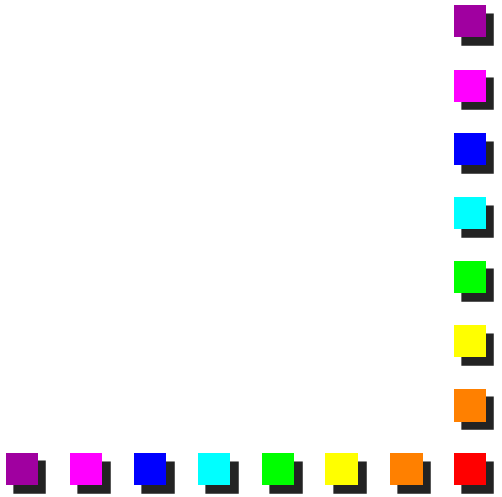
Packet processing with libpcap/WinPcap

Fulvio Risso

http://fulvio.frisso.net



Libpcap/WinPcap

- Libpcap is an open source C library that allows to capture/send raw packets from a Network Interface Card (NIC)
  - WinPcap is the Windows version
- Libpcap gives you access to exactly the packets that are being received/sent by **your NIC**
  - Be carefully that some packets may be filtered by your network switch!
- Main target are C programmers, but other languages (e.g., Java) may be supported through additional components
- This presentation will focus on the **capturing** capabilities of libpcap/WinPcap

Libpcap/Wincap do and don't

Does

- Capture raw packets
- Filter traffic
  - Capture a subset of traffic
- Supports reading traffic from physical interfaces and files
- Saves (filtered) traffic on file

Does not

- Parse packets
  - E.g., easy access to protocol fields (e.g., ip.src)
- Visualize packet content
- Support for post-capture filters
  - Unless data is saved on a file, which is then opened again (with a "capture" filter turned on)

tcpdump/WinDump

- Basic sniffer based on libpcap/WinDump
- Command line interface
- Often the only sniffer present when you install your Linux OS

```
C:\Users\Fulvio>windump -n -r sample.cap
reading from file sample.cap, link-type EN10MB (Ethernet)
13:55:56.141474 IP 130.192.3.76.138 > 130.192.3.255.138: UDP, length 201
13:55:56.141725 arp who-has 130.192.43.107 tell 130.192.4.34
13:55:56.239625 IP 130.192.25.132.138 > 130.192.25.255.138: UDP, length 232
13:55:56.338967 arp who-has 208.137.254.2 tell 130.192.3.24
13:55:56.439478 arp who-has 130.192.24.8 (ff:ff:ff:ff:ff:ff) tell 130.192.24.254
13:55:56.440093 arp who-has 130.192.24.16 (ff:ff:ff:ff:ff:ff) tell 130.192.24.254
13:55:56.440670 arp who-has 130.192.24.11 (ff:ff:ff:ff:ff:ff) tell 130.192.24.254
13:55:56.443275 router-hello l2rout vers 2 eco 0 ueco 0 src 60.29 blksize 1498 pri 64 hello 15
13:55:56.539253 IP 130.192.4.32.138 > 130.192.4.255.138: UDP, length 237
13:55:57.441384 IP 130.192.16.105.138 > 130.192.16.255.138: UDP, length 178
13:55:57.443861 IP 130.192.3.24.2301 > 255.255.255.255.2301: UDP, length 12
13:55:57.908574 IP 130.192.16.81.1210 > 130.192.3.21.53:  79+[|domain]
13:55:57.911728 IP 130.192.3.21.53 > 130.192.16.81.1210:  79*[|domain]
13:55:57.929764 IP 130.192.16.81 > 130.192.28.6: ICMP echo request, id 512, seq 432, length 40
13:55:57.931675 IP 130.192.28.6 > 130.192.16.81: ICMP echo reply, id 512, seq 432, length 40
```

tcpdump/Windump basics

- Some suggestions for the command line:
  - -D: prints the list of network interfaces available in the system
  - -i [interface]: captures from that network interface
  - -s [snaplen]: captures only the first [snaplen] bytes of each packet
    - Please set it to a large number to be sure that all the payload gets captured
    - Often modern NICs implement TCP reassembly, returning very large packets (e.g., up to 64KB) to the user
  - -n: do not resolve addresses into name (speeds up the capture and avoids generating DNS packets within the capture)
  - -r [filename]: reads packets from [filename] instead of NIC
  - -w [filename]: saves the read packet to [filename]
    - Can be used also to read from a first file, apply a filter, and save the result on a second filter
- Example

```
tcpdump –i eth0 –s 10000 – w sample.cap tcp port 80
```

