Ethernet Frame Capturing and Programming with Ethernet

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

Outline

- Ethernet frame capturing
    - Motivation
    - Tools and library
    - Using SciPy and a few related basic tasks
- Programming with Ethernet
    - Introduction to network application
        - Networking communication modes
        - Network application models
        - Programming and experimentation environment
        - Ethernet implementation in practice
    - Berkeley sockets for programming Ethernet
    - Applications/Upper-layer protocols
        - Unicast applications/protocols
        - Broadcast applications/protocols
        - Multicast application/protocols

Ethernet Frame Capturing: Why?

- To understand the design of Ethernet and the services provided by Ethernet,
- To help design upper layer protocols (protocols above Ethernet), and
- To debug network applications and configuration.

Frame Capture Libraries and Tools

- WireShark (See https://www.wireshark.org/)
- Tcpdump and Libpcap (See https://www.tcpdump.org)
- ScaPy (See https://scapy.net/)
- ……

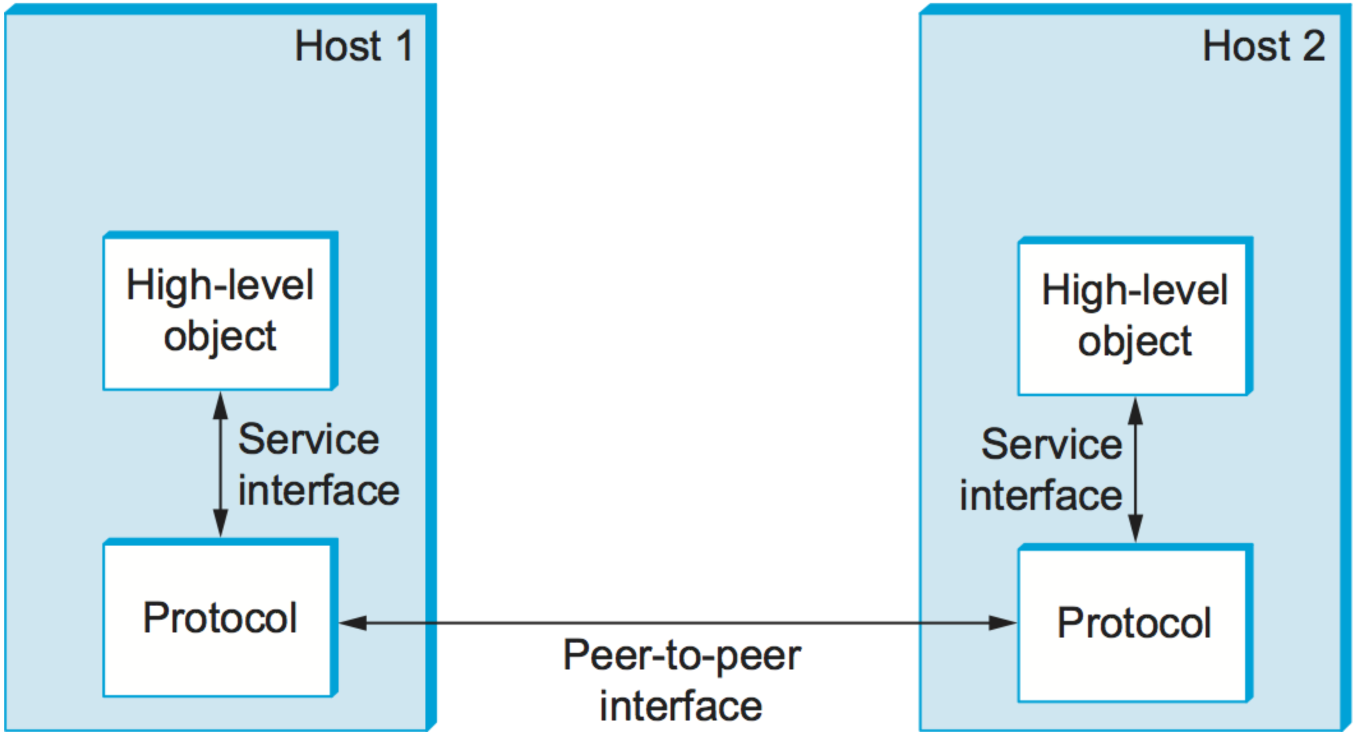
Frame Capturing using ScaPy

- Several tasks
  - Identify Ethernet NICs that a host has
  - Identify Ethernet the hosts are on
  - Sending frames
  - Receiving frames
  - Examining frames

