */,
                  (struct sockaddr*) &cli, &cli_len);
if(nbytes < 0) {
    perror("recvfrom"); exit(1);
}
```

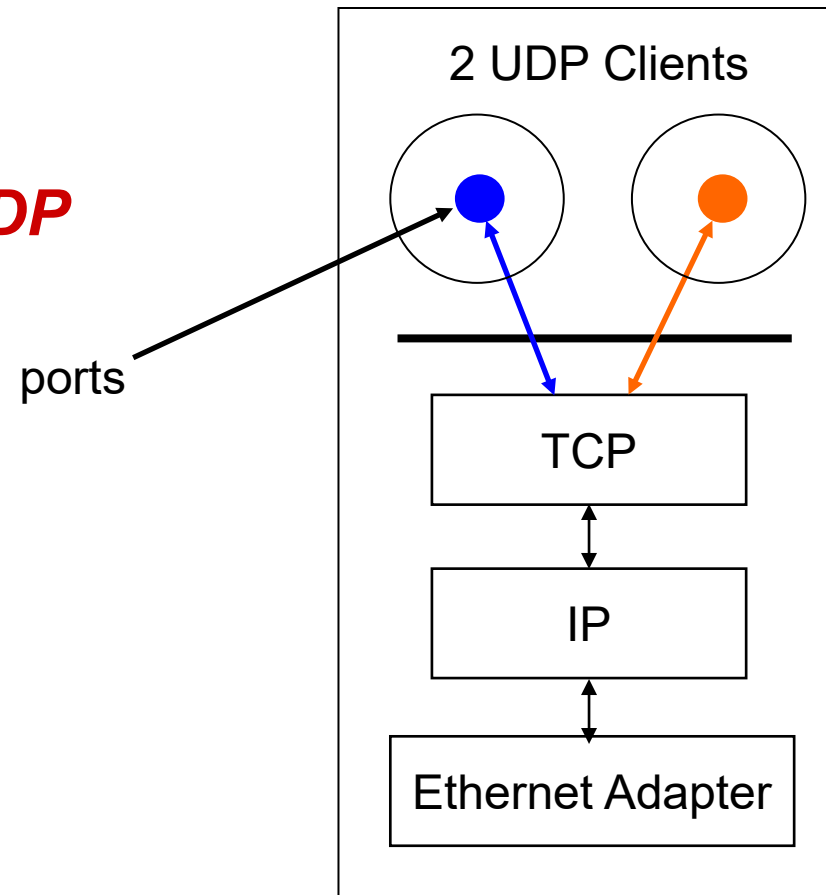
Socket I/O: recvfrom() continued...

```
nbytes = recvfrom(fd, buf, sizeof(buf), 0 /* flags */,  
                  (struct sockaddr*) cli, &cli_len);
```

- The actions performed by **recvfrom**
 - returns the number of bytes read (**nbytes**)
 - copies **nbytes** of data into **buf**
 - returns the address of the client (**cli**)
 - returns the length of **cli** (**cli_len**)
 - don't worry about flags

UDP Client Example

- How does a *UDP client* communicate with a *UDP server*?



Socket I/O: sendto()

- **write** is not allowed
- Notice that the UDP client does not **bind** a port number
 - a port number is **dynamically assigned** when the first **sendto** is called

```
int fd;                                /* socket descriptor */
struct sockaddr_in srv;                 /* used by sendto() */

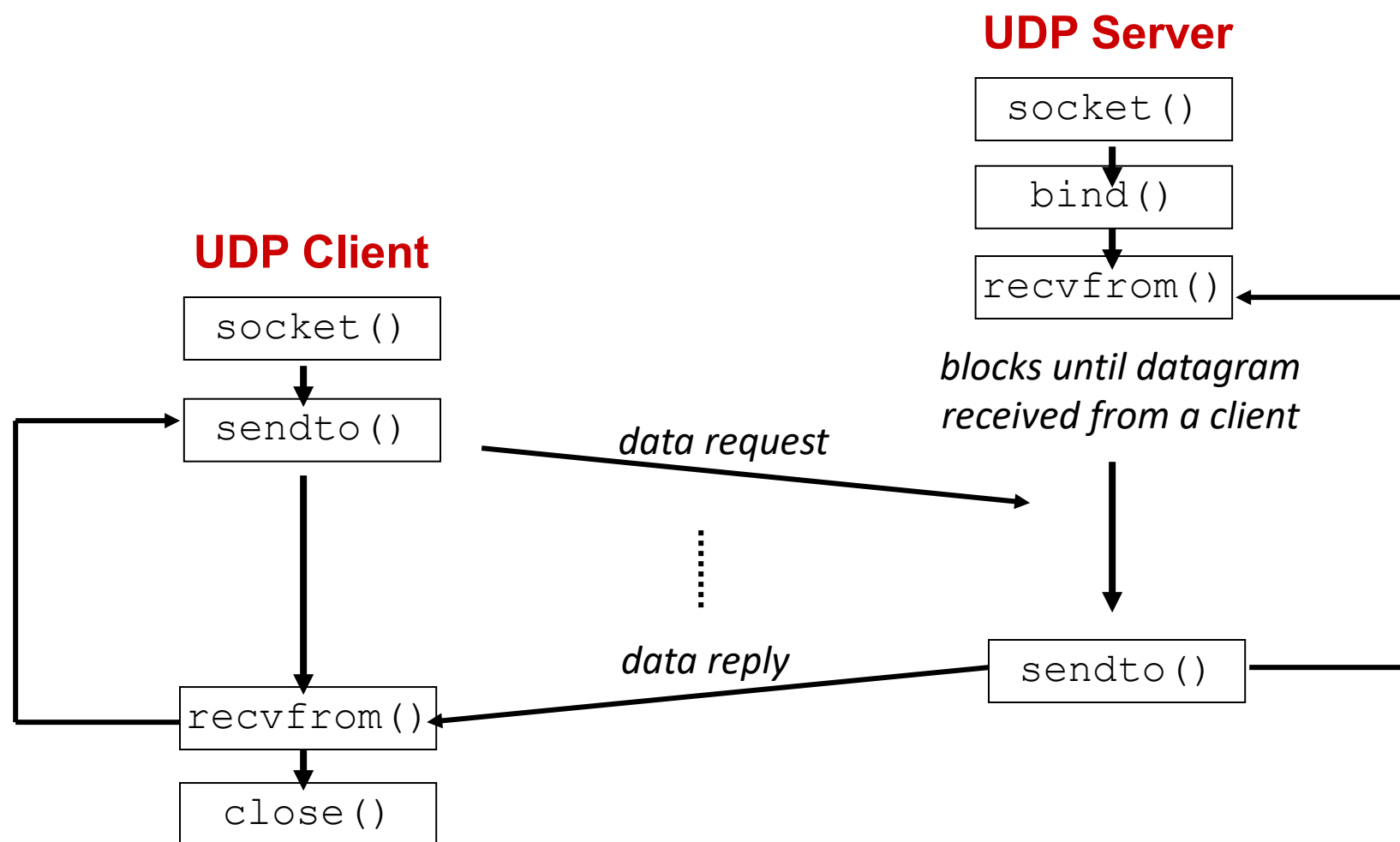
/* 1) create the socket */

/* sendto: send data to IP Address "128.2.35.50" port 80 */
srv.sin_family = AF_INET;
srv.sin_port = htons(80);
srv.sin_addr.s_addr = inet_addr("128.2.35.50");

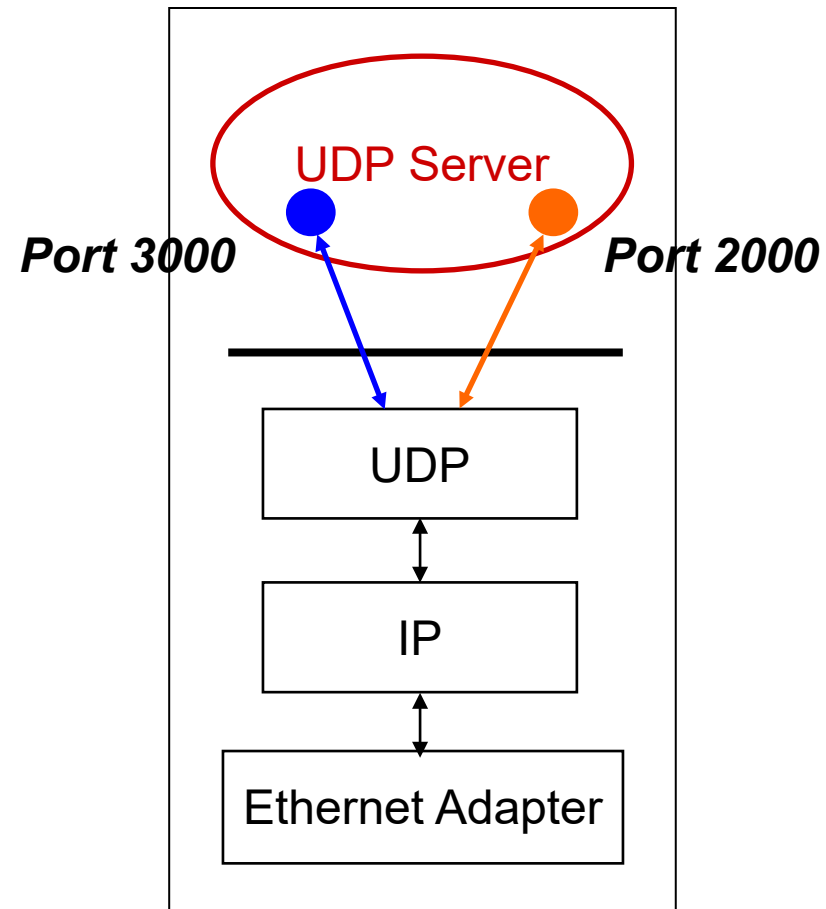
nbytes = sendto(fd, buf, sizeof(buf), 0 /* flags */,
                (struct sockaddr*) &srv, sizeof(srv));
if(nbytes < 0) {
    perror("sendto");    exit(1);
}
```



Review: UDP Client-Server Interaction



The UDP Server



- How can the *UDP* server service multiple ports simultaneously?

UDP Server: Servicing Two Ports

```
int s1;                /* socket descriptor 1 */
int s2;                /* socket descriptor 2 */

/* 1) create socket s1 */
/* 2) create socket s2 */
/* 3) bind s1 to port 2000 */
/* 4) bind s2 to port 3000 */

while(1) {
    recvfrom(s1, buf, sizeof(buf), ...);
    /* process buf */

    recvfrom(s2, buf, sizeof(buf), ...);
    /* process buf */
}
```

- What problems does this code have?

Socket I/O: select()

```
int select(int maxfds, fd_set *readfds, fd_set *writefds,
          fd_set *exceptfds, struct timeval *timeout);

FD_CLR(int fd, fd_set *fds);    /* clear the bit for fd in fds */
FD_ISSET(int fd, fd_set *fds); /* is the bit for fd in fds? */
FD_SET(int fd, fd_set *fds);    /* turn on the bit for fd in fds */
FD_ZERO(fd_set *fds);          /* clear all bits in fds */
```

- **maxfds**: number of descriptors to be tested
 - descriptors (0, 1, ... maxfds-1) will be tested
- **readfds**: a set of *fds* we want to check if data is available
 - returns a set of *fds* ready to read
 - if input argument is *NULL*, not interested in that condition
- **writefds**: returns a set of *fds* ready to write
- **exceptfds**: returns a set of *fds* with exception conditions

Socket I/O: select()

```
int select(int maxfds, fd_set *readfds, fd_set *writefds,  
          fd_set *exceptfds, struct timeval *timeout);  
  
struct timeval {  
    long tv_sec;          /* seconds */  
    long tv_usec;         /* microseconds */  
}
```

- ***timeout***
 - if NULL, wait forever and return only when one of the descriptors is ready for I/O
 - otherwise, wait up to a fixed amount of time specified by *timeout*
 - if we don't want to wait at all, create a timeout structure with timer value equal to 0
- Refer to the man page for more information

Socket I/O: select()

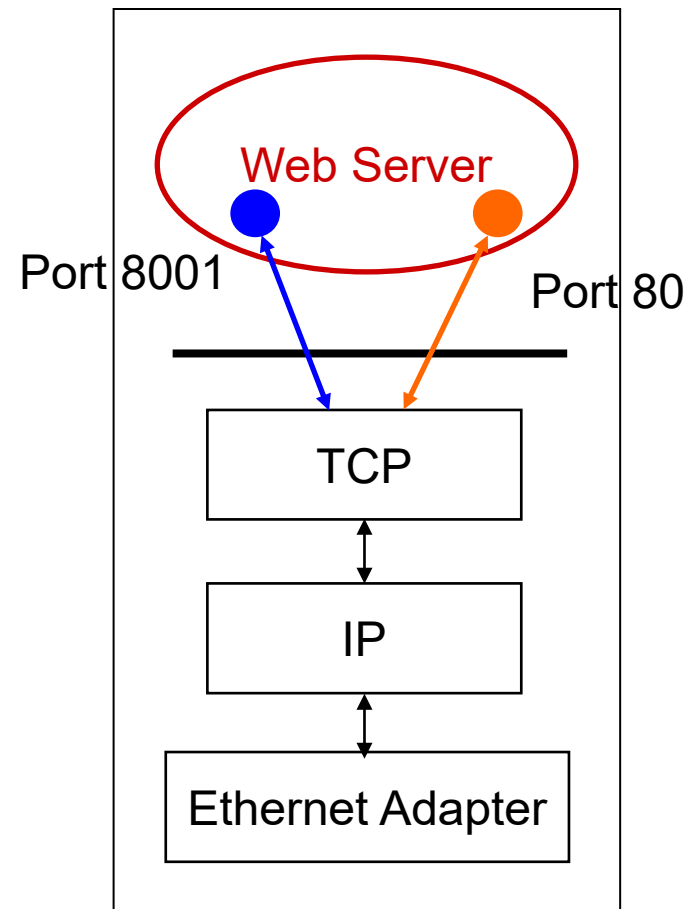
- **select** allows synchronous I/O multiplexing

```
int s1, s2;                                /* socket descriptors */
fd_set readfds;                            /* used by select() */

/* create and bind s1 and s2 */
while(1) {
    FD_ZERO(&readfds);                    /* initialize the fd set */
    /*
    FD_SET(s1, &readfds); /* add s1 to the fd set */
    FD_SET(s2, &readfds); /* add s2 to the fd set */

    if(select(s2+1, &readfds, 0, 0, 0) < 0) {
        perror("select");
        exit(1);
    }
    if(FD_ISSET(s1, &readfds)) {
        recvfrom(s1, buf, sizeof(buf), ...);
        /* process buf */
    }
    /* do the same for s2 */
}
```

More Details About a Web Server



How can a web server manage multiple connections simultaneously?

Socket I/O: select()

```
int fd, next=0;                                /* original socket */
int newfd[10];                                /* new socket descriptors */
while(1) {
    fd_set readfds;
    FD_ZERO(&readfds); FD_SET(fd, &readfds);

    /* Now use FD_SET to initialize other newfd's
       that have already been returned by accept() */

    select(maxfd+1, &readfds, 0, 0, 0);
    if(FD_ISSET(fd, &readfds)) {
        newfd[next++] = accept(fd, ...);
    }
    /* do the following for each descriptor newfd[n] */
    if(FD_ISSET(newfd[n], &readfds)) {
        read(newfd[n], buf, sizeof(buf));
        /* process data */
    }
}
```

- Now the web server can support multiple connections...

A Few Programming Notes: Building a Packet in a Buffer

```
struct packet {
    u_int32_t type;
    u_int16_t length;
    u_int16_t checksum;
    u_int32_t address;
};