Getting and installing the library

- Linux/UNIX
  - Binaries (libpcap + tcpdump) usually included in your distribution
  - May require administrative privileges (e.g., *sudo*)
- Windows
  - Need to download and install the binaries from http://www.winpcap.org
  - Two separate downloads for the library binary (a couple of DLL files) and tcpdump (a standalone executable)
- For developers
  - Need to install the developer version, which includes headers
  - Linux: often called *libpcap-dev*
  - Windows: install the "*Developer's Pack*"

6

CMake

- Cross-platform generator for project files (e.g., makefiles, Windows Studio projects, etc)
- Not strictly needed, but definitely useful in order to write cross-platform software
- Download and install from:
  - Linux: use your package manager
  - Windows: http://www.cmake.org

Create your first libpcap program

- Objective: open a capture file and print the hex dump of the packets on screen

```
C:\Users\Fulvio\test-libpcap\debug> readfile-ex sample.cap
1387179358:265840 (186)
00 0d 29 0b a4 3f 20 cf 30 14 c9 0d 08 00 45 10
00 ac 20 18 40 00 40 06 2e 00 82 c0 e1 46 82 c0
05 5d 00 16 c3 25 f2 8c 0b 21 f9 1a 05 22 50 18
00 50 ec c2 00 00 b2 76 79 0b 39 8f e8 00 c2 c3
d0 ae ec 6e b0 e7 a7 97 d1 66 da b5 b4 46 ab fd
e0 d0 d3 c6 37 e1 85 ea ca cf 2f da 2d 5e 10 7b
25 1c 90 63 e4 68 cc ca d6 42 32 9b df e6 f1 d6
59 41 55 74 e6 28 01 15 8d d8 c0 99 58 c2 95 45
c3 ea c4 66 b6 b7 ed 79 2d a2 fc 7d 20 0b a1 72
76 0f 37 50 93 fd 89 33 ba 31 39 2b f5 ba 13 35
a1 d1 76 53 c5 a7 60 7f ca dd fd d5 97 fa 76 4a
9f 11 b7 b5 5e 15 11 12 df 3c

1387179358:267072 (60)
20 cf 30 14 c9 0d 00 0d 29 0b a4 3f 08 00 45 00
00 28 7a f6 40 00 7e 06 95 b5 82 c0 05 5d 82 c0
e1 46 c3 25 00 16 f9 1a 05 22 f2 8c 0b a5 50 10
3f 79 c4 8c 00 00 00 00 b1 27 f5 f1
```

Create your first libpcap program

- Unpack the sample
- Create the project files:

```
C:\Users\Fulvio\test-libpcap> cmake .
```

- Compile the executable
  - Windows: open the Microsoft Studio Solution file (.sln) and compile the solution
  - Linux:

```
fulvio@zerg:~/test-libpcap$ make
```

- Launch the executable
  - Windows:

```
C:\Users\Fulvio\test-libpcap\debug> readfile-ex ..\sample.cap
```

  - Linux:

```
fulvio@zerg:~/test-libpcap$ ./readfile-ex sample.cap
```

The source code (1)

```
// Standard C include file for I/O functions
#include <stdio.h>

// Include files for libpcap/WinPcap functions
#include <pcap.h>

#define LINE_LEN 16

int main(int argc, char **argv)
{
    pcap_t *fp;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct pcap_pkthdr *header;
    const u_char *pkt_data;
    u_int i=0;
    int res;

    if (argc != 2)
    {
        printf("usage: %s filename\n\n", argv[0]);
        return -1;
    }

    /* Open the capture file */
    if ((fp = pcap_open_offline(argv[1],     // name of the device to open
                                errbuf       // error buffer
                                )) == NULL)
    {
        fprintf(stderr, "Unable to open the file %s.\n\n", argv[1]);
        return -1;
    }
```

The source code (2)

```
    /* Retrieve the packets from the file */
    while((res = pcap_next_ex(fp, &header, &pkt_data)) >= 0)
    {
        /* print pkt timestamp and pkt len */
        printf("%ld:%ld (%d)\n", header->ts.tv_sec, header->ts.tv_usec, header->len);

        /* Print the packet */
        for (i=1; (i < header->caplen + 1 ) ; i++)
        {
            printf("%.2x ", pkt_data[i-1]);

            if ( (i % LINE_LEN) == 0)
                printf("\n");
        }

        printf("\n\n");
    }

    if (res == -1)
    {
         printf("Error reading the packets: %s\n", pcap_geterr(fp));
    }

    pcap_close(fp);
    return 0;
}
```