Programming with Ethernet

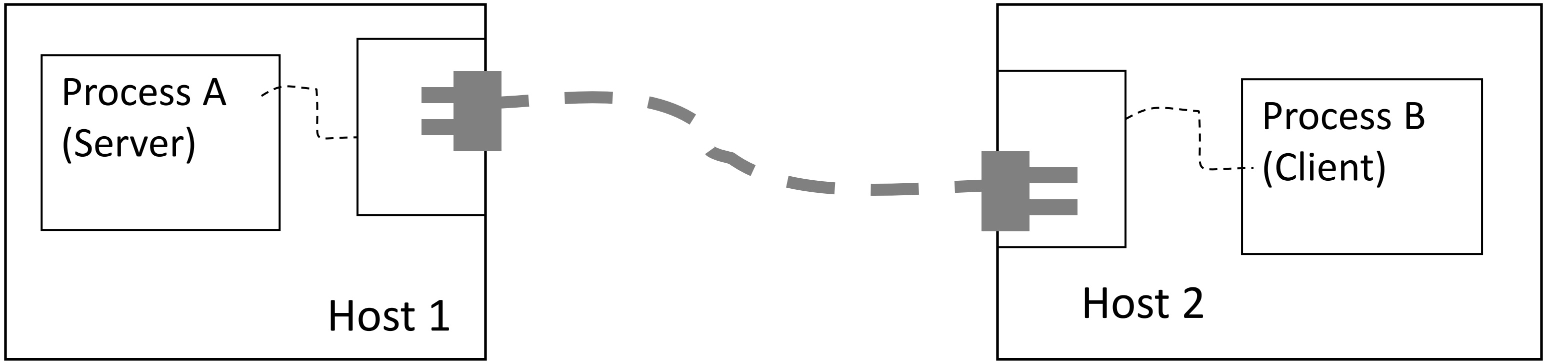
- Introduction to network application
  - Networking communication modes
  - Network application models
  - Programming and experimentation environment
  - Ethernet implementation in practice
- Berkeley sockets for programming Ethernet
- Applications/Upper-layer protocols
  - Unicast applications/protocols
  - Broadcast applications/protocols
  - Multicast application/protocols

Recall Service and Peer-to-Peer Interfaces …

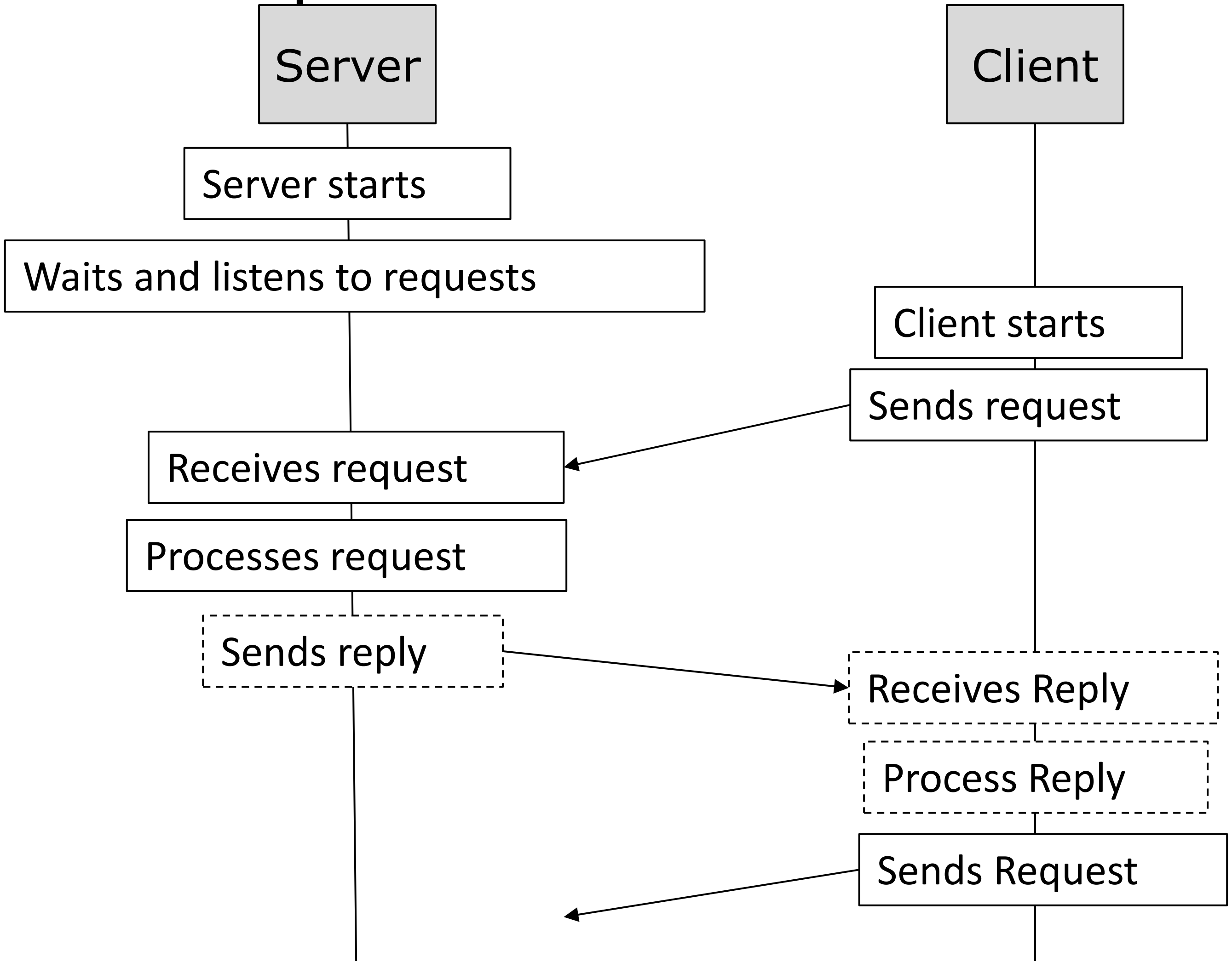


Network Application

- At least two processes (two running programs, typically at two hosts)
- *Server* logic: listening and processing client's requests
- *Client* logic: sending requests to server
- Example setup
  - Process A: server logic
  - Process B: client logic