/* ===== */
char buf[1024];
struct packet *pkt;

pkt = (struct packet*) buf;
pkt->type = htonl(1);
pkt->length = htons(2);
pkt->checksum = htons(3);
pkt->address = htonl(4);
```



Socket Programming References

- Man page
 - usage: man <function name>
- Textbook
 - Sections 2.6, 2.7
 - demo programs written in Java
- Unix Network Programming : Networking APIs: Sockets and XTI (Volume 1)
 - Section 2, 3, 4, 6, 8
 - ultimate socket programming bible!



Apa itu Wireshark ?

- Network Protocol Analyzer yang paling populer and “de-facto” untuk standar analisa packets
 - Open-Source (GNU Public License)
 - Multi-platform (Windows, Linux, OS X, Solaris, FreeBSD, NetBSD, and others)
 - Easily extensible
 - Large development group
- Sebelumnya bernama “Ethereal”



Apa itu Wireshark ?

- Features

- Deep inspection untuk ribuan jenis protocol
- Live capture and offline analysis
- Standard three-pane packet browser
- Mempunyai tiga jenis user interface: GUI, or via the TTY-mode TShark utility
- The most powerful display filters in the industry
- Kaya fitur untuk VoIP analysis
- Penjadapan Data online dapat menggunakan beberapa interface seperti :Ethernet, IEEE 802.11, PPP/HDLC, ATM, Bluetooth, USB, Token Ring, Frame Relay, FDDI, and others
- Coloring rules dapat diaplikasikan ke packet list for quick, intuitive analysis
- Output dapat ke XML, PostScript®, CSV, or plain text

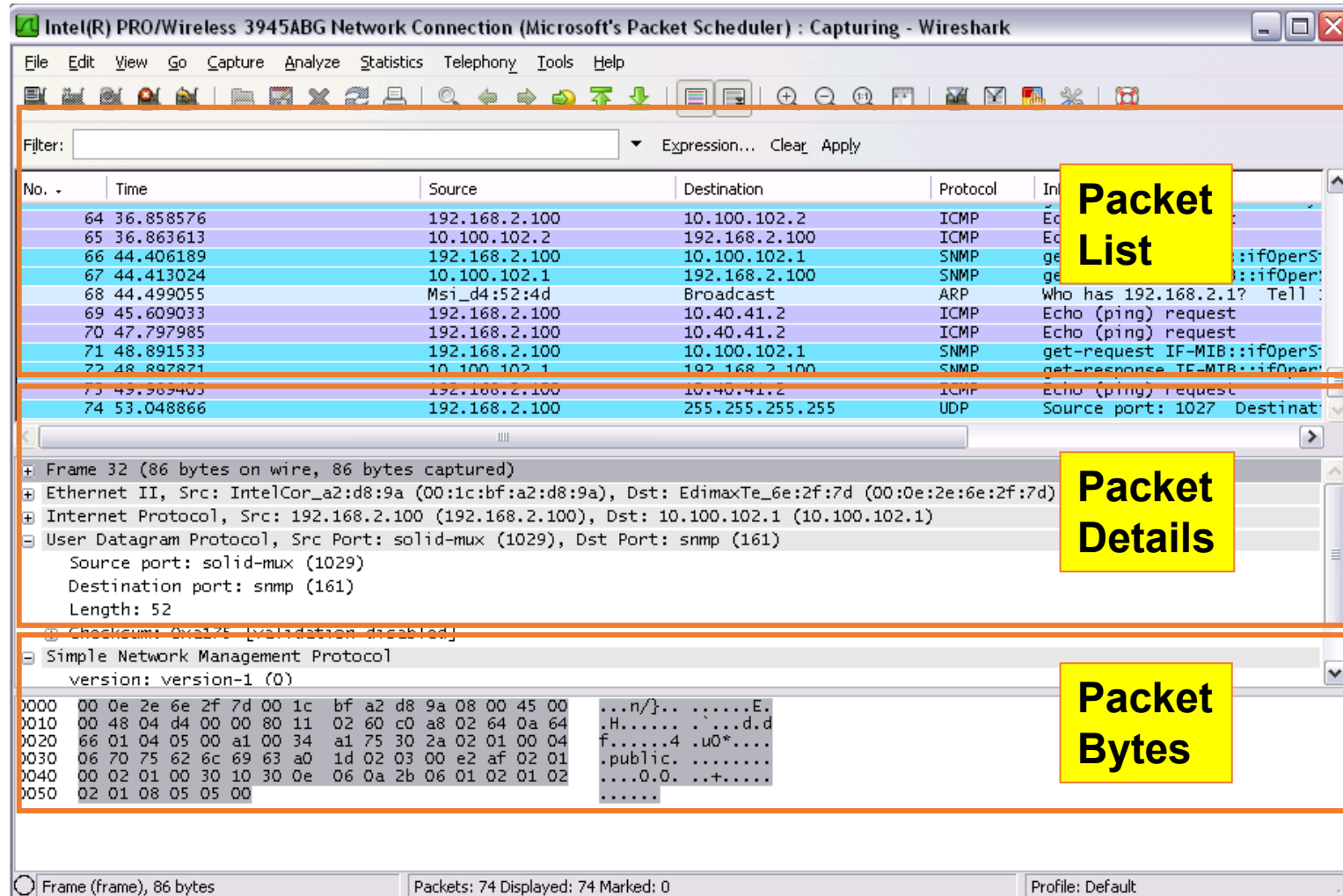


What is Wireshark?

- What we can:
 - Capture network traffic
 - Decode packet protocols using dissectors
 - Define filters – capture and display
 - Watch smart statistics
 - Analyze problems
 - Interactively browse that traffic
- Some examples people use Wireshark for:
 - Network administrators: **troubleshoot network problems**
 - Network security engineers: **examine security problems**
 - Developers: **debug protocol implementations**
 - People: **learn network protocol internals**



Interfaces

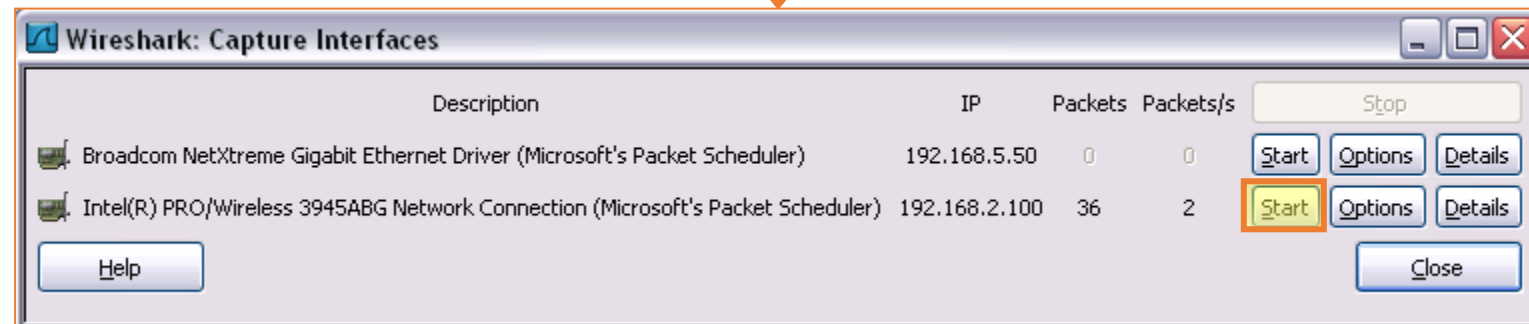
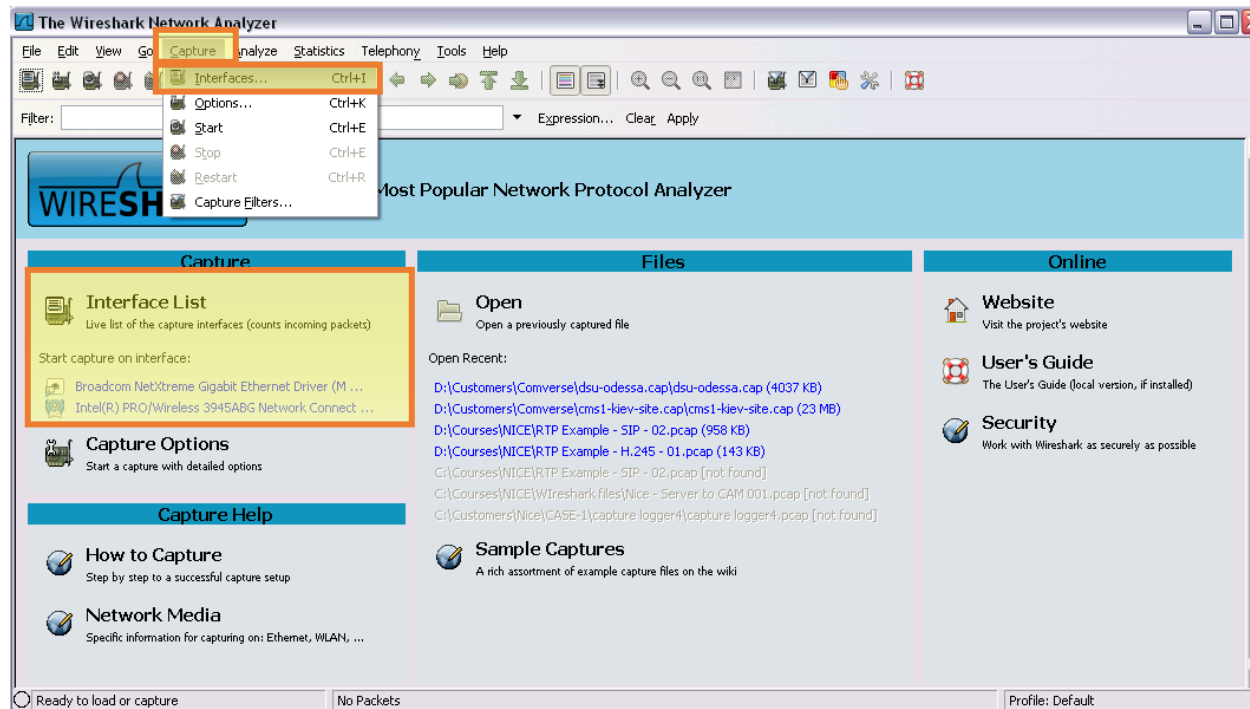


The image shows a Wireshark packet capture window titled "Intel(R) PRO/Wireless 3945ABG Network Connection (Microsoft's Packet Scheduler) : Capturing - Wireshark". The interface is divided into three main sections:

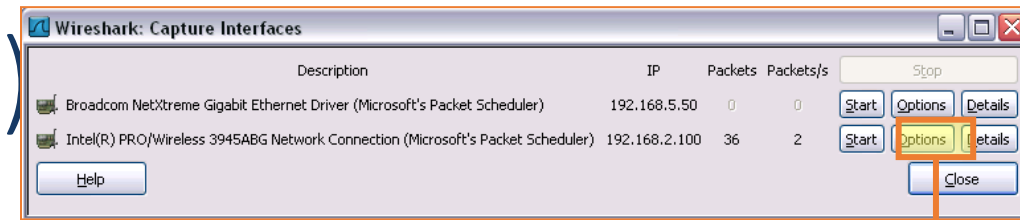
- Packet List:** A table showing a list of captured packets. The columns are No., Time, Source, Destination, Protocol, and Info. The table contains 74 packets, with the last few being ICMP Echo (ping) requests and a UDP packet.
- Packet Details:** A hierarchical view of the selected packet (Frame 32). It shows the Ethernet II header, Internet Protocol header, User Datagram Protocol header, and Simple Network Management Protocol (SNMP) data. The SNMP data includes version: version-1 (0).
- Packet Bytes:** A hex dump of the packet data, showing the raw bytes in hexadecimal and their corresponding ASCII representation.

Yellow callout boxes highlight these three sections: "Packet List", "Packet Details", and "Packet Bytes".

Capturing Packets (1/3)



Capturing Packets (2/3)



Capture all packets on the network

Buffer size – in order not to fill your laptop disk

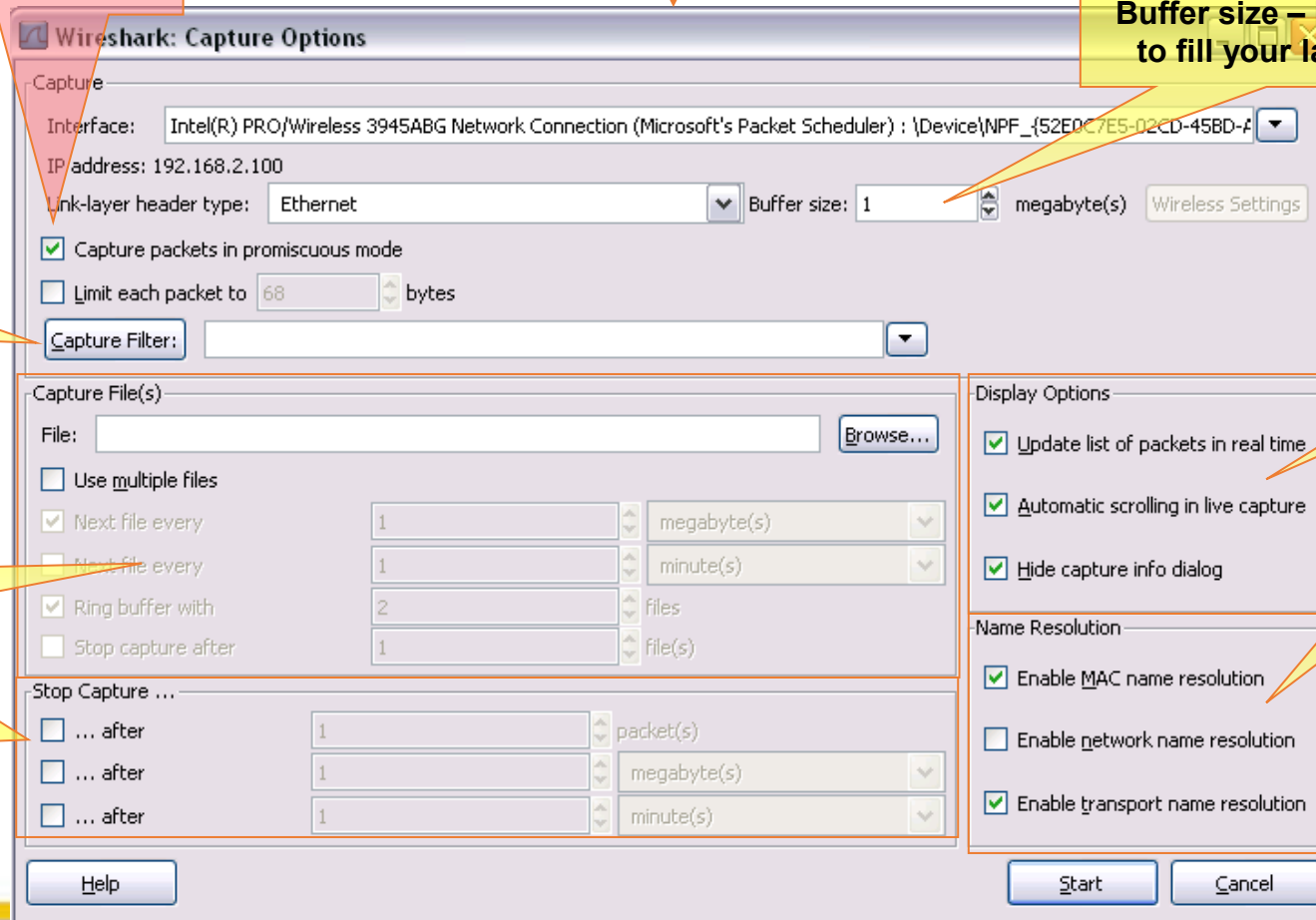
Capture filter

Display options

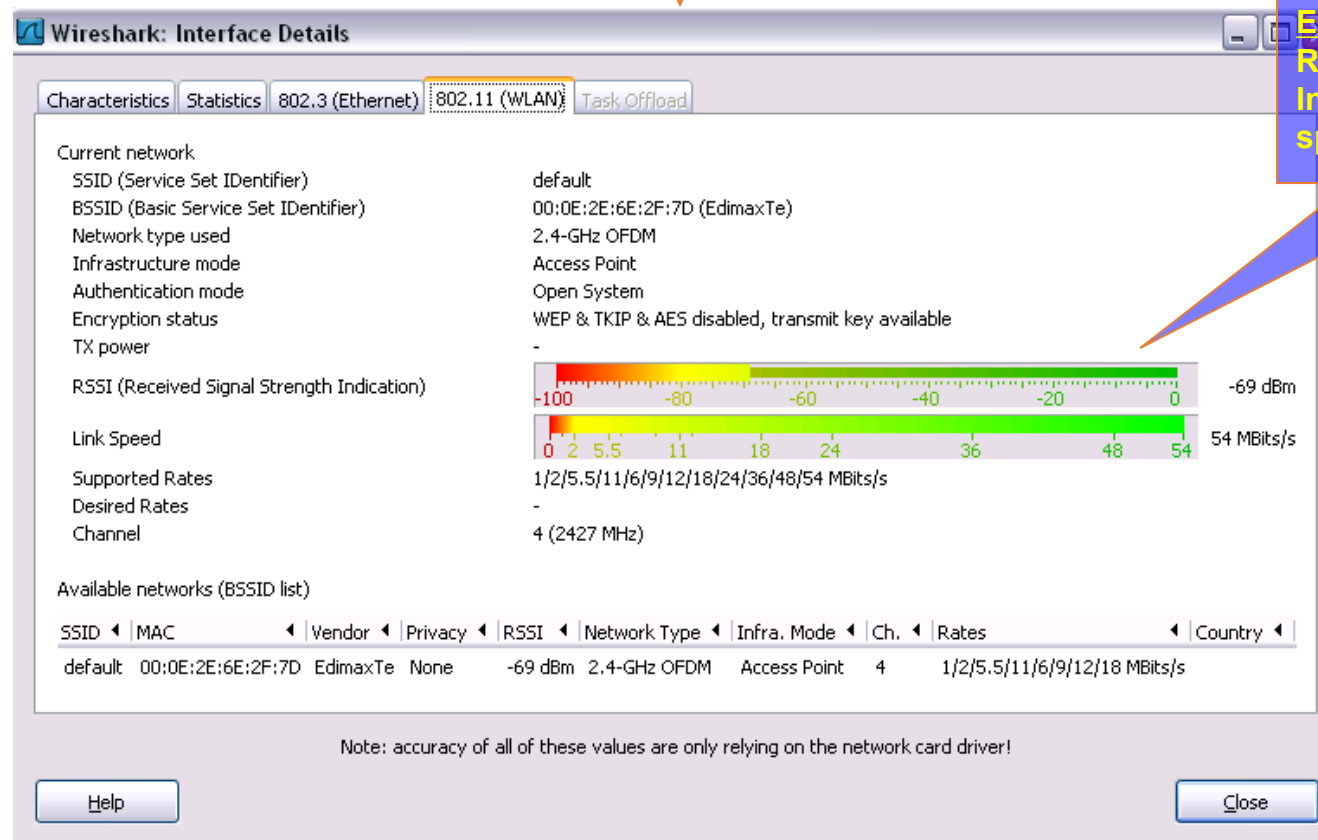
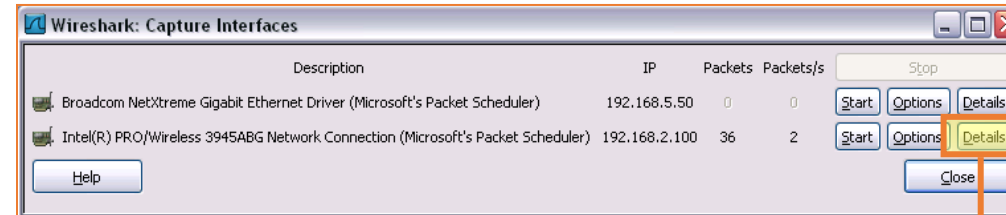
Capture in multiple files

Name resolution options

When to automatically stop the capture



Capturing Packets (3/3)



Example (W-LAN):
Received Signal Strength
Indication (RSSI) and Link
speed (BW)

Analyzing Packets (1/9)

Ethernet Frame Example

No. -	Time	Source	Destination	Protocol	Info
4	23.227339	1.1.1.1	127.0.0.1	UDP	Source port: 55555 Destination
5	23.838867	212.179.1.202	10.159.3.103	FTP	Response: 200 Type set to I.
6	23.857421	10.159.3.103	212.179.1.202	FTP	Request: SIZE upload1_1936
7	23.996093	212.179.1.202	10.159.3.103	FTP	Response: 213 11026917
8	24.012695	10.159.3.103	212.179.1.202	FTP	Request: MDTM upload1_1936
9	24.208984	212.179.1.202	10.159.3.103	FTP	Response: 213 20071202174050
10	24.266601	10.159.3.103	212.179.1.202	FTP	Request: PASV
11	24.391601	212.179.1.202	10.159.3.103	FTP	Response: 227 Entering Passi