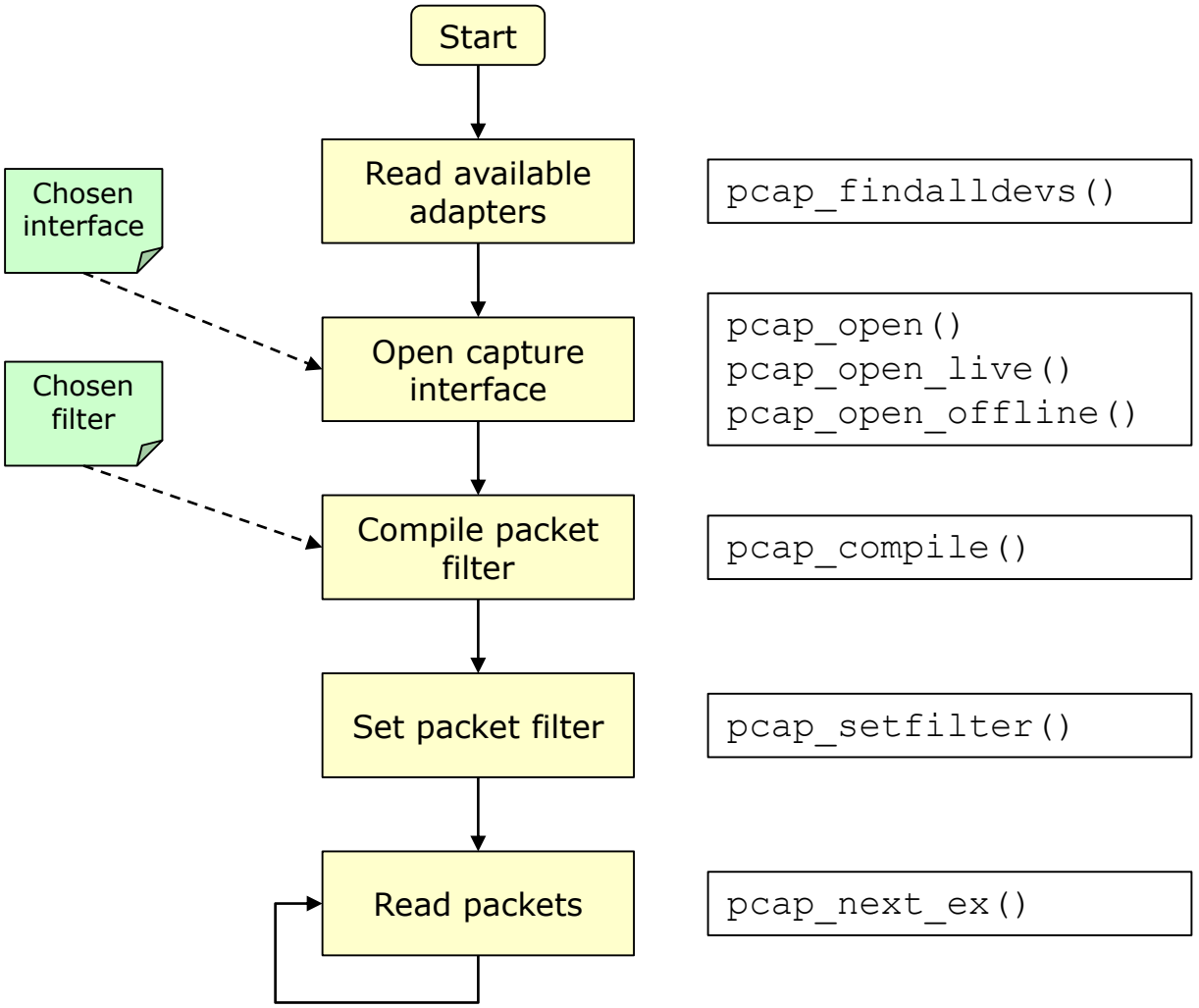
The project file

```
PROJECT(READFILE-SAMPLE)
CMAKE_MINIMUM_REQUIRED(VERSION 2.4)

# Set source files
SET(SOURCES
    readfile_ex.c
    )

# Default directories for include files
IF(WIN32)
    INCLUDE_DIRECTORIES (
        ${READFILE-SAMPLE_SOURCE_DIR}
        ${READFILE-SAMPLE_SOURCE_DIR}/../WPdPack/Include
        )
ELSE(WIN32)
    INCLUDE_DIRECTORIES (
        ${READFILE-SAMPLE_SOURCE_DIR}
        )
ENDIF(WIN32)

# Default directories for linking
#   (in Linux those are already in the system path)
IF(WIN32)
    LINK_DIRECTORIES(${READFILE-SAMPLE_SOURCE_DIR}/../WPdPack/Lib)
ENDIF(WIN32)
```