Server and Client Interaction: An Example



Server

Server starts

Waits and listens to requests

Receives request

Processes request

Sends reply

Client

Client starts

Sends request

Receives Reply

Process Reply

Sends Request

Client-Server and Peer-to-Peer Models

Client-Server Model

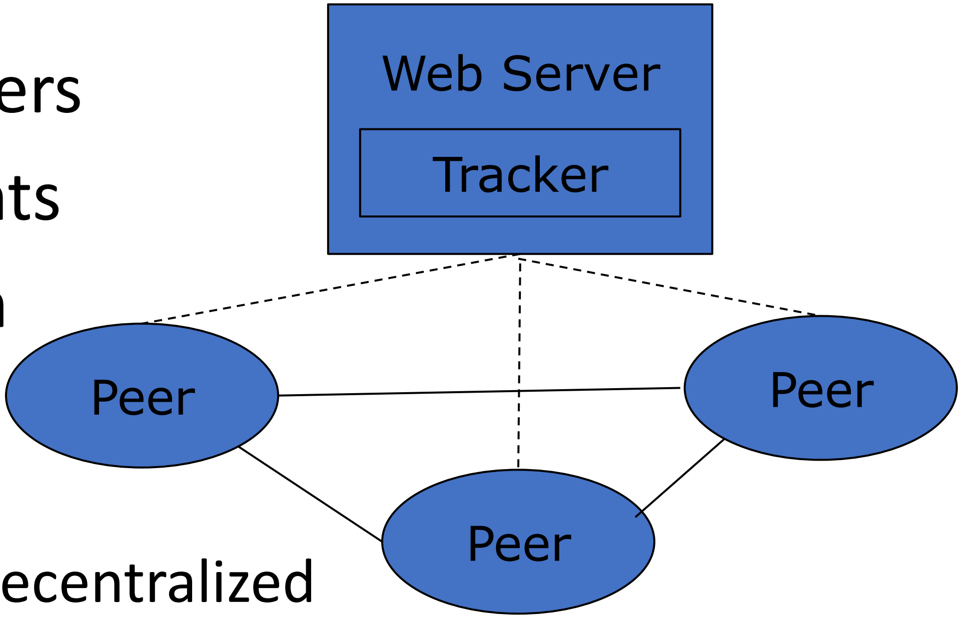
- Server
  - Running server logic
  - Passively waiting: listening to client requests
  - Serving client requests
- Client
  - Running client logic
  - Actively requesting service from server (sending requests)

Peer-to-Peer Model

- Any of the communicating party contains both server and client logics
- Each party listens to and serves requests from other parties
- Each party can initiate requests and send requests

Hybrid Model combines the both models

Hybrid Model Example



- Some hosts act as servers
- Some hosts act as clients
- Some hosts act as both
- Example: BitTorrent
  - Searching: centralized
  - Downloading: largely decentralized
  - Torrent file
    - File name, length, hashes of pieces of the file, URL to a tracker

Connectionless & Connection-Oriented Modes

- Network applications or protocols can follow either one of the two communication modes
- Connectionless communication
    - Does not require to *establish a connection* before transmitting data and to *tear down the connection* after transmitting the data
- Connection-oriented communication
    - Requires to *establish a connection* before transmitting data

Connection-Oriented Mode

- Setting up a connection
  - Determine whether there is a communication path between the two communication parties
  - Reserve network resources
- Transmitting and receiving data
- Tearing down the connection
  - Release resources

Choosing Connected-Oriented or Connectionless Modes

- Consider application requirement and decide which one works best for the application*
    - How reliable must the connection be?
    - Must the data arrive in the same order as it was sent?
    - Must the connection be able to handle duplicate data packets?
    - Must the connection have flow control?
    - Must the connection acknowledge the messages it receives?
    - What kind of service can the application live with?
    - What level of performance is required?
- If reliability is paramount, then connection-oriented transport services (COTS) is the better choice.

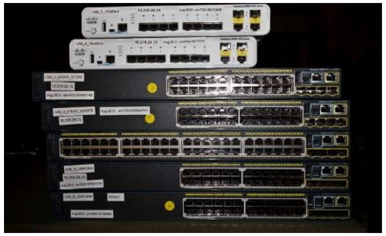
*From *Transport Interfaces Programming Guide, SunSoft, 1995*

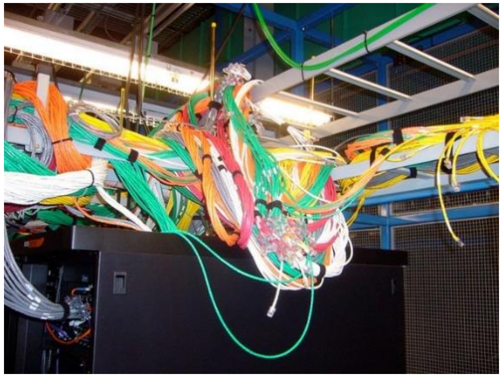
Programming with Ethernet

- Writing programs using functionality provided by ***Ethernet*** adapters and availed by their drivers
- Low-level program for creating network applications
- Useful to create new upper-layer network protocols or application
- Where is ***Ethernet***?