Frame 10 (60 bytes on wire, 60 bytes captured)

Arrival Time: Jan 13, 2008 11:44:18.844726000

[Time delta from previous captured frame: 0.057617000 seconds]

[Time delta from previous displayed frame: 0.057617000 seconds]

[Time since reference or first frame: 24.266601000 seconds]

Frame Number: 10

Frame Length: 60 bytes

Capture Length: 60 bytes

[Frame is marked: False]

[Protocols in frame: eth:ip:tcp:ftp]

[Coloring Rule Name: TCP]

[Coloring Rule String: tcp]

Ethernet II, Src: Xerox_00:00:00 (01:00:01:00:00:00), Dst: d4:c8:20:00:01:00 (d4:c8:20:00:01:00)

Destination: d4:c8:20:00:01:00 (d4:c8:20:00:01:00)

Address: d4:c8:20:00:01:00 (d4:c8:20:00:01:00)

.... 0 = IG bit: Individual address (unicast)

.... 0 = LG bit: Globally unique address (factory default)

Source: Xerox_00:00:00 (01:00:01:00:00:00)

Address: Xerox_00:00:00 (01:00:01:00:00:00)

.... 1 = IG bit: Group address (multicast/broadcast)

.... 0 = LG bit: Globally unique address (factory default)

Type: IP (0x0800)

Internet Protocol, Src: 10.159.3.103 (10.159.3.103), Dst: 212.179.1.202 (212.179.1.202)

Transmission Control Protocol, Src Port: mps-raft (1700), Dst Port: ftp (21), Seq: 47, Ack: 55, Len: 6

File Transfer Protocol (FTP)

Analyzing Packets (2/9)

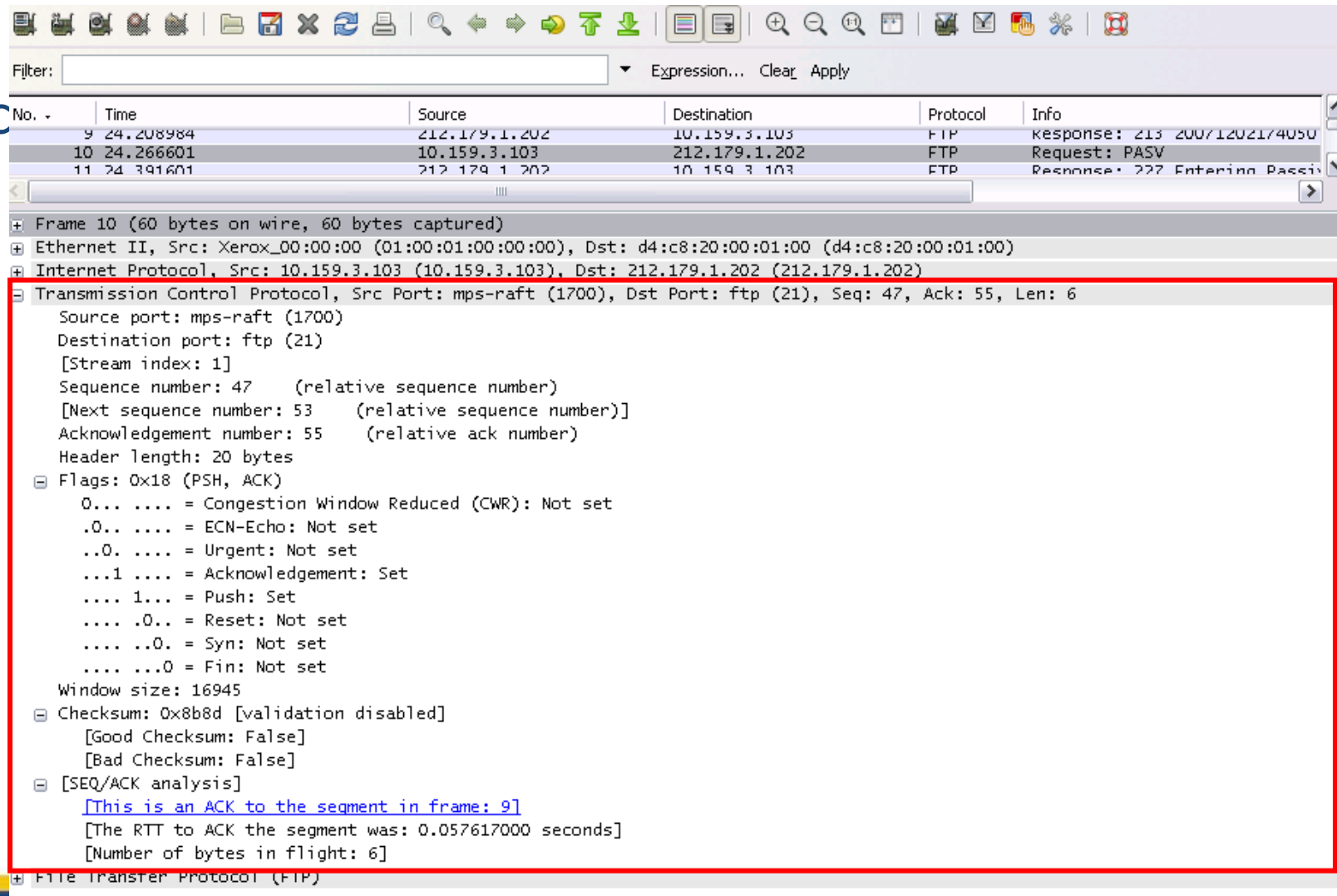
- IP Pack

No. -	Time	Source	Destination	Protocol	Info
4	23.227539	1.1.1.1	127.0.0.1	UDP	Source port: 33333 Destination
5	23.838867	212.179.1.202	10.159.3.103	FTP	Response: 200 Type set to I.
6	23.857421	10.159.3.103	212.179.1.202	FTP	Request: SIZE upload1_1936
7	23.996093	212.179.1.202	10.159.3.103	FTP	Response: 213 11026917
8	24.012695	10.159.3.103	212.179.1.202	FTP	Request: MDTM upload1_1936
9	24.208984	212.179.1.202	10.159.3.103	FTP	Response: 213 20071202174050
10	24.266601	10.159.3.103	212.179.1.202	FTP	Request: PASV

+	Frame 10 (60 bytes on wire, 60 bytes captured)
+	Ethernet II, Src: Xerox_00:00:00 (01:00:01:00:00:00), Dst: d4:c8:20:00:01:00 (d4:c8:20:00:01:00)
-	Internet Protocol, Src: 10.159.3.103 (10.159.3.103), Dst: 212.179.1.202 (212.179.1.202)
	Version: 4
	Header length: 20 bytes
-	Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
	0000 00.. = Differentiated Services Codepoint: Default (0x00)
0. = ECN-Capable Transport (ECT): 0
0 = ECN-CE: 0
	Total Length: 46
	Identification: 0x5f49 (24393)
-	Flags: 0x04 (Don't Fragment)
	0... = Reserved bit: Not set
	.1.. = Don't fragment: Set
	..0. = More fragments: Not set
	Fragment offset: 0
	Time to live: 128
	Protocol: TCP (0x06)
-	Header checksum: 0xb6fd [correct]
	[Good: True]
	[Bad : False]
	Source: 10.159.3.103 (10.159.3.103)
	Destination: 212.179.1.202 (212.179.1.202)
+	Transmission Control Protocol, Src Port: mps-raft (1700), Dst Port: ftp (21), Seq: 47, Ack: 55, Len: 6
+	File Transfer Protocol (FTP)

Analyzing Packets (3/9)

- TCP Pack



Filter: Expression... Clear Apply

No.	Time	Source	Destination	Protocol	Info
9	24.208984	212.179.1.202	10.159.3.103	FTP	Response: 213 200/1202174050
10	24.266601	10.159.3.103	212.179.1.202	FTP	Request: PASV
11	24.391601	212.179.1.202	10.159.3.103	FTP	Response: 227 Entering Passive

Frame 10 (60 bytes on wire, 60 bytes captured)

- Ethernet II, Src: Xerox_00:00:00 (01:00:01:00:00:00), Dst: d4:c8:20:00:01:00 (d4:c8:20:00:01:00)
- Internet Protocol, Src: 10.159.3.103 (10.159.3.103), Dst: 212.179.1.202 (212.179.1.202)
- Transmission Control Protocol, Src Port: mps-raft (1700), Dst Port: ftp (21), Seq: 47, Ack: 55, Len: 6
 - Source port: mps-raft (1700)
 - Destination port: ftp (21)
 - [Stream index: 1]
 - Sequence number: 47 (relative sequence number)
 - [Next sequence number: 53 (relative sequence number)]
 - Acknowledgement number: 55 (relative ack number)
 - Header length: 20 bytes
 - Flags: 0x18 (PSH, ACK)
 - 0... = Congestion Window Reduced (CWR): Not set
 - .0.. = ECN-Echo: Not set
 - ..0. = Urgent: Not set
 - ...1 = Acknowledgement: Set
 - 1... = Push: Set
 -0.. = Reset: Not set
 -0. = Syn: Not set
 -0 = Fin: Not set
 - Window size: 16945
 - Checksum: 0x8b8d [validation disabled]
 - [Good Checksum: False]
 - [Bad Checksum: False]
 - [SEQ/ACK analysis]
 - [\[This is an ACK to the segment in frame: 9\]](#)