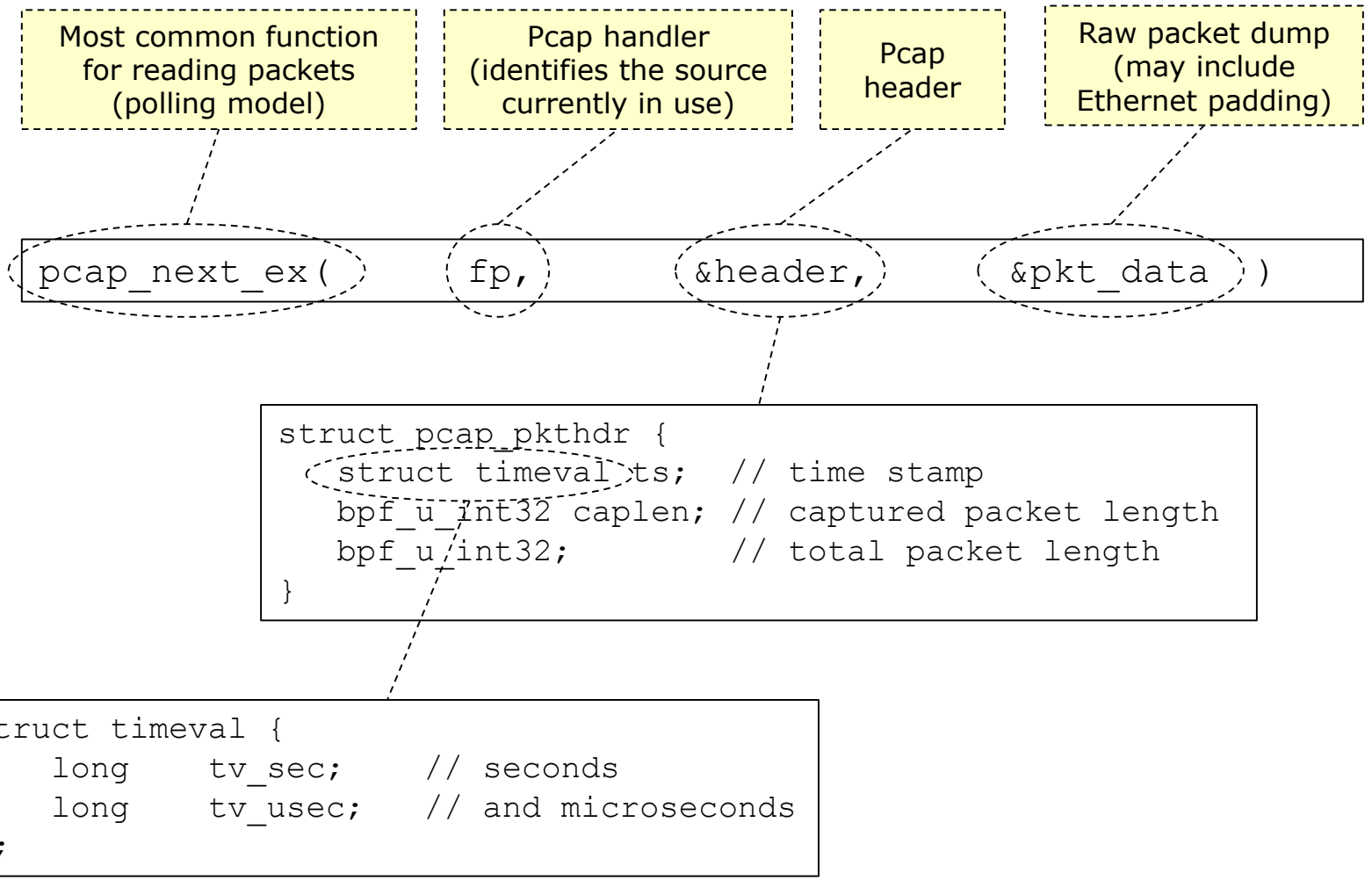
Assume the "WPdPack" folder is at the same level of our sample

```
# Platform-specific definitions
IF(WIN32)
    ADD_DEFINITIONS(
        -D_CRT_SECURE_NO_WARNINGS
        -D_CRT_SECURE_NO_DEPRECATE
        -DWIN32_LEAN_AND_MEAN
    )
ENDIF(WIN32)

# Create executable
ADD_EXECUTABLE(
    readfile-ex
    ${SOURCES}
)

# Link the executable to the
#   required libraries
IF(WIN32)
    TARGET_LINK_LIBRARIES(
        readfile-ex
        wpcap
        )
ELSE(WIN32)
    TARGET_LINK_LIBRARIES(
        readfile-ex
        pcap
        )
ENDIF(WIN32)
```