Ethernet: Where is it?

- Infrastructure





Ethernet: Where is it?

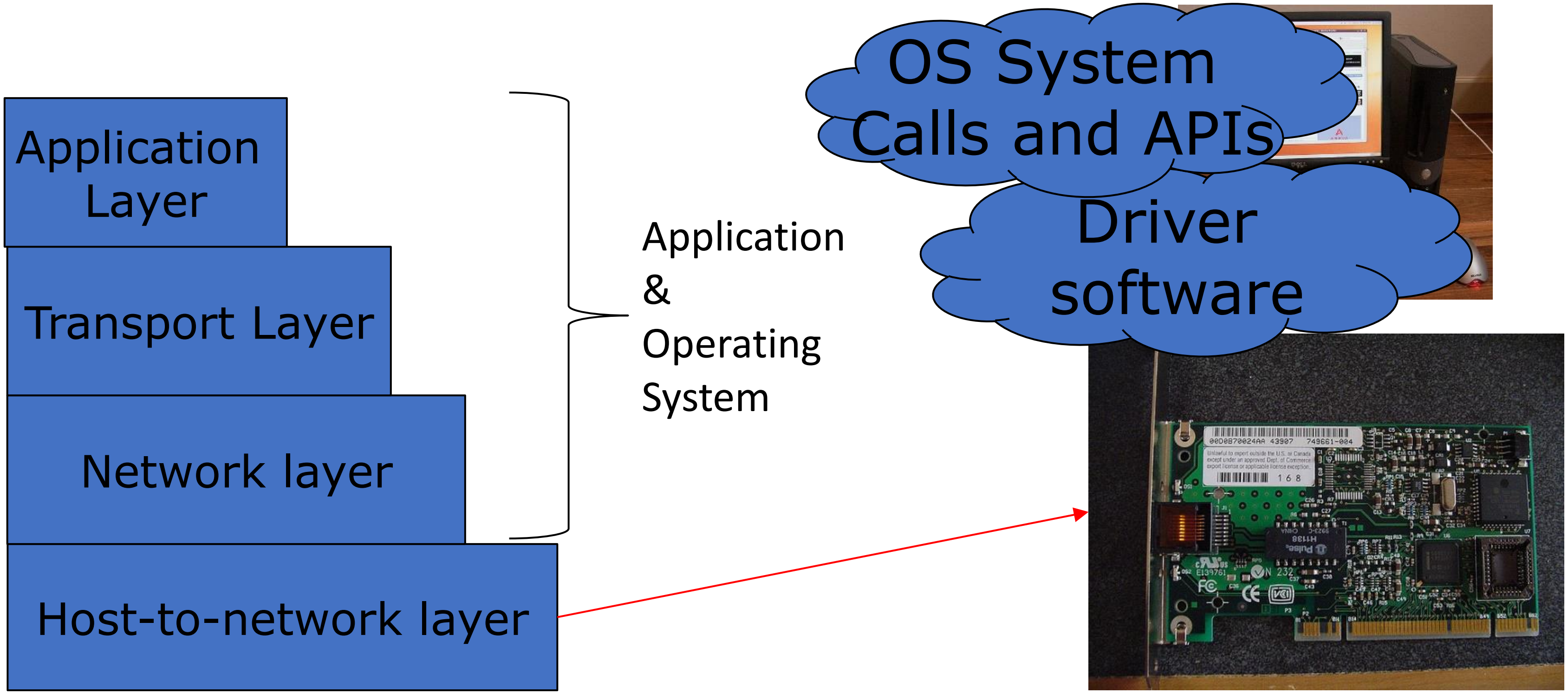
- Ethernet Adapter

Ethernet: Where is it?

- Beside hardware, firmware inside
  - Encoding
  - Error Detection
  - Medium Access Control (CSMA/CD)

Ethernet: Upper Layer Protocol Design and Programming

- How to access functionality of Ethernet adapter?