 - [The RTT to ACK the segment was: 0.057617000 seconds]
 - [Number of bytes in flight: 6]
- File Transfer Protocol (FTP)

Analyzing Packets (4/9)

- TCP

Wireshark packet capture analysis showing a TCP SYN sequence. The packet list shows a SYN packet (No. 7) and its corresponding ACK (No. 11). The packet details pane shows the structure of the captured frame, including Ethernet II, Internet Protocol, User Datagram Protocol, and Domain Name System. The packet bytes pane shows the raw data in hexadecimal and ASCII.

No.	Time	Source	Destination	Protocol	Info
1	0.000000	192.168.2.100	10.40.41.2	ICMP	Echo (ping) request
2	2.183304	192.168.2.100	10.40.41.2	ICMP	Echo (ping) request
3	3.430100	192.168.2.100	212.150.49.10	DNS	Standard query A www.ynet.co.il
4	3.457181	212.150.49.10	192.168.2.100	DNS	Standard query response CNAME ynet.co.il.d4p.net CNAME a39.g.
5	3.461602	192.168.2.100	212.150.49.10	DNS	Standard query A www.lenovo.com
6	3.623867	192.168.2.100	212.143.162.157	TCP	dzdaemon > http [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=1 TSV
7	3.728385	212.143.162.157	192.168.2.100	TCP	http > dzdaemon [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0 MSS=145
8	3.728429	192.168.2.100	212.143.162.157	TCP	dzdaemon > http [ACK] Seq=1 Ack=1 Win=128480 Len=0
9	3.728839	192.168.2.100	212.143.162.157	HTTP	GET / HTTP/1.1
10	3.768896	212.143.162.157	192.168.2.100	TCP	http > dzdaemon [ACK] Seq=1 Ack=580 Win=6948 Len=0
11	3.770703	212.143.162.157	192.168.2.100	HTTP	HTTP/1.0 301 Moved Permanently
12	3.772411	192.168.2.100	212.143.162.157	HTTP	GET /home/O.7340.L-8.00.html HTTP/1.1

Frame 5 (74 bytes on wire, 74 bytes captured)

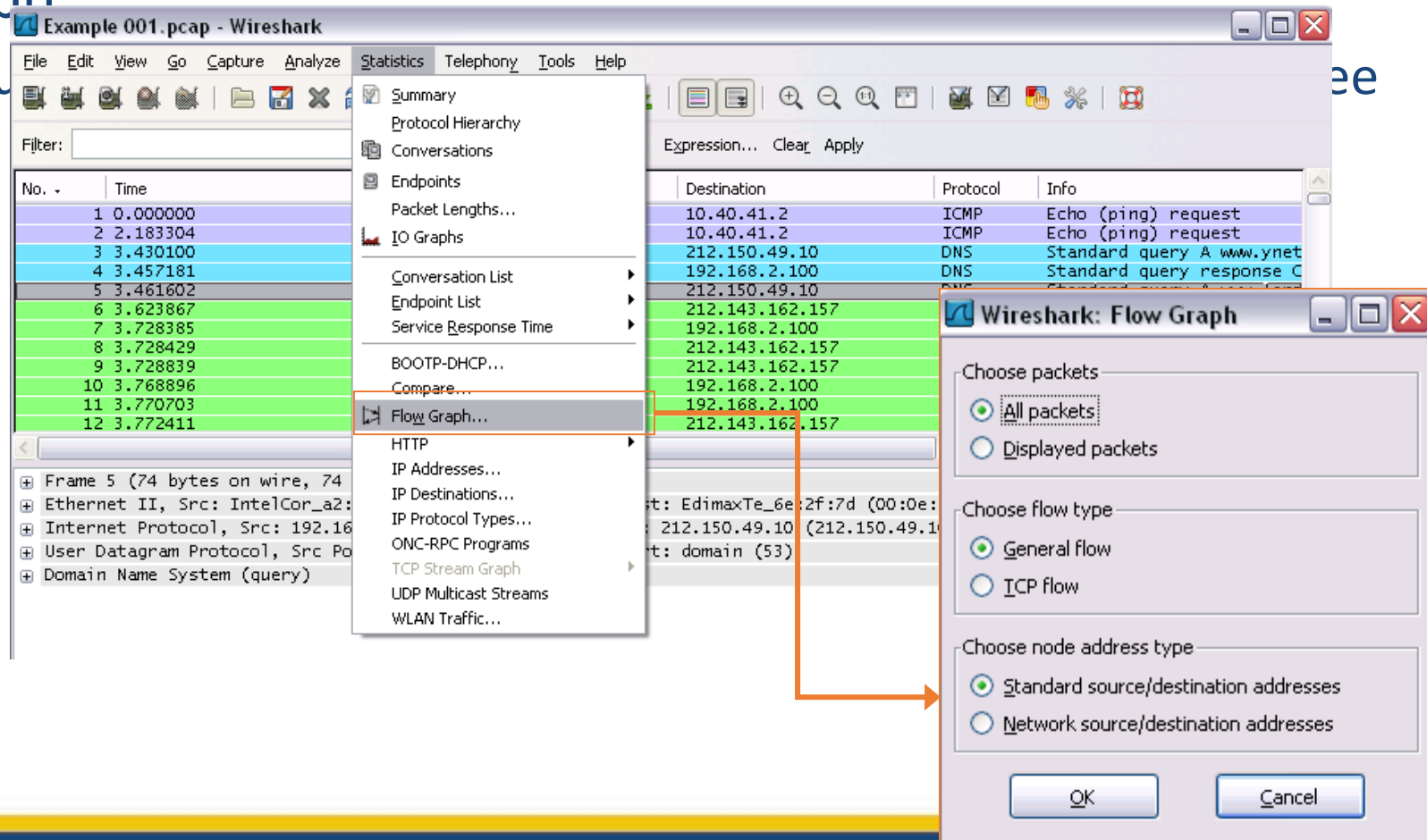
- Ethernet II, Src: IntelCor_a2:d8:9a (00:1c:bf:a2:d8:9a), Dst: EdimaxTe_6e:2f:7d (00:0e:2e:6e:2f:7d)
- Internet Protocol, Src: 192.168.2.100 (192.168.2.100), Dst: 212.150.49.10 (212.150.49.10)
- User Datagram Protocol, Src Port: natuslink (2895), Dst Port: domain (53)
- Domain Name System (query)

0000 00 0e 2e 6e 2f 7d 00 1c bf a2 d8 9a 08 00 45 00 ...n/}..E.
 0010 00 3c 7f ea 00 00 80 11 f2 19 c0 a8 02 64 d4 96 .<.....d..
 0020 31 0a 0b 4f 00 35 00 28 f5 df 9e d7 01 00 00 01 1..0.5.(.....
 0030 00 00 00 00 00 00 03 77 77 77 06 6c 65 6e 6f 76w ww.lenov
 0040 6f 03 63 6f 6d 00 00 01 00 01 o.com... ..

Analyzing Packets (5/9)

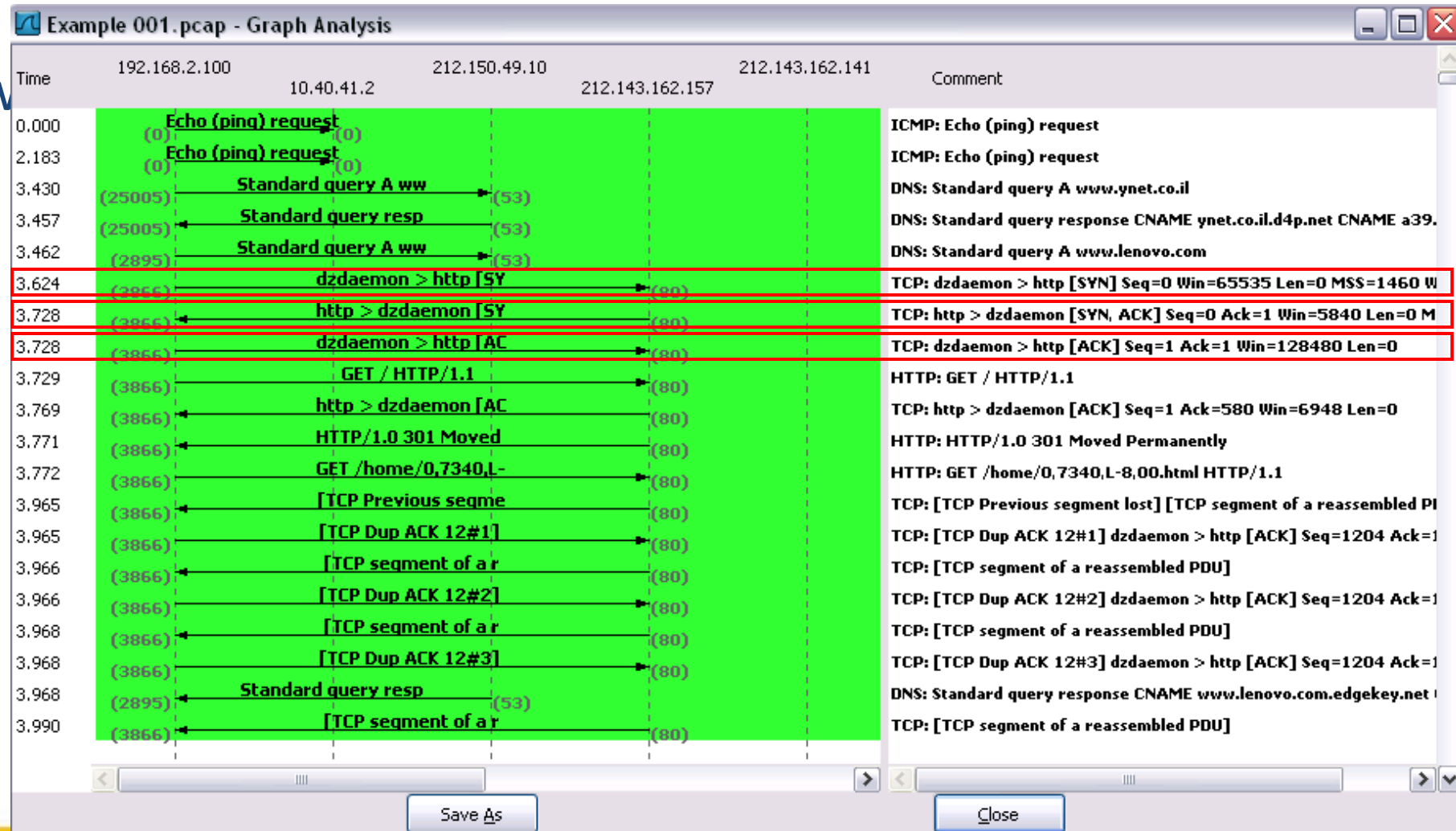
- Flow Graph

- Giving up



Analyzing Packets (6/9)

- Flow



Analyzing Packets (7/9)

- Filter

The image shows the Wireshark network protocol analyzer interface. The main window displays a list of captured packets. A context menu is open over packet 42, with the 'Follow TCP Stream' option highlighted. A red arrow points from this option to a secondary window titled 'Follow TCP Stream'.

Packet List:

No.	Time	Source	Destination	Protocol	Info
38	10.031657	10.115.243.30	10.114.30.180	DNS	Standard query response CNAME toolbarqueries.l.google.
39	10.032397	10.114.30.180	64.233.183.99	TCP	peport > http [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=
40	10.035182	64.233.183.99	10.114.30.180	TCP	http > peport [SYN, ACK] Seq=0 Ack=1 Win=32768 Len=0 M
41	10.035234	10.114.30.180	64.233.183.99	TCP	peport > http [ACK] Seq=1 Ack=1 Win=131400 Len=0 TSV=1
42	10.035443	10.114.30.180	64.233.183.99	HTTP	nt-auto&ch=6174981939&freshn
43	10.145399	64.233.183.99	10.114.30.180	TCP	Ack=447 Win=32768 Len=0 TSV=
44	10.312738	64.233.183.99	10.114.30.180	TCP	led PDU]
45	10.312814	64.233.183.99	10.114.30.180	HTTP	1)
46	10.312862	10.114.30.180	64.233.183.99	TCP	7 Ack=322 Win=131076 Len=0 T
47	10.399875	10.114.30.180	10.115.243.30	DNS	tcp.dc._msdcs.ndi.local
48	10.400571	10.115.243.30	10.114.30.180	DNS	o such name

Packet Details (Frame 42):

- Frame 42 (512 bytes on wire, 512 bytes captured)
- Ethernet II, Src: Ibm_42:c2:4d (00:09:6b:42:c2:4d), Dst: LucentTe_cf:cd:2c (00:30:6d:cf:cd:2c)
- Internet Protocol, Src: 10.114.30.180 (10.114.30.180), Dst: 64.233.183.99 (64.233.183.99)
- Transmission Control Protocol, Src Port: peport (1449), Dst Port: http (80), Seq: 1, Ack: 1, Len: 0
- Hypertext Transfer Protocol

Follow TCP Stream Window:

Stream Content