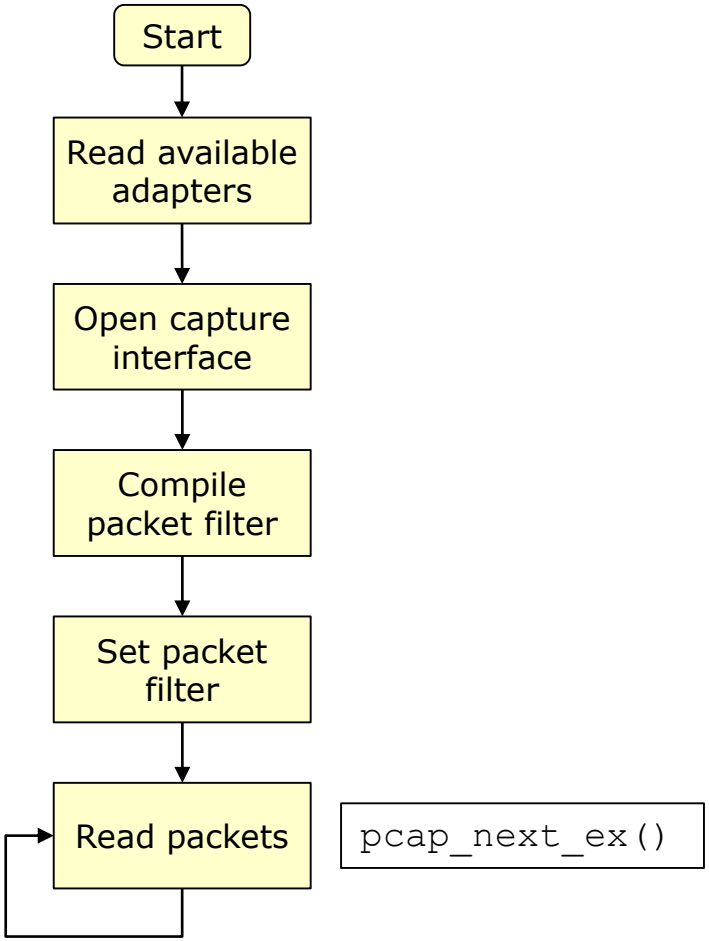
Libpcap programming basics: basic workflow