Application Layer

Transport Layer

Network layer

Host-to-network layer

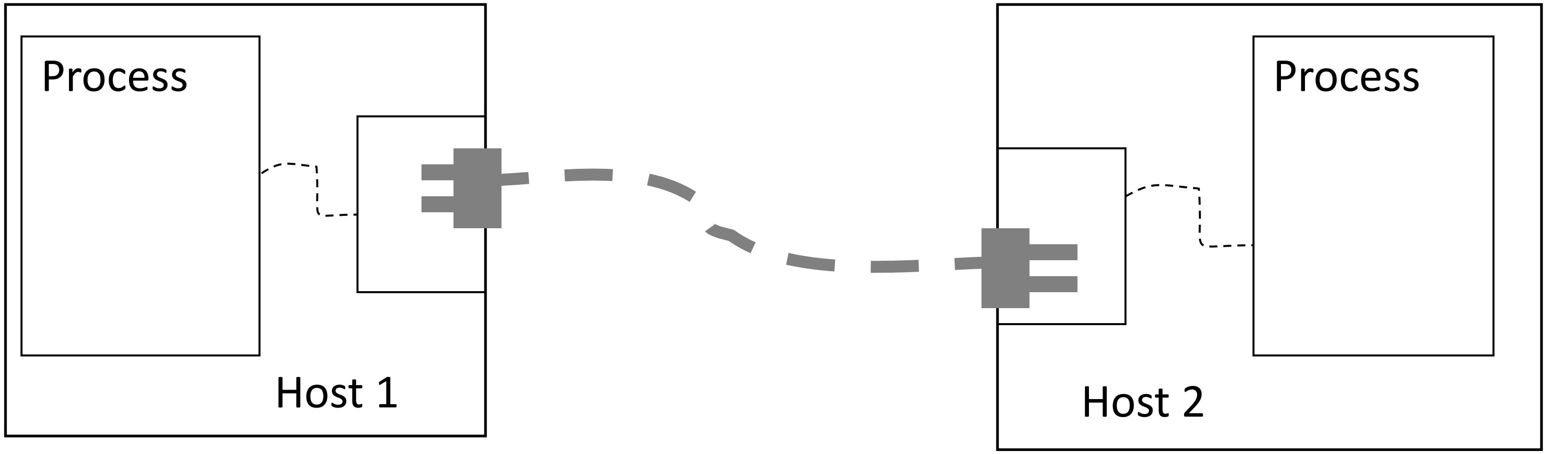
Application & Operating System

OS System Calls and APIs

Driver software

Berkeley Sockets

- Protocol provides a set of interfaces → abstract
- API (application programming interface) → how the interfaces exposed in a particular operating system
- Berkeley socket interfaces
  - APIs to multiple protocols
  - Socket: a “point” where an application process attaches to the network; “end-point” of communication



Programming Ethernet with Socket API

- Learn socket APIs to
  - Create a socket
  - Send messages via the socket
  - Receive message via the socket
- Using C and Linux manual pages
  - Python Socket API is a wrapper
- Example programs using a typical setup
  - Write two programs (A, B)
    - Program A contains and runs the server logic
    - Program B contains and runs the client logic

Creating Socket

```
int socket(int domain, int type, int
protocol)
```

- Creates an endpoint for communication and returns a descriptor.
- Look it up in Linux manual: see socket(2)
  - which means issue command "*man 2 socket*".

Communication Domain

- `int socket(`**`int domain`**`, int type, int protocol)`
- AF_PACKET is our interest: Low level packet interface

  "Packet sockets are used to receive or send raw packets at the device driver (OSI Layer 2) level. They allow the user to implement protocol modules in user space on top of the physical layer."
- More information, see packet(7)

Communication Type

- `int socket(int domain, int type, int protocol)`

- Specify a communication semantics with a communication domain
- For AF_PACKET domain
  - SOCK_RAW: for raw packets (including the link level header)
  - SOCK_DGRAM: for cooked packets (with the link level header removed)

Protocol

- `int socket(int domain, int type,` **`int protocol`**`)`
- Specifies a particular protocol to be used with the socket.
- Protocol is a protocol number in network order
- For AP_PACKET domain
  - Protocol can be the IEEE 802.3 protocol number in network order.
  - linux/if_ether.h lists acceptable protocol numbers for Ethernet (typical location: /usr/include/linux/if_ether.h)

Protocol Number for Ethernet

- linux/if_ether.h lists acceptable protocol numbers for Ethernet
  - typical location: /usr/include/linux/if_ether.h

```
……
#define ETH_P_LOOP  0x0060      /* Ethernet Loopback packet */
#define ETH_P_PUP   0x0200      /* Xerox PUP packet     */
#define ETH_P_PUPAT 0x0201      /* Xerox PUP Addr Trans packet  */
#define ETH_P_IP    0x0800      /* Internet Protocol packet */
……
#define ETH_P_802_3 0x0001      /* Dummy type for 802.3 frames  */
#define ETH_P_AX25  0x0002      /* Dummy protocol id for AX.25  */
#define ETH_P_ALL   0x0003      /* Every packet (be careful!!!) */
        ……
```

Protocol Number

- Which protocol number to use?
- Depending on payload
  - If payload is an IP packet, use ETH_P_IP, i.e., 0x0800
  - If payload is an ARP packet, use ETH_P_ARP, i.e., 0x0806
  - ***If payload is a packet of your own upper layer protocol?***

Protocol Number: Byte Order

- Protocol number must be in network order
- Use functions to convert between host and network order

```
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

- Example
  - htons (0x0800) or htons(ETH_P_IP)

Protocol Number: New Protocol

- What about developing a new protocol?
  - Choose a number not used
    - May run into the problem that other people also choose the same unused number as you
    - Get approval from the IANA
- What about receiving all frames

Protocol Number: All Frames

- What about receiving all frames
- Use protocol number ETHER_P_ALL
- In network order, htons(ETH_P_ALL) or htons(0x0003)

Putting Together: Raw Packet

```
#define MY_PROTOCOL_NUM 0x60001
int sockfd;

……
sockfd = socket(AP_PACKET,
                SOCK_RAW,
                htons(MY_PROTOCOL_NUM));

if (sockfd == -1) {
     /* deal with error */
}
```

Putting Together: Cooked Packet