```
GET /search?client=navclient-auto&ch=6174981939&freshness_check=4ilp-GrPqKEX_r_lNxaYw&iqrn=q4&orig=0J&ie=UTF-8&oe=UTF-8&features=Rank&q=info:http%3A%2F%2Fwww%2Eeynet%2Eeco%2Eil%2FHTTP/1.1
User-Agent: Mozilla/4.0 (compatible; GoogleToolbar 2.0.114.9-big; Windows XP 5.1)
Host: toolbarqueries.google.com
Cache-Control: no-cache
Cookie: PREF=ID=1a18560743a17669;TB=2;CR=1;TM=1113765996;LM=1119978279;GM=1;S=7NmjkGkIc845ngM; rememberme=false

HTTP/1.1 200 OK
Transfer-Encoding: chunked
Date: Mon, 11 Jul 2005 08:21:03 GMT
Content-Type: text/html
Cache-Control: private
Server: GWS/2.1
Via: 1.1 cache1 (NetCache NetApp/5.5R2D5), Version 2.0-Build_Linux_1336 $Date: 04/13/2005 15:53:0038$(IWSS), 1.1 cache1 (NetCache NetApp/5.5R2D5)

e
```

Buttons: Find, Save As, Print, Entire conversation (767 bytes), Filter Out This Stream, Close

Analyzing Packets (8/9)

- Filter

The image shows a Wireshark window titled "Snif2 --- HTTP Example.cap - Wireshark". The filter bar at the top contains the expression "(tcp.stream eq 5)". The packet list shows 12 packets, all of which are highlighted in green. The details pane for the selected packet (No. 39) shows the following layers:

- Frame 39 (78 bytes on wire, 78 bytes captured)
- Ethernet II, Src: IbmL42:c2:4d (00:09:6b:42:c2:4d), Dst: LucentTe_cf:cd:2c (00:30:6d:cf:cd:2c)
- Internet Protocol, Src: 10.114.30.180 (10.114.30.180), Dst: 64.233.183.99 (64.233.183.99)
- Transmission Control Protocol, Src Port: peport (1449), Dst Port: http (80), Seq: 0, Len: 0

The packet bytes pane shows the raw data of the packet, which is a TCP segment. The status bar at the bottom indicates "Packets: 1648 Displayed: 12 Marked: 0".

Analyzing Packets (9/9)

- RTP

The image shows the Wireshark network protocol analyzer interface. The main window displays a list of captured packets, with packet 1001 selected. The packet details pane shows the structure of the packet: Ethernet II, Internet Protocol, User Datagram Protocol, and Real-time Transport Protocol (RTP). The RTP packet is of type 102 (DYNAMIC) and has a sequence number of 33244.

A red box highlights the 'Stream Analysis...' option in the 'Telephony' menu. This opens the 'Wireshark: RTP Stream Analysis' window, which provides a detailed view of the RTP stream. The window shows the stream is from 192.168.2.100 port 5004 to 78.136.29.109 port 6062, with SSRC = 0x39FF6709. The stream analysis table shows a list of packets with their sequence numbers, delta times, jitter, skew, and IP bandwidth.

Packet	Sequence	Delta(ms)	Filtered Jitter(ms)	Skew(ms)	IP BW(kbps)	Marker	Status
1417	19063	0.00	0.00	0.00	24.48		[Ok]
1419	19064	0.00	0.00	0.00	24.48		[Ok]
1421	19065	0.00	0.00	0.00	24.48		[Ok]
1423	19066	0.00	0.00	0.00	24.48		[Ok]
1425	19067	0.00	0.00	0.00	24.48		[Ok]
1427	19068	0.00	0.00	0.00	24.48		[Ok]
1429	19069	0.00	0.00	0.00	24.48		[Ok]
1431	19070	0.00	0.00	0.00	24.48		[Ok]

Summary statistics at the bottom of the RTP Stream Analysis window:

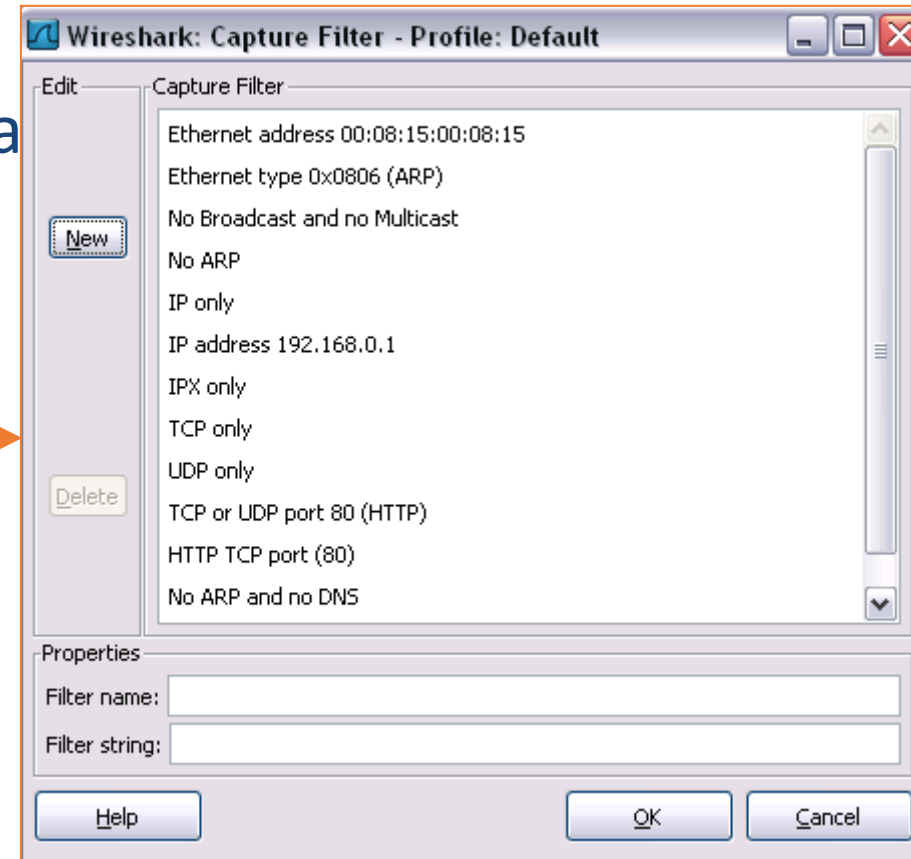
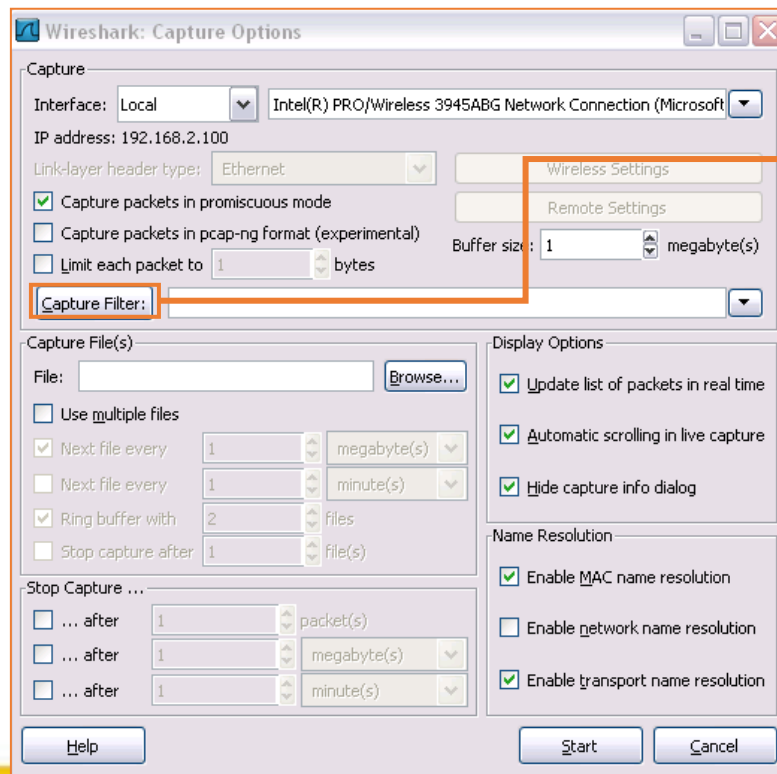
- Max delta = 0.00 ms at packet no. 0
- Max jitter = 0.00 ms. Mean jitter = 0.00 ms.
- Max skew = 0.00 ms.
- Total RTP packets = 2098 (expected 2098) Lost RTP packets = 0 (0.00%) Sequence errors = 0
- Duration 62.85 s (0 ms clock drift, corresponding to 1 Hz (+0.00%))

Buttons at the bottom: Save payload..., Save as CSV..., Refresh, Jump to, Graph, Next non-Ok, Close.

Stable
stream BW

Filtering Packets (1/4)

- Applying Filter when Capturing Pa
Capture → Interfaces → Options:



Filtering Packets (2/4)



Example 003.pcap - Wireshark

File Edit View Go Capture Analyze Statistics Telephony Tools Help

Filter: Expression... Clear Apply

No.	Time	Source	Destination	Protocol	Info
485	47.639995	192.168.2.100	212.143.162.152	TCP	17231 http [FIN, ACK] Seq=1409 Ack=380 Win=12810
486	47.649881	192.168.2.100	212.143.162.152	TCP	17241 http [SYN] Seq=0 Win=65535 Len=0 MSS=1460
487	47.666485	212.143.162.152	192.168.2.100	TCP	http > 17237 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0
488	47.666530	192.168.2.100			
489	47.666898	192.168.2.100			
490	47.667431	192.168.2.100			
491	47.675090	212.143.162.152			
492	47.675136	192.168.2.100			
493	47.677582	212.143.162.152			
494	47.677624	192.168.2.100			
495	47.677858	192.168.2.100			
496	47.695323	212.143.162.152			
497	47.697056	212.143.162.152			
498	47.737850	212.143.162.152			
499	47.739896	212.143.162.152			
500	47.788636	212.143.162.152			
501	47.797761	192.168.2.100			
502	47.826481	212.143.162.152			
503	47.826527	192.168.2.100			
504	47.826913	192.168.2.100			
505	47.839189	212.143.162.152			
506	47.841074	212.143.162.152			
507	47.856460	212.143.162.152			
508	47.858425	212.143.162.152			

Wireshark: Filter Expression - Profile: Default

Field name

- message/http - Media Type: message/http
- Messenger - Microsoft Messenger Service
- MGCP - Media Gateway Control Protocol
- MGMT - DCE/RPC Remote Management
- MIKEY - Multimedia Internet KEYing
- MIME multipart - MIME Multipart Media Encapsulation
- MIOP - Unreliable Multicast Inter-ORB Protocol
- MIPv6 - Mobile IPv6 / Network Mobility
- MMS - MMS
- MMSE - MMS Message Encapsulation
- Mobile IP - Mobile IP
- Modbus/TCP - Modbus/TCP

Relation

- is present
- ==
- !=
- >
- <
- >=
- <=
- contains
- matches

Value (character string)

Predefined values:

Range (offset:length)

OK Cancel

Frame 485 (54 bytes on wire, 54 bytes captured)

Ethernet II, Src: IntelCor_a2:d8:9a (00:1c:bf:a2:d8:9a), Dst: 212.143.162.152 (01:00:0c:00:00:00)

Internet Protocol, Src: 192.168.2.100 (192.168.2.100), Dst: 212.143.162.152 (212.143.162.152)

Transmission Control Protocol, Src Port: 17231 (17231), Dst Port: 80 (80)

0000 00 0e 2e 6e 2f 7d 00 1c bf a2 d8 9a 08 00 45 00

0010 00 28 94 52 40 00 80 06 2c 49 c0 a8 02 64 d4 80

0020 a2 98 43 4f 00 50 b6 d6 a4 44 e9 7c 0f 5f 50 10

0030 fa 32 e3 d5 00 00

Filtering Packets (3/4)

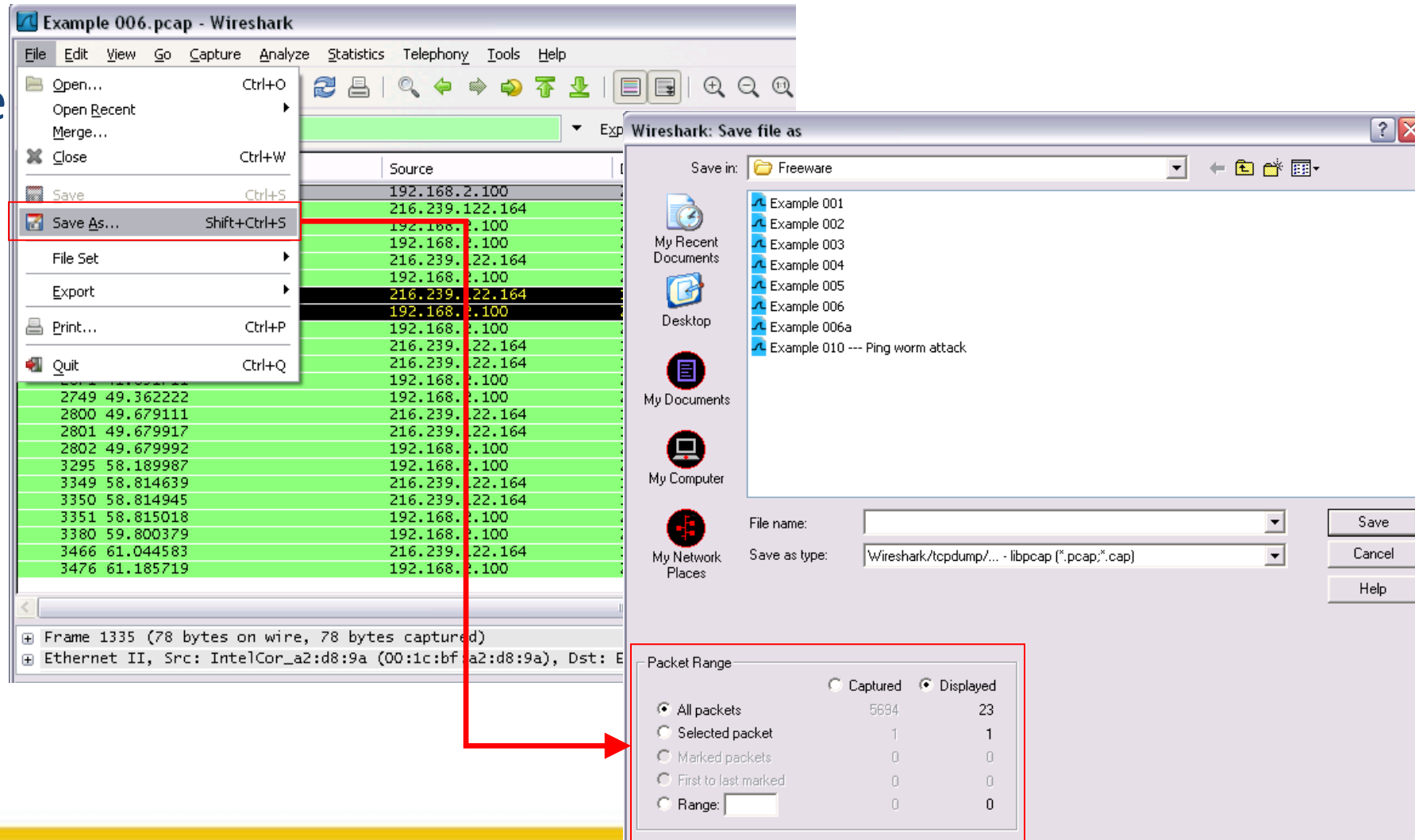
- Examples:
 - Capture only traffic to or from IP address 172.18.5.4
 - **host 172.18.5.4**
 - Capture traffic to or from a range of IP addresses
 - **net 192.168.0.0/24**
 - **net 192.168.0.0 mask 255.255.255.0**
 - Capture traffic from a range of IP addresses
 - **src net 192.168.0.0/24**
 - **src net 192.168.0.0 mask 255.255.255.0**
 - Capture traffic to a range of IP addresses
 - **dst net 192.168.0.0/24**
 - **dst net 192.168.0.0 mask 255.255.255.0**
 - Capture only DNS (port 53) traffic
 - **port 53**
 - Capture non-HTTP and non-SMTP traffic on your server