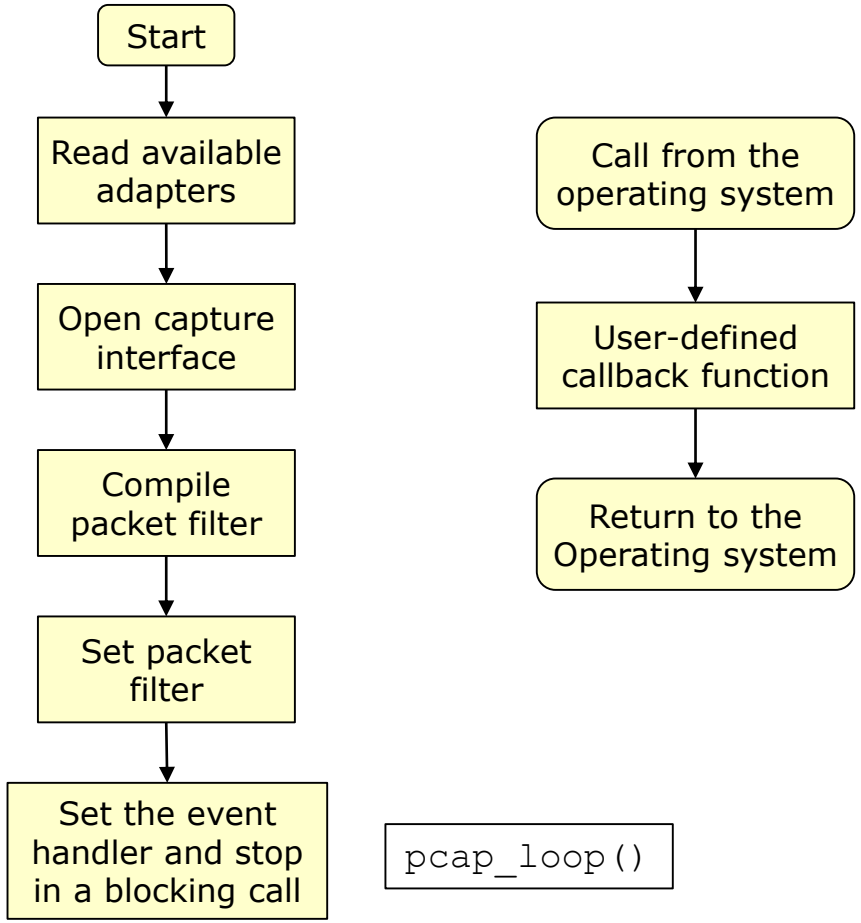


Libpcap programming basics: data structures



Polling vs. event-driven libpcap programming





Relevant source code for the “callback” mode

```
    /* Open the capture file */
    if ((fp = pcap_open_offline(argv[1],    // name of the device
                                errbuf      // error buffer
                                )) == NULL)
    {
        fprintf(stderr, "Unable to open the file %s.\n\n", argv[1]);
        return -1;
    }

    /* Register the callback handler and block until EOF is reached */
    /* Read and dispatch packets */
    pcap_loop(fp, 0, dispatcher_handler, NULL);

    pcap_close(fp);
    return 0;
}

void dispatcher_handler(u_char *temp1, const struct pcap_pkthdr *header, const u_char *pkt_data)
{
    u_int i=0;

    /* print pkt timestamp and pkt len */
    printf("%ld:%ld (%ld)\n", header->ts.tv_sec, header->ts.tv_usec, header->len);

    /* Print the packet */
    for (i=1; (i < header->caplen + 1 ) ; i++)
    {
        printf("%.2x ", pkt_data[i-1]);
        if ( (i % LINE_LEN) == 0) printf("\n");
    }

    printf("\n\n");
}
```

Parsing packets: defining protocol headers

Define a structure that helps parsing the Ethernet header; this is possible only because the Ethernet header has fixed length

Cast the raw packet (starting at offset zero) into an `ether_header` structure

```
struct ether_header {
    u_int8_t ether_dhost[6];    /* 6 bytes destination address */
    u_int8_t ether_shost[6];    /* 6 bytes source address */
    u_int16_t ether_type;       /* 2 bytes ethertype */
    };


int main(int argc, char **argv) { ... }


void dispatcher_handler(u_char *temp1, const struct pcap_pkthdr *header, const u_char *pkt_data)
{
    /* Pointer to the ether_header structure */
    struct ether_header *eptr;

    /* Cast the raw packet to the ether_header structure, in order to facilitate the parsing */
    eptr = (struct ether_header *) pkt_data;

    /* Print on screen the MAC addresses of each packet */
    printf("%02x:%02x:%02x:%02x:%02x:%02x --> %02x:%02x:%02x:%02x:%02x:%02x\n\n",
            eptr->ether_shost[0], eptr->ether_shost[1], eptr->ether_shost[2],
            eptr->ether_shost[3], eptr->ether_shost[4], eptr->ether_shost[5],
            eptr->ether_dhost[0], eptr->ether_dhost[1], eptr->ether_dhost[2],
            eptr->ether_dhost[3], eptr->ether_dhost[4], eptr->ether_dhost[5]);
}
```

Print the value in hex using two digits and prepend '0' if needed

Parsing packets: accessing to protocol fields

```
#ifdef WIN32
  #include <winsock2.h> /* Needed for ntohs() in Windows */
#endif

struct ether_header {
    u_int8_t ether_dhost[6];    /* 6 bytes destination address */
    u_int8_t ether_shost[6];    /* 6 bytes source address */
    u_int16_t ether_type;       /* 2 bytes ethertype */
    };

int main(int argc, char **argv) { ... }

void dispatcher_handler(u_char *temp1, const struct pcap_pkthdr *header, const u_char *pkt_data)
{
    struct ether_header *eptr; /* Pointer to the ether_header structure */
    u_int16_t ethertype;       /* Variable that keeps the ethertype in host-based format */

    /* Cast the raw packet to the ether_header structure, in order to facilitate the parsing */
    eptr = (struct ether_header *) pkt_data;

    /* Print on screen the MAC addresses of each packet */
    printf("%02x:%02x:%02x:%02x:%02x:%02x --> %02x:%02x:%02x:%02x:%02x:%02x",
             eptr->ether_shost[0], eptr->ether_shost[1], eptr->ether_shost[2],
             eptr->ether_shost[3], eptr->ether_shost[4], eptr->ether_shost[5],
             eptr->ether_dhost[0], eptr->ether_dhost[1], eptr->ether_dhost[2],
             eptr->ether_dhost[3], eptr->ether_dhost[4], eptr->ether_dhost[5]);

    ethertype= ntohs(eptr->ether_type); /* Converting ethertype from network to host byte order */

    printf("(ethertype: 0x%04x)\n\n", ethertype);
}
```