```
#define MY_PROTOCOL_NUM 0x60001
int sockfd;

……
sockfd = socket(AP_PACKET,
                SOCK_DGRAM,
                htons(MY_PROTOCOL_NUM));

if (sockfd == -1) {
     /* deal with error */
}
```

Putting Together: All Raw Packet

```
int sockfd;
```

……

```
sockfd = socket(AP_PACKET,
                SOCK_RAW,
                htons(ETH_P_ALL));

if (sockfd == -1) {
     /* deal with error */
}
```

Sending Messages

```
ssize_t sendto(int sockfd, const void *buf, size_t
len, int flags, const struct sockaddr *dest_addr,
socklen_t addrlen);
```

```
ssize_t send(int sockfd, const void *buf, size_t
len, int flags);
```

```
ssize_t write(int fd, const void *buf, size_t
count);
```

```
ssize_t sendmsg(int sockfd, const struct msghdr
*msg, int flags);
```

Sending Messages: Manual Pages

- See send(2)
- See sendto(2)
- See sendmsg(2)
- See write(2)

Sending Message: Differences

- Relationship among the system calls
  - write(fd, buf, len);

    is equivalent to

    send(sockfd, buf, len, 0);
  - send(sockfd, buf, len, flags);

    is equivalent to

    sendto(sockfd, buf, len, flags, NULL, 0);
  - write(fd, buf, len);

    is equivalent to

    sendto(sockfd, buf, len, 0, NULL, 0);

Sending Messages: sendto(…)

- ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr, socklen_t addrlen);
  - sockfd: the file descriptor of the sending socket
  - buf: message to send
  - len: message length
  - flags: the bitwise OR of flags or 0
  - dest_addr: the address of the target
  - addrlen: the size of the target address

Message

- Case 1: raw packet

```
sockfd = socket(AP_PACKET, SOCK_RAW,
        htons(MY_PROTOCOL_NUM));
```

  - *buf* contains Ethernet header and data (i.e., payload)
- Case 2: cooked packet

```
sockfd = socket(AP_PACKET, SOCK_DGRAM,
        htons(MY_PROTOCOL_NUM));
```

  - *buf* contains data (i.e, payload)

Destination Address

- struct sockaddr *desk_addr
  - struct sockaddr * is a place holder
  - desk_addr should points to an instance of struct sockaddr_ll

Link Layer Address

- See packet(7)

```
struct sockaddr_ll {
  unsigned short sll_family;   /* Always AF_PACKET */
  unsigned short sll_protocol; /* Physical layer protocol */
  int            sll_ifindex;  /* Interface number */
  unsigned char  sll_pkttype;  /* Packet type */
  unsigned char  sll_halen;    /* Length of address */
  unsigned char  sll_addr[8];  /* Physical layer address */
};
```

Receiving Messages

```
ssize_t recvfrom(int sockfd, void *buf, size_t
len, int flags, struct sockaddr *src_addr,
socklen_t *addrlen);
```

```
ssize_t recv(int sockfd, void *buf, size_t len,
int flags);
```

```
ssize_t write(int fd, const void *buf, size_t
count);
```

```
ssize_t recvmsg(int sockfd, struct msghdr *msg,
int flags);
```

Receiving Message: Manual Pages

- See recv(2)
- See recvfrom(2)
- See recvmsg(2)
- See read(2)

Receiving Message: Differences

- Relationship among the system calls
  - read(fd, buf, len);

    is equivalent to

    recv(sockfd, buf, len, 0);
  - recv(sockfd, buf, len, flags);

    is equivalent to

    recvfrom(sockfd, buf, len, flags, NULL, NULL);
  - read(fd, buf, len);

    is equivalent to

    recvfrom(sockfd, buf, len, 0, NULL, NULL);

Message

- Case 1: raw packet

```
sockfd = socket(AP_PACKET, SOCK_RAW,
           htons(MY_PROTOCOL_NUM));
```

  - *buf* contains Ethernet header and data (i.e., payload)
- Case 2: cooked packet