 - **host www.example.com and not (port 80 or port 25)**
 - **host www.example.com and not port 80 and not port 25**

Filtering Packets (4/4)

- Examples:
 - Capture except all ARP and DNS traffic
 - **port not 53 and not arp**
 - Capture traffic within a range of ports
 - **(tcp[2:2] > 1500 and tcp[2:2] < 1550) or (tcp[4:2] > 1500 and tcp[4:2] < 1550)**
 - **tcp portrange 1501-1549**
 - Capture only Ethernet type EAPOL
 - **ether proto 0x888e**
 - Capture only IP traffic
(the shortest filter, but sometimes very useful to get rid of lower layer protocols like ARP and STP)
 - **ip**
 - Capture only unicast traffic
(useful to get rid of noise on the network if you only want to see traffic to and from your machine, not, for example, broadcast and multicast announcements)
 - **not broadcast and not multicast**

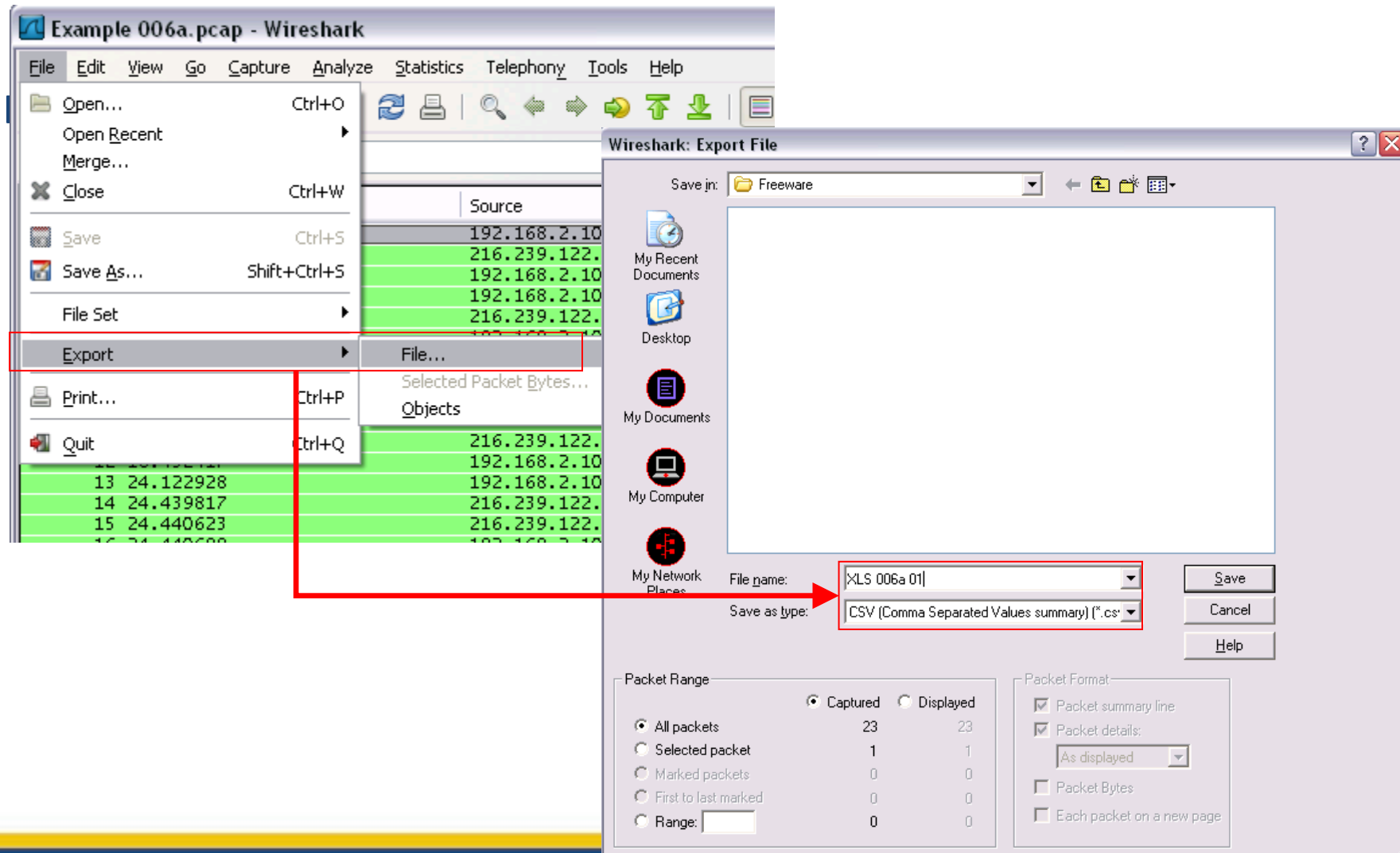
Saving and Manipulating Packets (1/3)

- Save



Saving and Manipulating Packets (2/3)

- Export



Saving and Manipulating Packets (3/3)

- Export

No.	Time	Time Variation	Source	Destination	Protocol	Info
1	0	0	192.168.2.100	216.239.122.164	TCP	27837 > http [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=1 TSV=0 TSER=0
2	0.226724	0.226724	216.239.122.164	192.168.2.100	TCP	http > 27837 [SYN, ACK] Seq=0 Ack=1 Win=8190 Len=0 MSS=1380
3	0.226772	4.8E-05	192.168.2.100	216.239.122.164	TCP	27837 > http [ACK] Seq=1 Ack=1 Win=65535 Len=0
4	0.227146	0.227098	192.168.2.100	216.239.122.164	HTTP	GET /i/b.jpg HTTP/1.1
5	0.700674	0.473576	216.239.122.164	192.168.2.100	HTTP	HTTP/1.1 200 OK (JPEG JFIF image)
6	0.883533	0.409957	192.168.2.100	216.239.122.164	TCP	27837 > http [ACK] Seq=649 Ack=767 Win=64769 Len=0
7	1.161312	0.751355	216.239.122.164	192.168.2.100	HTTP	[TCP Retransmission] HTTP/1.1 200 OK (JPEG JFIF image)
8	1.161361	0.410006	192.168.2.100	216.239.122.164	TCP	[TCP Dup ACK 6#1] 27837 > http [ACK] Seq=649 Ack=767 Win=64769 Len=0
9	16.211468	15.801462	192.168.2.100	216.239.122.164	HTTP	GET /i/b.jpg HTTP/1.1
10	16.452024	0.650562	216.239.122.164	192.168.2.100	TCP	[TCP segment of a reassembled PDU]
11	16.452343	15.801781	216.239.122.164	192.168.2.100	HTTP	HTTP/1.1 200 OK (JPEG JFIF image)
12	16.452417	0.650636	192.168.2.100	216.239.122.164	TCP	27837 > http [ACK] Seq=1539 Ack=1533 Win=65535 Len=0
13	24.122928	23.472292	192.168.2.100	216.239.122.164	HTTP	GET /i/b.jpg HTTP/1.1
14	24.439817	0.967525	216.239.122.164	192.168.2.100	TCP	[TCP segment of a reassembled PDU]
15	24.440623	23.473098	216.239.122.164	192.168.2.100	HTTP	HTTP/1.1 200 OK (JPEG JFIF image)
16	24.440698	0.9676	192.168.2.100	216.239.122.164	TCP	27837 > http [ACK] Seq=2384 Ack=2299 Win=64769 Len=0
17	32.950693	31.983093	192.168.2.100	216.239.122.164	HTTP	GET /i/b.jpg HTTP/1.1
18	33.575345	1.592252	216.239.122.164	192.168.2.100	TCP	[TCP segment of a reassembled PDU]
19	33.575651	31.983399	216.239.122.164	192.168.2.100	HTTP	HTTP/1.1 200 OK (JPEG JFIF image)
20	33.575724	1.592325	192.168.2.100	216.239.122.164	TCP	27837 > http [ACK] Seq=3269 Ack=3065 Win=65535 Len=0
21	34.561085	32.96876	192.168.2.100	216.239.122.164	HTTP	GET /b.gif HTTP/1.1
22	35.805289	2.836529	216.239.122.164	192.168.2.100	HTTP	HTTP/1.1 200 OK (GIF89a)
23	35.946425	33.109896	192.168.2.100	216.239.122.164	TCP	27837 > http [ACK] Seq=4080 Ack=3567 Win=65033 Len=0

Packet Statistics (1/8)

• Protocol

Wireshark: Protocol Hierarchy Statistics

Display filter: none

Protocol	% Packets	Packets	Bytes	Mbit/s	End Packets	End Bytes	End Mbit/s
[-] Frame	100.00 %	1276	385508	0.030	0	0	0.000
[-] Ethernet	100.00 %	1276	385508	0.030	0	0	0.000
[-] Internet Protocol	99.69 %	1272	385340	0.030	0	0	0.000
Internet Control Message Protocol	8.23 %	105	7770	0.001	105	7770	0.001
[-] User Datagram Protocol	4.86 %	62	11029	0.001	0	0	0.000
Simple Network Management Protocol	1.57 %	20	1825	0.000	20	1825	0.000
Bootstrap Protocol	0.31 %	4	1864	0.000	4	1864	0.000
NetBIOS Name Service	0.24 %	3	276	0.000	3	276	0.000
Domain Name Service	2.66 %	34	6922	0.001	34	6922	0.001
Data	0.08 %	1	142	0.000	1	142	0.000
[-] Transmission Control Protocol	86.60 %	1105	366541	0.028	658	149347	0.011
Post Office Protocol	1.41 %	18	1486	0.000	18	1486	0.000
[-] Hypertext Transfer Protocol	33.62 %	429	215708	0.017	388	188597	0.014
Line-based text data	2.27 %	29	21877	0.002	29	21877	0.002
JPEG File Interchange Format	0.08 %	1	59	0.000	1	59	0.000
eXtensible Markup Language	0.39 %	5	2730	0.000	5	2730	0.000
Media Type	0.16 %	2	1160	0.000	2	1160	0.000
Compuserve GIF	0.31 %	4	1285	0.000	4	1285	0.000
Address Resolution Protocol	0.31 %	4	168	0.000	4	168	0.000

Help Close

Packet Statistics (2/8)

- Con
- Tr

Conversations: (Untitled)

Ethernet: 4 Fibre Channel FD **IPv4: 38** IXX JXTA NCP RSVP SCTP **TCP: 79** Token Ring **UDP: 23** USB WLAN

IPv4 Conversations

Address A	Address B	Packets	Bytes	Packets A->B	Bytes A->B	Packets A<-B	Bytes A<-B	Rel Start	Duration
192.168.2.100	255.255.255.255	1	142	1	142	0	0	96.384838000	0.0000
192.168.2.101	255.255.255.255	2	684	2	684	0	0	47.852757000	3.0721
192.168.2.1	255.255.255.255	2	1180	2	1180	0	0	47.857905000	3.0722
192.168.2.100	212.143.162.144	10	2194	6	815	4	1379	87.473054000	65.0876
192.168.2.100	212.179.31.90	10	1342	6	971	4	371	91.655266000	60.9113
62.90.102.31	192.168.2.100	10	1373	4	441	6	932	91.660203000	60.9174
192.168.2.100	212.150.22.226	10	1327	6	956	4	371	91.732692000	60.8200
192.168.2.100	212.150.236.220	10	1470	6	981	4	489	91.742363000	60.8122
192.168.2.100	212.179.58.84	10	1725	6	954	4	771	92.287214000	2.4984
82.80.238.109	192.168.2.100	11	1282	5	440	6	842	91.646648000	60.9214
10.12.44.2	192.168.2.100	12	888	6	444	6	444	31.421091000	164.384
10.10.10.2	192.168.2.100	12	888	6	444	6	444	31.531676000	164.804
10.12.20.2	192.168.2.100	12	888	6	444	6	444	5.250457000	164.523
10.31.68.1	192.168.2.100	12	888	6	444	6	444	5.578447000	164.631
10.100.102.2	192.168.2.100	12	888	6	444	6	444	17.858674000	164.363

☒ Name resolution ☐ Limit to display filter

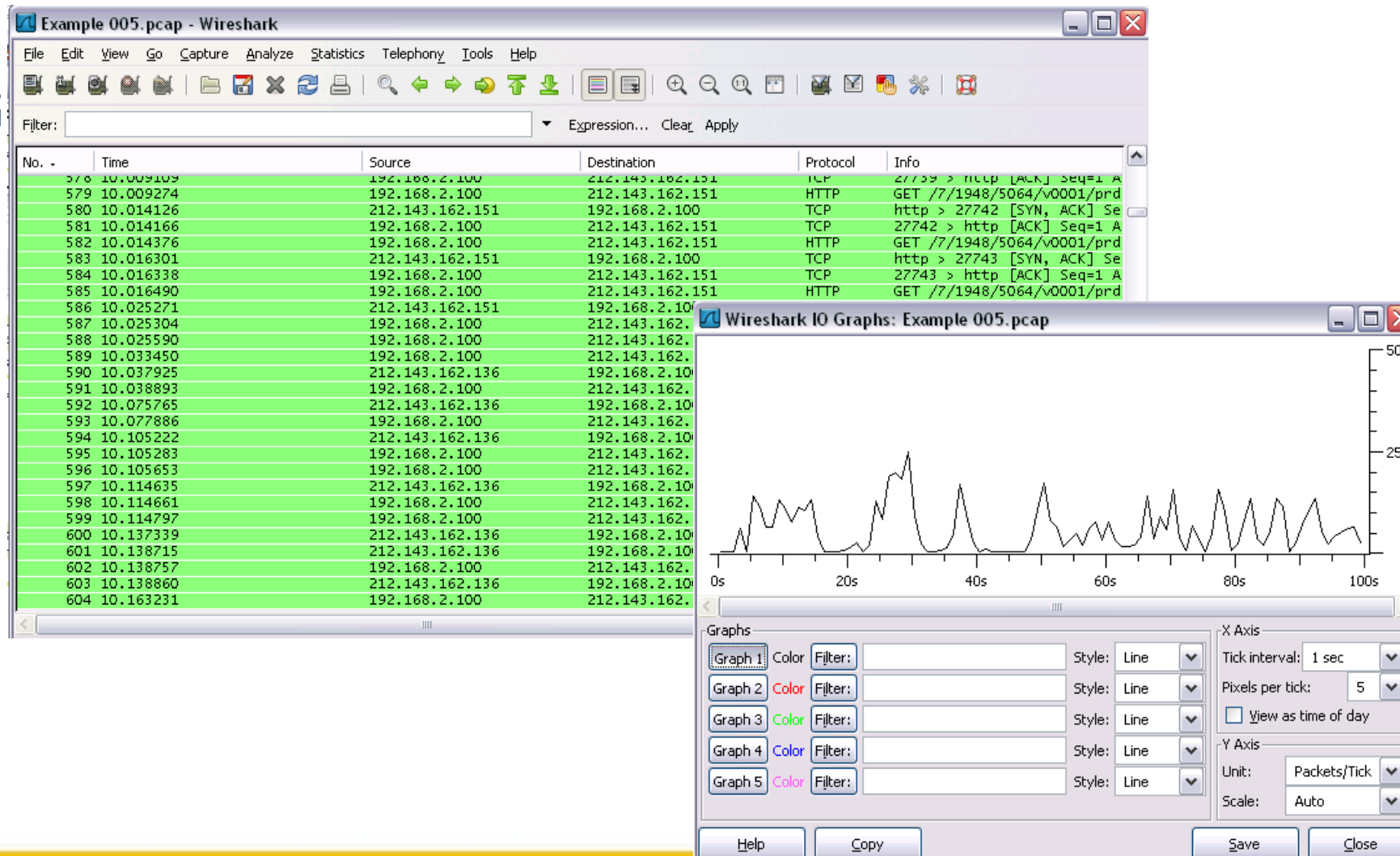
Help Copy Close

With some manipulation

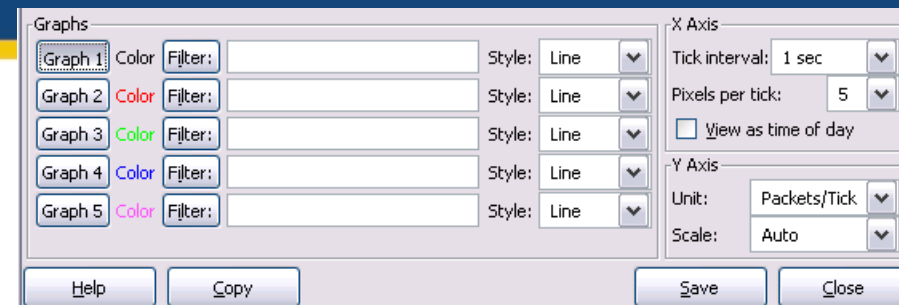
	A	B	C	D	E	F	G	H	I	J	K	L
1	Address A	Address B	Packets	Bytes	Packets A->B	Bytes A->B	Packets A<-B	Bytes A<-B	Rel Start	Duration	bps A->B	bps A<-B
2	10.10.10.1	10.159.3.103	2	124	0	0	2	124	0	42.0039	N/A	23.62
3	1.1.1.1	10.159.3.103	2	120	0	0	2	120	24.49414	1.1885	N/A	807.76
4	1.1.1.1	127.0.0.1	4	248	4	248	0	0	10.713867	14.9814	132.43	N/A
5	10.159.3.103	212.179.1.202	491	458158	185	10643	306	447515	23.216796	15.4082	5525.89	232351.54

Packet Statistics (3/8)

- I/O G



Packet Statistics (4/8)



- Configurable Options