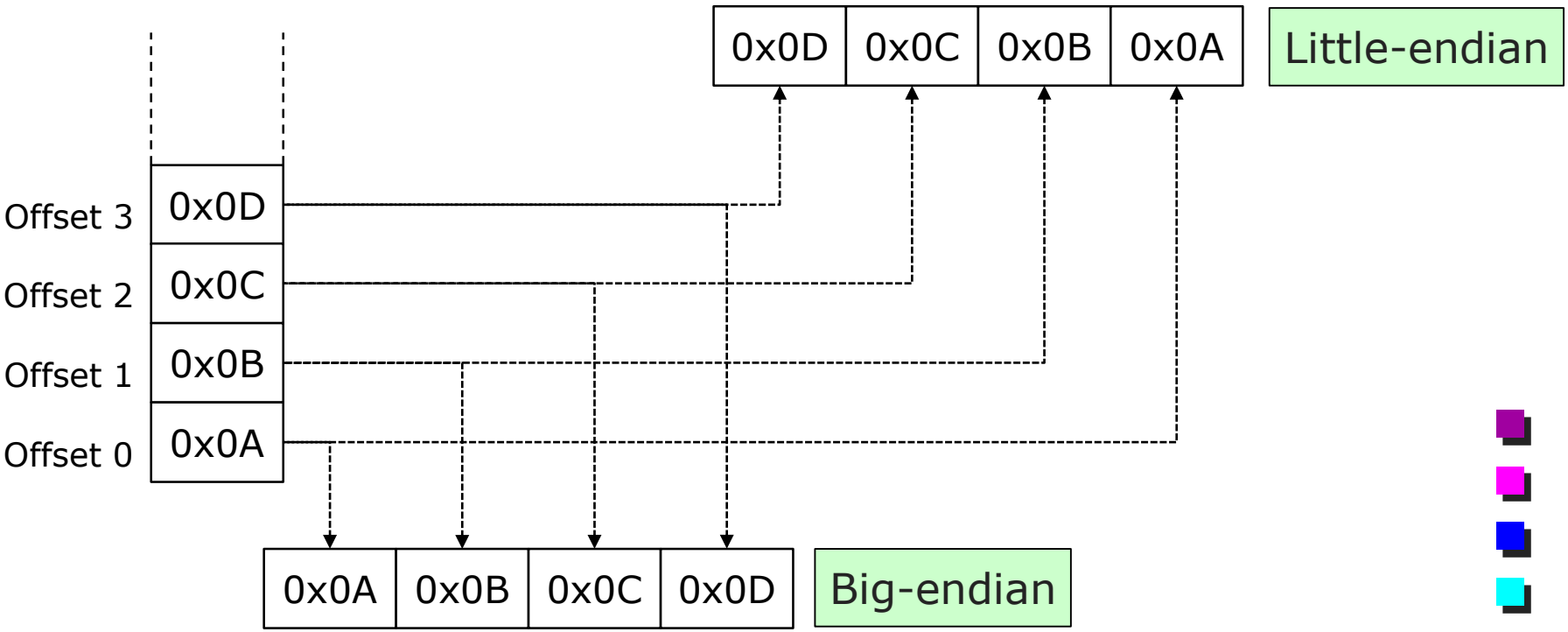
Warning!
In Windows, remember to add the library "ws2_32" in CMakeLists.txt in order to resolve the linking for function ntohs()

Function that converts the ethertype from network byte order to host byte order

Endianess

Need to use ntohX() and htonX() to convert data

Host-based order for Intel CPU



Network byte order (raw packet data)

Parsing packets: a more complex example (1)

```
#ifdef WIN32
  #include <winsock2.h> /* Needed for ntohs() in Windows */
#endif

struct ether_header {
    u_int8_t ether_dhost[6];    /* 6 bytes destination address*/
    u_int8_t ether_shost[6];    /* 6 bytes source address */
    u_int16_t ether_type;       /* 2 bytes ethertype */
    };

struct ip_header {
    u_int8_t  lenver;       /* header length + version*/
    u_int8_t   ip_tos;      /* type of service */
    u_int16_t  ip_len;      /* total length */
    u_int16_t  ip_id;       /* identification */
    u_int16_t  ip_off;      /* fragment offset field */
    u_int8_t   ip_ttl;      /* time to live */
    u_int8_t   ip_p;        /* protocol */
    u_int16_t  ip_sum;      /* checksum */
    u_int8_t   ip_src[4];   /* source and dest address */
    u_int8_t   ip_dst[4];   /* source and dest address */
};

int main(int argc, char **argv) { ... }
```