```
sockfd = socket(AP_PACKET, SOCK_DGRAM,
          htons(MY_PROTOCOL_NUM));
```

  - *buf* contains data (i.e., payload)

Socket Option

- Packet sockets can be used to configure *physical layer multicasting* and *promiscuous mode*.
- Get socket option
  - int getsockopt(int sockfd, int level, int optname, void *optval, socklen_t *optlen);
- Set socket option
  - int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);

Socket Option: Promiscuous Mode

- See packet(7) for PACKET_MR_PROMISC and PACKET_ADD_MEMBERSHIP
- See setsockopt(2) and getsockopt(2)

Putting Together

- Learn from two examples
  - Write two programs (A, B) using client & server model
  - Program A contains the server logic
  - Program B contains the client logic
  - Use ***Ethernet***
- Sample programs (Explore more on your own)
- Two pairs of C programs
  - etherinj and ethercap
  - ethersend and etherrecv
- Two pairs of Python programs

Recall: Types of Ethernet Addresses

- Unicast, multicast, and broadcast
- Creating broadcast and multicast Ethernet applications?

Experiment Environment

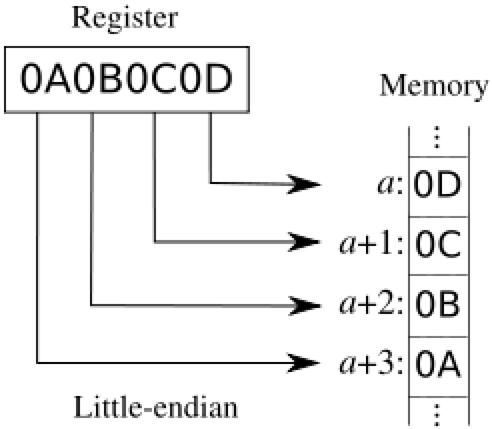
- Use multiple Linux virtual machines
- Recommend Oracle Virtual Box
    - Free for Mac OS X, Windows, and Linux
    - Support various networking setups
- See class website for additional information

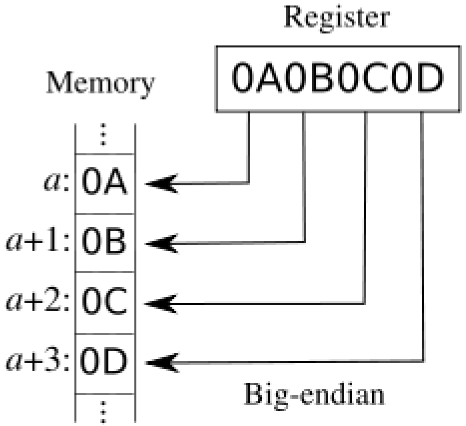
Summary