 - I/O Graphs
 - **Graph 1-5:** enable the specific graph 1-5 (graph 1 by default)
 - **Filter:** a display filter for this graph (only the packets that pass this filter will be taken into account for this graph)
 - **Style:** the style of the graph (Line/Impulse/FBar/Dot)
 - X Axis
 - **Tick interval:** an interval in x direction lasts (10/1 minutes or 10/1/0.1/0.01/0.001 seconds)
 - **Pixels per tick:** use 10/5/2/1 pixels per tick interval
 - **View as time of day:** option to view x direction labels as time of day instead of seconds or minutes since beginning of capture
 - Y Axis
 - **Unit:** the unit for the y direction (Packets/Tick, Bytes/Tick, Bits/Tick, Advanced...)
 - **Scale:** the scale for the y unit (Logarithmic, Auto, 10, 20, 50, 100, 200, ...)

Packet Statistics (5/8)

• TCP

The screenshot shows the Wireshark interface with the 'Statistics' menu open. The 'TCP Stream Graph' option is selected, which has opened a sub-menu with the following options:

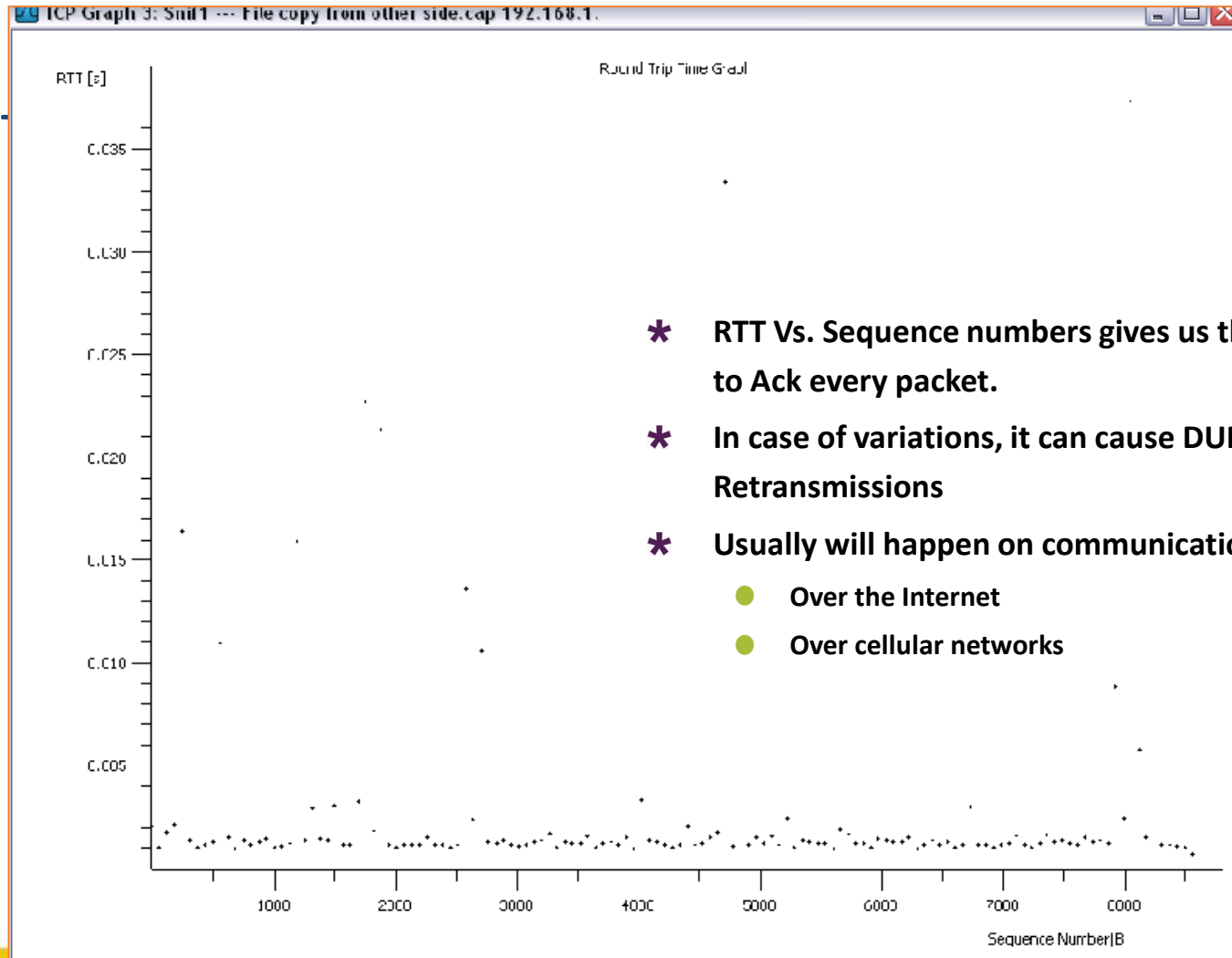
- Round Trip Time Graph
- Throughput Graph
- Time-Sequence Graph (Stevens)
- Time-Sequence Graph (tcptrace)

The main packet list on the left shows a filter of '(tcp.stream eq 0)'. The packet details pane on the right shows the selected packet (No. 771) with the following details:

- Frame 771 (60 bytes on wire, 60 bytes captured)
- Ethernet II, Src: Intel_4c:cc:89 (00:d0:b7:4c:cc:89), Dst: Intel_2e:32:a9 (00:90:27:2e:32:a9)
- Internet Protocol, Src: 192.168.1.102 (192.168.1.102), Dst: 192.168.104.77 (192.168.104.77)
- Transmission Control Protocol, Src Port: paradym-31port (1864), Dst Port: microsoft-ds (445), Seq: 883, Ack: 695704, Len: 0

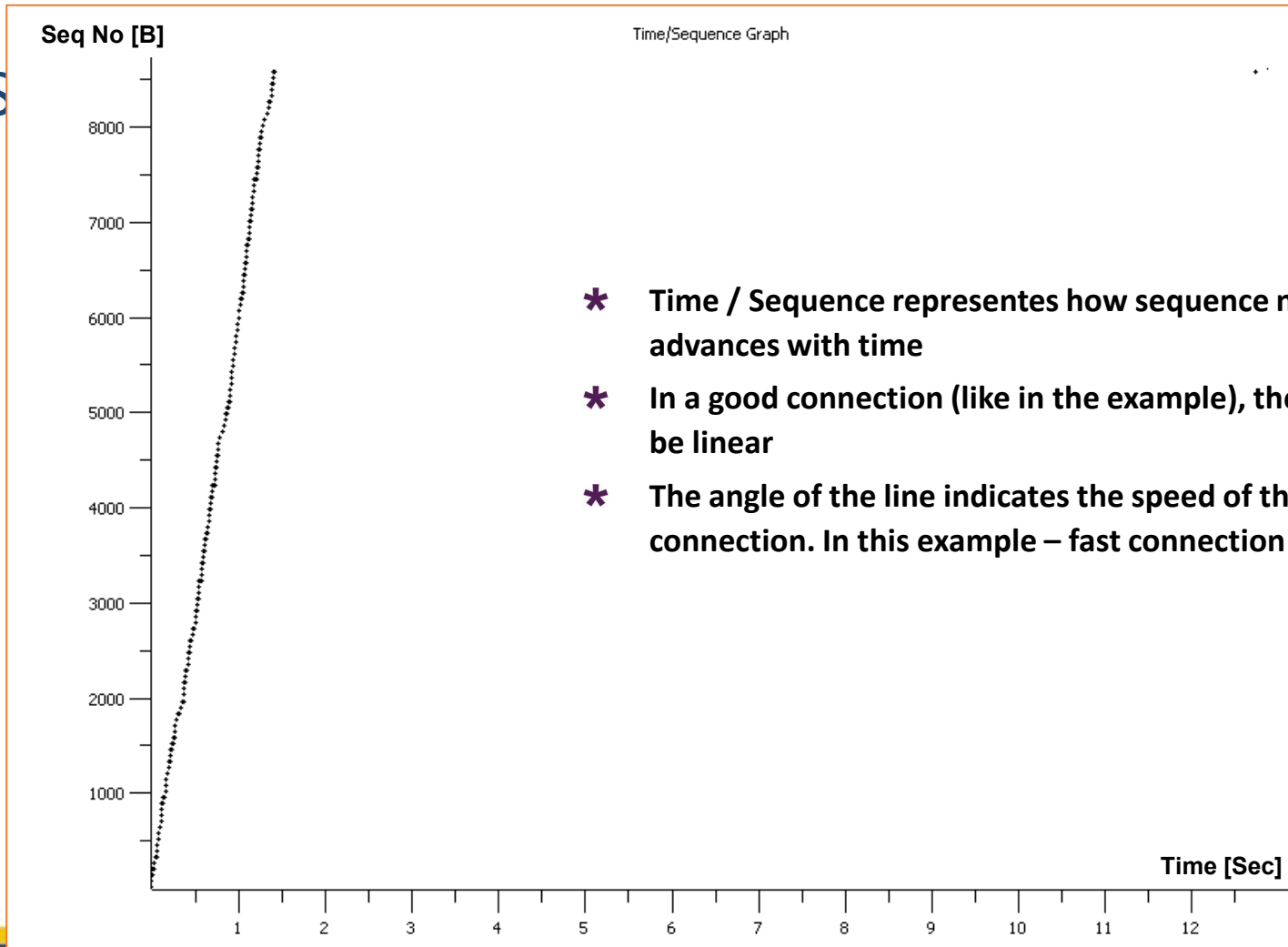
Packet Statistics (6/8)

- Round-



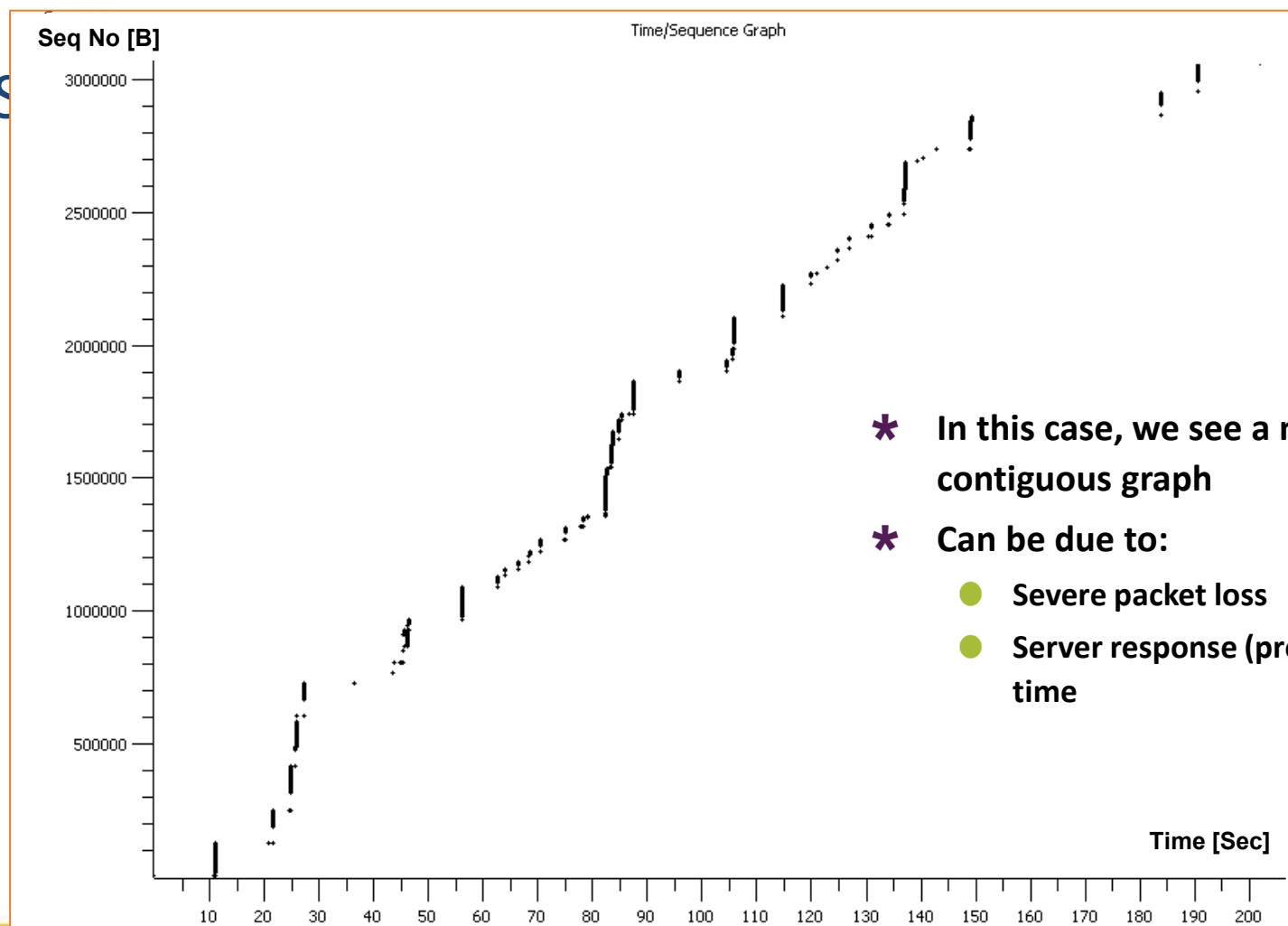
Packet Statistics (7/8)

- Time / S



Packet Statistics (8/8)

- Time / S




- * In this case, we see a non-contiguous graph
- * Can be due to:
 - Severe packet loss
 - Server response (processing) time

Colorizing Specific Packets (1/4)

- Colorize packets according to a filter
 - Allow to emphasize the packets interested in
 - A lot of Coloring Rule examples at the Wireshark Wiki
- Coloring Rules page at <http://wiki.wireshark.org/ColoringRules>

We want to watch a specific protocol through out the capture file



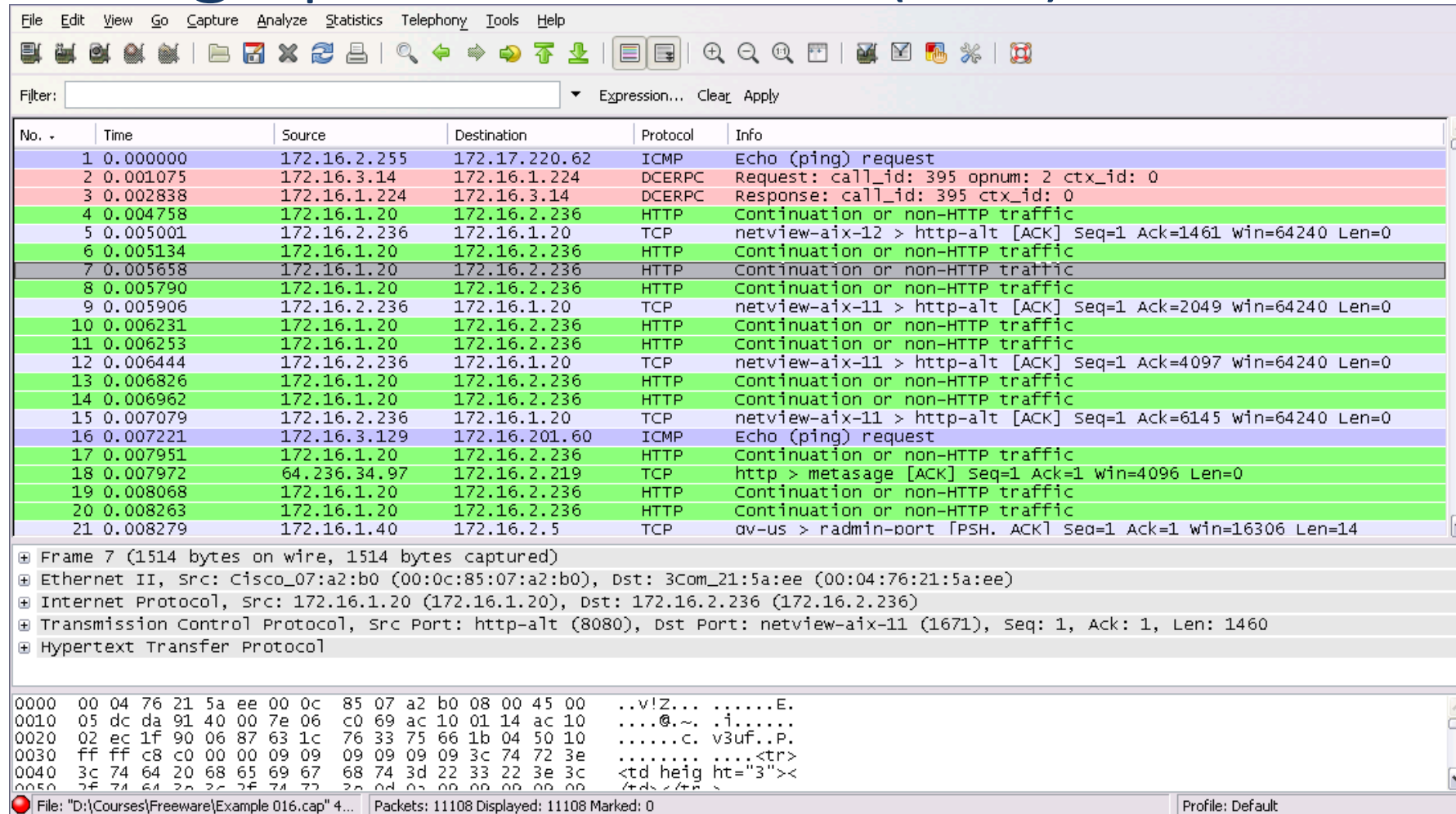
No. -	Time	Source	Destination	Protocol	Info
1	0.000000	172.16.2.255	172.17.220.62	ICMP	Echo (ping) request
2	0.001075	172.16.3.14	172.16.1.224	DCERPC	Request: call_id: 395 opnum: 2 ctx_id: 0
3	0.002838	172.16.1.224	172.16.3.14	DCERPC	Response: call_id: 395 ctx_id: 0
4	0.004738	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
5	0.005001	172.16.2.236	172.16.1.20	TCP	netview-aix-12 > http-alt [ACK] Seq=1 Ack=1461 win=64240 Len=0
6	0.005134	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
7	0.005658	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
8	0.005790	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
9	0.005906	172.16.2.236	172.16.1.20	TCP	netview-aix-11 > http-alt [ACK] Seq=1 Ack=2049 win=64240 Len=0
10	0.006231	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
11	0.006253	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
12	0.006444	172.16.2.236	172.16.1.20	TCP	netview-aix-11 > http-alt [ACK] Seq=1 Ack=4097 win=64240 Len=0
13	0.006826	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
14	0.006962	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
15	0.007079	172.16.2.236	172.16.1.20	TCP	netview-aix-11 > http-alt [ACK] Seq=1 Ack=6145 win=64240 Len=0
16	0.007221	172.16.3.129	172.16.201.60	ICMP	Echo (ping) request
17	0.007951	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
18	0.007972	64.236.34.97	172.16.2.219	TCP	http > metasage [ACK] Seq=1 Ack=1 win=4096 Len=0
19	0.008068	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
20	0.008263	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
21	0.008279	172.16.1.40	172.16.2.5	TCP	av-us > radmin-port [PSH, ACK] Seq=1 Ack=1 win=16306 Len=14

Colorizing Specific Packets (2/4)

The image shows the Wireshark network protocol analyzer interface. The main pane displays a list of captured packets. The 'Info' pane on the right shows the details of the selected packet (packet 21, a DCE RPC Request). A context menu is open over the packet list, with the 'Colorize Conversation' option highlighted. This option has opened a submenu where a list of color rules (Color 1 through Color 10) is displayed. The status bar at the bottom indicates that 11108 packets are displayed and 11108 are marked.

No.	Time	Source	Destination	Protocol	Info
1	0.000000	172.16.2.255	172.17.220.62	ICMP	Echo (ping) request
2	0.001075	172.16.3.14	172.16.1.224	DCERPC	Request: call id: 395 num: 2 ctx_id: 0
3	0.002838	172.16.1.224	172.16.3.14	DCERPC	Response: call id: 395 num: 2 ctx_id: 0
4	0.004758	172.16.1.20	172.16.2.236	HTTP	GET / HTTP/1.1
5	0.005001	172.16.2.236	172.16.1.20	TCP	64.236.34.97 [ACK] Seq=1 Ack=1461 win=64240 Len=0
6	0.005134	172.16.1.20	172.16.2.236	HTTP	200 OK HTTP/1.1
7	0.005658	172.16.1.20	172.16.2.236	HTTP	200 OK HTTP/1.1
8	0.005790	172.16.1.20	172.16.2.236	HTTP	200 OK HTTP/1.1
9	0.005906	172.16.2.236	172.16.1.20	TCP	64.236.34.97 [ACK] Seq=1 Ack=2049 win=64240 Len=0
10	0.006231	172.16.1.20	172.16.2.236	HTTP	200 OK HTTP/1.1
11	0.006253	172.16.1.20	172.16.2.236	HTTP	200 OK HTTP/1.1
12	0.006444	172.16.2.236	172.16.1.20	TCP	64.236.34.97 [ACK] Seq=1 Ack=1 win=4096 Len=0
13	0.006826	172.16.1.20	172.16.2.236	HTTP	200 OK HTTP/1.1
14	0.006962	172.16.1.20	172.16.2.236	HTTP	200 OK HTTP/1.1
15	0.007079	172.16.2.236	172.16.1.20	TCP	64.236.34.97 [ACK] Seq=1 Ack=1 win=4096 Len=0
16	0.007221	172.16.3.129	172.16.201.60	ICMP	Echo (ping) request
17	0.007951	172.16.1.20	172.16.2.236	HTTP	200 OK HTTP/1.1
18	0.007972	64.236.34.97	172.16.2.219	TCP	64.236.34.97 [ACK] Seq=1 Ack=1 win=4096 Len=0
19	0.008068	172.16.1.20	172.16.2.236	HTTP	200 OK HTTP/1.1
20	0.008263	172.16.1.20	172.16.2.236	HTTP	200 OK HTTP/1.1
21	0.008279	172.16.1.40	172.16.2.5	TCP	64.236.34.97 [ACK] Seq=1 Ack=1 win=4096 Len=0

Colorizing Specific Packets (3/4)



File Edit View Go Capture Analyze Statistics Telephony Tools Help

Filter: Expression... Clear Apply

No.	Time	Source	Destination	Protocol	Info
1	0.000000	172.16.2.255	172.17.220.62	ICMP	Echo (ping) request
2	0.001075	172.16.3.14	172.16.1.224	DCERPC	Request: call_id: 395 opnum: 2 ctx_id: 0
3	0.002838	172.16.1.224	172.16.3.14	DCERPC	Response: call_id: 395 ctx_id: 0
4	0.004758	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