Warning!
Byte ordering problems are present here as well

This definition makes easy to get access to each single digit, but is not very efficient when we need to compare addresses; in that case, better to transform the IP address in a u_int32_t

Parsing packets: a more complex example (2)

```
void dispatcher_handler(u_char *temp1, const struct pcap_pkthdr *header, const u_char *pkt_data)
{
    struct ether_header *eptr;
    u_int16_t ethertype;

    eptr = (struct ether_header *) pkt_data;

    ethertype= ntohs(eptr->ether_type);

    if (ethertype == 0x800)
    {
        struct ip_header *ipptr;

        ipptr = (struct ip_header *) &pkt_data[14];

        /* Print on screen the IP addresses of each packet */
        printf("%d.%d.%d.%d --> %d.%d.%d.%d (length %d)\n\n",
                ipptr->ip_src[0], ipptr->ip_src[1], ipptr->ip_src[2], ipptr->ip_src[3],
                ipptr->ip_dst[0], ipptr->ip_dst[1], ipptr->ip_dst[2], ipptr->ip_dst[3],
                ntohs(ipptr->ip_len));

    }
    else
        printf("Not an IP packet (ethertype: 0x%04x)\n\n", ethertype);

}
```

Even trivial protocol parsing usually requires several checks on the packets; for instance, this code does not support VLANs, does not check if the IP header contains options, etc.

Although it works (in this case), better to avoid fixed offsets in the code

Parsing packets: check for valid offset

- Correct code always checks for the existence of the given offset in the packet dump before accessing to the data
  - In theory, check that the current offset is less than the packet length
  - In practice, need to check against the *captured length*, as libpcap/WinPcap may capture only a *snapshot* of the packet (hence, originating apparently "truncated" packets)
  - Both variables are part of the `pcap_header` structure

```
void dispatcher_handler(u_char *temp1, const struct pcap_pkthdr *header, const u_char *pkt_data)
{
    /* ... */

    // Check if the destination IP port is 80 and the TCP is followed by a valid payload
    if ( (tcp.dport == htons(80)) && (header->caplen > (etherlen + iplen + tcplen)) )
    {
        /* If so, print the (supposed) HTTP header */
        /* Please remember that the packet may not end with a \0 terminated string, hence this
           function can raise a buffer overflow */
        printf("HTTP header: %s", &pkt_data[etherlen + iplen + tcplen]);
    }
}
```

Filtering traffic

- The setfilter() function allows to filter incoming traffic
- The filtering syntax is common among all the libpcap-based tools (e.g., tcpdump/WinDump)
- Some sample filters
  - ip
  - host 1.2.3.4
  - src host 1.1.1.1 and dst port 80
- Filtering syntax can be found in the tcpdump/WinDump man page:

  `http://www.winpcap.org/windump/docs/manual.htm`

Filtering syntax: examples

Filter	Description
[nothing]	Capture all packets
arp	Capture only ARP packets. Similarly, we can specify ip, ip6, tcp, udp, icmp, etc.
host [ip_address]	Capture all the packets coming from or directed to the given IP host.
src host [ip_adddress]	Capture all the packets coming from the given IP host. Similarly, dst host [ip_address] captures all the packets directed to the given IP host.
tcp and port [port]	Capture all the TCP packets coming from or directed to the given tcp port.
src port [port]	Capture all the TCP or UDP packets coming from the given port.
ether host [MAC_address]	Capture all the TCP packets coming from or directed to the given MAC address. The MAC address must be written as aa:bb:cc:dd:ee:ff.
ether broadcast	Capture all the Ethernet broadcast packets.
Boolean operators: and, or, not	Filtering strings can be combined through Boolean operators, while parenthesis (i.e., "()") can be used to specify precedence.

Documentation

- Full list of (working) samples and a step-by-step tutorial to libpcap/WinPcap programming

  `http://www.winpcap.org`

  - (section “Documentation” – “Manual”)

- A small example (slightly modified from the WinPcap site) available on the course website

Caveats

- Creating packet processing code is easy
- Creating good packet processing code is hard
  - Efficiency matters, when you have a few clock cycles per packet
  - Avoid copying memory
  - Avoid memory allocations
  - Avoid pointers
  - Avoid function calls
  - Consider CPU-level issues (cache alignment, cache and memory latencies, CPU cores bindings, etc)
- Necessity to deal with tons of low-level details
  - Complex protocol encapsulations
  - IP segmentation
  - TCP session handling and payload reassembly

Conclusions

- Libpcap/WinPcap is the de-facto standard library for packet sniffing and raw-packet creation
- Rather easy primitives
- Leaves a lot of code under the responsibility of the programmer
- Customized versions available (through NIC-specific accelerated drivers) if performance is an issue