5	0.005001	172.16.2.236	172.16.1.20	TCP	netview-aix-12 > http-alt [ACK] Seq=1 Ack=1461 win=64240 Len=0
6	0.005134	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
7	0.005658	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
8	0.005790	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
9	0.005906	172.16.2.236	172.16.1.20	TCP	netview-aix-11 > http-alt [ACK] Seq=1 Ack=2049 win=64240 Len=0
10	0.006231	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
11	0.006253	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
12	0.006444	172.16.2.236	172.16.1.20	TCP	netview-aix-11 > http-alt [ACK] Seq=1 Ack=4097 win=64240 Len=0
13	0.006826	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
14	0.006962	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
15	0.007079	172.16.2.236	172.16.1.20	TCP	netview-aix-11 > http-alt [ACK] Seq=1 Ack=6145 win=64240 Len=0
16	0.007221	172.16.3.129	172.16.201.60	ICMP	Echo (ping) request
17	0.007951	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
18	0.007972	64.236.34.97	172.16.2.219	TCP	http > metasage [ACK] Seq=1 Ack=1 win=4096 Len=0
19	0.008068	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
20	0.008263	172.16.1.20	172.16.2.236	HTTP	Continuation or non-HTTP traffic
21	0.008279	172.16.1.40	172.16.2.5	TCP	av-us > radmin-port [PSH. ACK] Seq=1 Ack=1 win=16306 Len=14

+ Frame 7 (1514 bytes on wire, 1514 bytes captured)
 + Ethernet II, Src: Cisco_07:a2:b0 (00:0c:85:07:a2:b0), Dst: 3Com_21:5a:ee (00:04:76:21:5a:ee)
 + Internet Protocol, Src: 172.16.1.20 (172.16.1.20), Dst: 172.16.2.236 (172.16.2.236)
 + Transmission Control Protocol, Src Port: http-alt (8080), Dst Port: netview-aix-11 (1671), Seq: 1, Ack: 1, Len: 1460
 + Hypertext Transfer Protocol

0000 00 04 76 21 5a ee 00 0c 85 07 a2 b0 08 00 45 00 ..v!Z... ..E.
 0010 05 dc da 91 40 00 7e 06 c0 69 ac 10 01 14 ac 10@.. .i.....
 0020 02 ec 1f 90 06 87 63 1c 76 33 75 66 1b 04 50 10C. v3uf..P.
 0030 ff ff c8 c0 00 00 09 09 09 09 09 3c 74 72 3e<tr>
 0040 3c 74 64 20 68 65 69 67 68 74 3d 22 33 22 3e 3c <td heig ht="3"><
 0050 2f 74 64 20 68 65 69 67 68 74 3d 22 33 22 3e 3c </td></tr>

File: "D:\Courses\Freeware\Example 016.cap" 4... Packets: 11108 Displayed: 11108 Marked: 0 Profile: Default

Colorizing Specific Packets (4/4)

- TLS

No. -	Time	Source	Destination	Protocol	Info
1	0.000000	192.168.2.100	198.65.166.131	UDP	Source port: 64064 Destination port: sip
2	1.708525	192.168.2.100	130.94.88.123	TCP	appworxsr > https [FIN, ACK] Seq=1 Ack=1 Win=64240 Len=0
3	1.709469	192.168.2.100	130.94.88.123	TCP	lv-jc > https [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=1 TSV=0 TSER=
4	2.001023	130.94.88.123	192.168.2.100	TCP	https > lv-jc [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0 MSS=1460 WS=2
5	2.001077	192.168.2.100	130.94.88.123	TCP	lv-jc > https [ACK] Seq=1 Ack=1 Win=128480 Len=0
6	2.001180	130.94.88.123	192.168.2.100	TCP	https > appworxsr [ACK] Seq=1 Ack=2 Win=3756 Len=0
7	2.001777	192.168.2.100	130.94.88.123	SSL	Client Hello
8	2.308152	130.94.88.123	192.168.2.100	TCP	https > lv-jc [ACK] Seq=1 Ack=103 Win=5840 Len=0
9	2.308490	130.94.88.123	192.168.2.100	TLSv1	Server Hello,
10	2.309543	130.94.88.123	192.168.2.100	TCP	[TCP segment of a reassembled PDU]
11	2.309618	192.168.2.100	130.94.88.123	TCP	lv-jc > https [ACK] Seq=103 Ack=2705 Win=128480 Len=0
12	2.617428	130.94.88.123	192.168.2.100	TCP	[TCP segment of a reassembled PDU]
13	2.619328	130.94.88.123	192.168.2.100	TLSv1	Certificate, Server Hello Done
14	2.619440	192.168.2.100	130.94.88.123	TCP	lv-jc > https [ACK] Seq=103 Ack=4549 Win=128480 Len=0
15	2.620478	192.168.2.100	130.94.88.123	TLSv1	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
16	2.922741	130.94.88.123	192.168.2.100	TLSv1	Change Cipher Spec, Encrypted Handshake Message
17	2.926069	192.168.2.100	130.94.88.123	TCP	[TCP segment of a reassembled PDU]
18	2.926211	192.168.2.100	130.94.88.123	TLSv1	Application Data
19	3.229909	130.94.88.123	192.168.2.100	TCP	https > lv-jc [ACK] Seq=4592 Ack=1782 Win=12320 Len=0
20	3.234770	130.94.88.123	192.168.2.100	TCP	[TCP segment of a reassembled PDU]
21	3.235519	130.94.88.123	192.168.2.100	TLSv1	Application Data
22	3.235588	192.168.2.100	130.94.88.123	TCP	lv-jc > https [ACK] Seq=1782 Ack=6110 Win=128480 Len=0
23	3.737122	192.168.2.100	130.94.88.123	TCP	[TCP segment of a reassembled PDU]
24	3.737295	192.168.2.100	130.94.88.123	TLSv1	Application Data
25	4.151556	130.94.88.123	192.168.2.100	TCP	https > lv-jc [ACK] Seq=6110 Ack=3261 Win=15024 Len=0
26	4.151984	130.94.88.123	192.168.2.100	TCP	[TCP segment of a reassembled PDU]
27	4.152276	130.94.88.123	192.168.2.100	TLSv1	Application Data
28	4.152370	192.168.2.100	130.94.88.123	TCP	lv-jc > https [ACK] Seq=3261 Ack=7776 Win=128480 Len=0
29	7.936331	192.168.2.100	212.150.49.10	DNS	Standard query A mail.barak.net.il
30	8.025917	212.150.49.10	192.168.2.100	DNS	Standard query response A 194.90.6.40
31	8.077161	192.168.2.100	194.90.6.40	TCP	dynamic3d > pop3 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=1 TSV=0 TS
32	8.098732	194.90.6.40	192.168.2.100	TCP	pop3 > dynamic3d [SYN, ACK] Seq=0 Ack=1 Win=49580 Len=0 TSV=8290107
33	8.098776	192.168.2.100	194.90.6.40	TCP	dynamic3d > pop3 [ACK] Seq=1 Ack=1 Win=128480 Len=0 TSV=7813 TSER=8
34	8.118204	194.90.6.40	192.168.2.100	POP	S: +OK POP3 service
35	8.118745	192.168.2.100	194.90.6.40	POP	C: USER yoram-ndi.co.il
36	8.138633	194.90.6.40	192.168.2.100	TCP	pop3 > dynamic3d [ACK] Seq=19 Ack=23 Win=49580 Len=0 TSV=829010732
37	8.140050	194.90.6.40	192.168.2.100	POP	S: +OK password required for user yoram-ndi.co.il

References

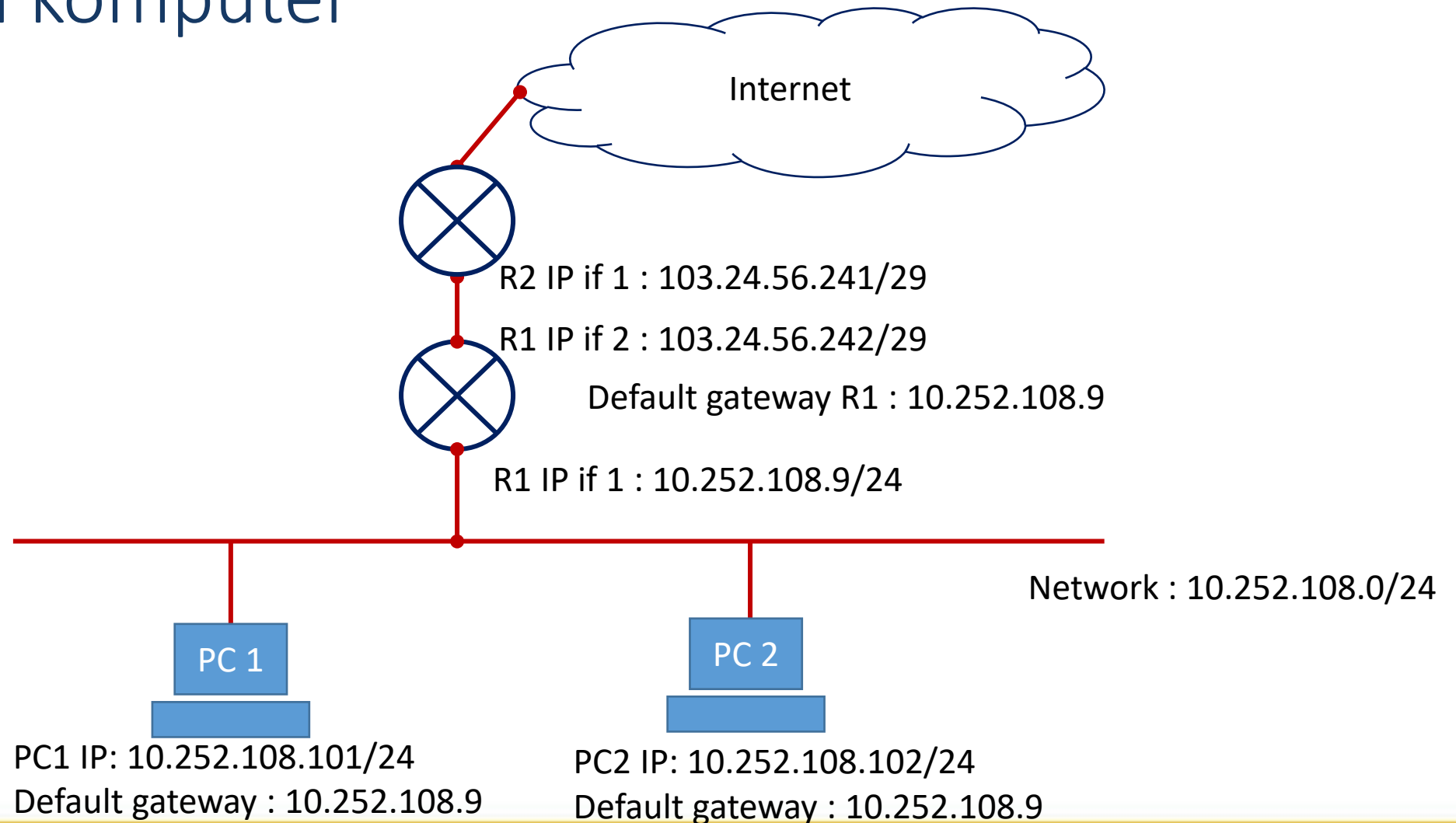
- Wireshark Website
 - <http://www.wireshark.org>
- Wireshark Documentation
 - <http://www.wireshark.org/docs/>
- Wireshark Wiki
 - <http://wiki.wireshark.org>
- Network analysis Using Wireshark Cookbook
 - <http://www.amazon.com/Network-Analysis-Using-Wireshark-Cookbook/dp/1849517649>



Perangkat Router

- Router merupakan peralatan jaringan yang bertugas untuk menghubungkan dua jaringan atau lebih
- Jenis perangkat router dapat berupa
 - komputer yang diaktifkan fungsi *packet forwarding* pada TCP/IP modulnya
 - peralatan khusus yang didesain untuk melakukan fungsi sebagai router
- Router meneruskan paket (packet forwarding) dengan menggunakan protokol routing (*routing protocol*)
- Routing protocol dibedakan menurut algoritma routingnya

Ilustrasi : topologi jaringan di laboratorium jaringan komputer



Referensi

- Aaaaaaaaaa
- Bbbbbbbb
- cccccc



bridge to the future

<http://www.eepis-its.edu>