- Client-Server and Peer-to-Peer models
- Connection-oriented and Connectionless communication modes
- Programming Ethernet with Socket APIs
- Byte order and network order
  - If you forgot byte order, continue to study the rest of the slides
  - Need to know: hton* and ntoh* APIs

Byte Order: Big Endian and Little Endian

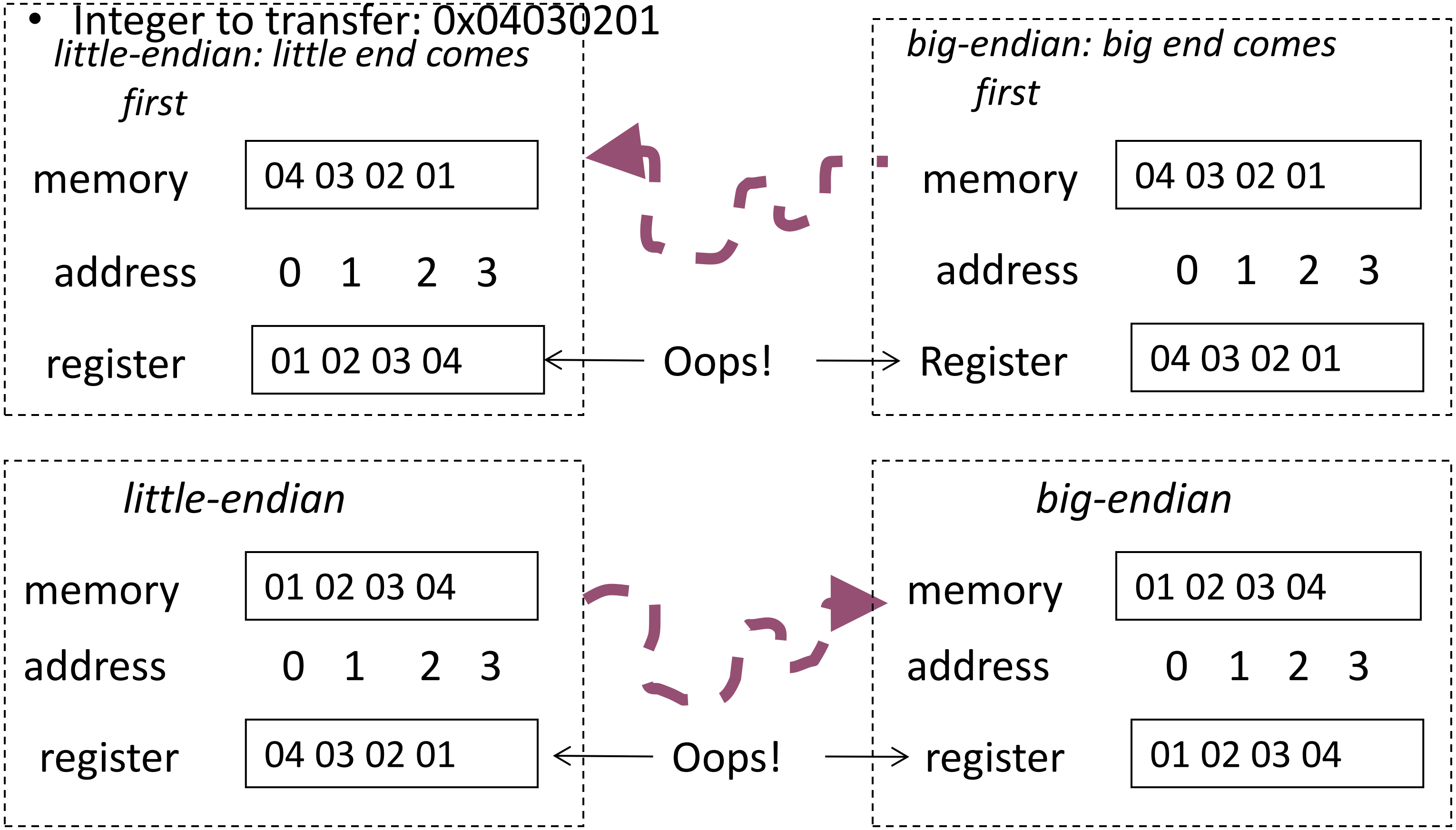
- Little Endian
  - Low-order byte of a word is stored in memory at the lowest address, and the high-order byte at the highest address → The little end comes first
- Big Endian
  - high-order byte of a word is stored in memory at the lowest address, and the low-order byte at the highest address → The big end comes first





Endian-ness: Transfer Integer over Network

- Integer to transfer: 0x04030201



little-endian: little end comes first

memory 04 03 02 01

address 0 1 2 3

register 01 02 03 04

big-endian: big end comes first

memory 04 03 02 01

address 0 1 2 3

Register 04 03 02 01

Oops!

little-endian

memory 01 02 03 04

address 0 1 2 3

register 04 03 02 01

big-endian

memory 01 02 03 04

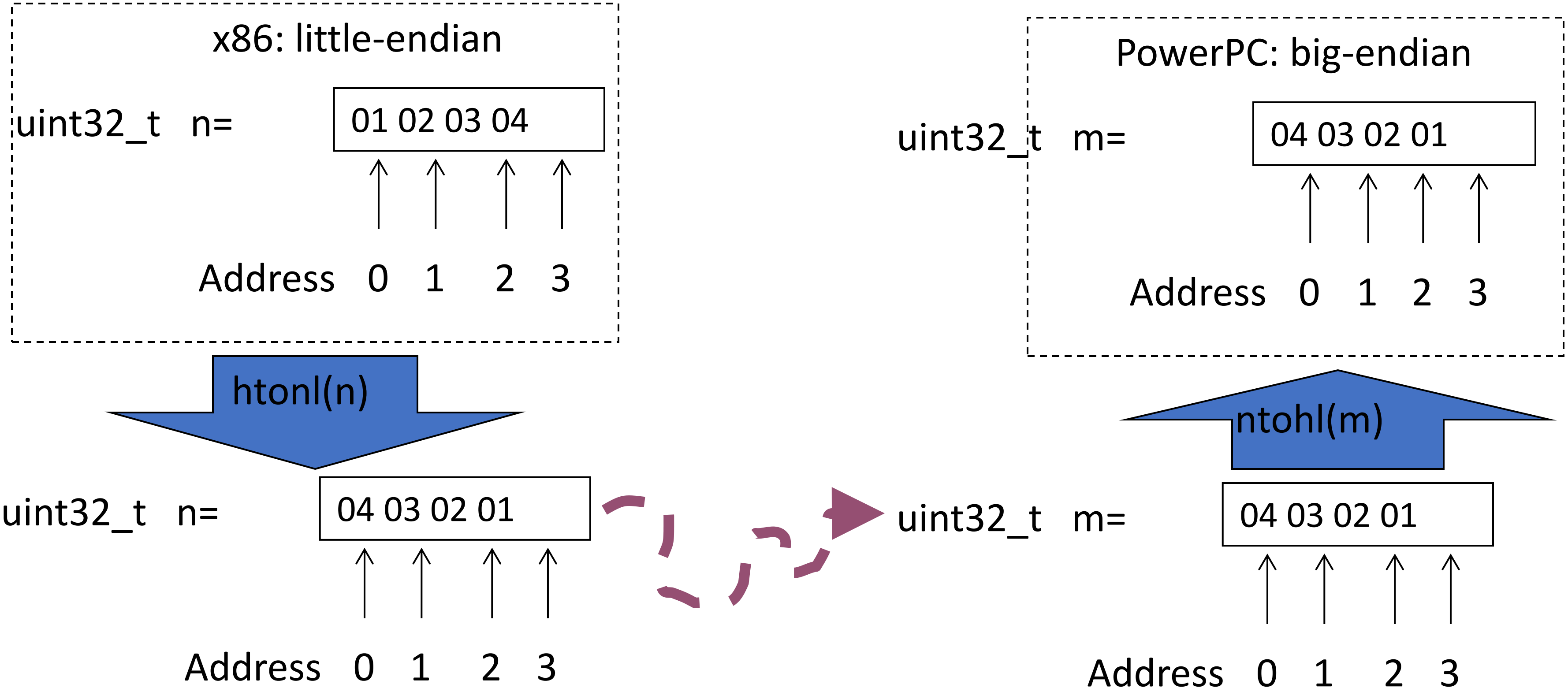
address 0 1 2 3

register 01 02 03 04

Oops!

Network Order

- Integer to transfer: 0x04030201



x86: little-endian

uint32_t n= 01 02 03 04

Address 0 1 2 3

htonl(n)

uint32_t n= 04 03 02 01

Address 0 1 2 3

PowerPC: big-endian

uint32_t m= 04 03 02 01

Address 0 1 2 3

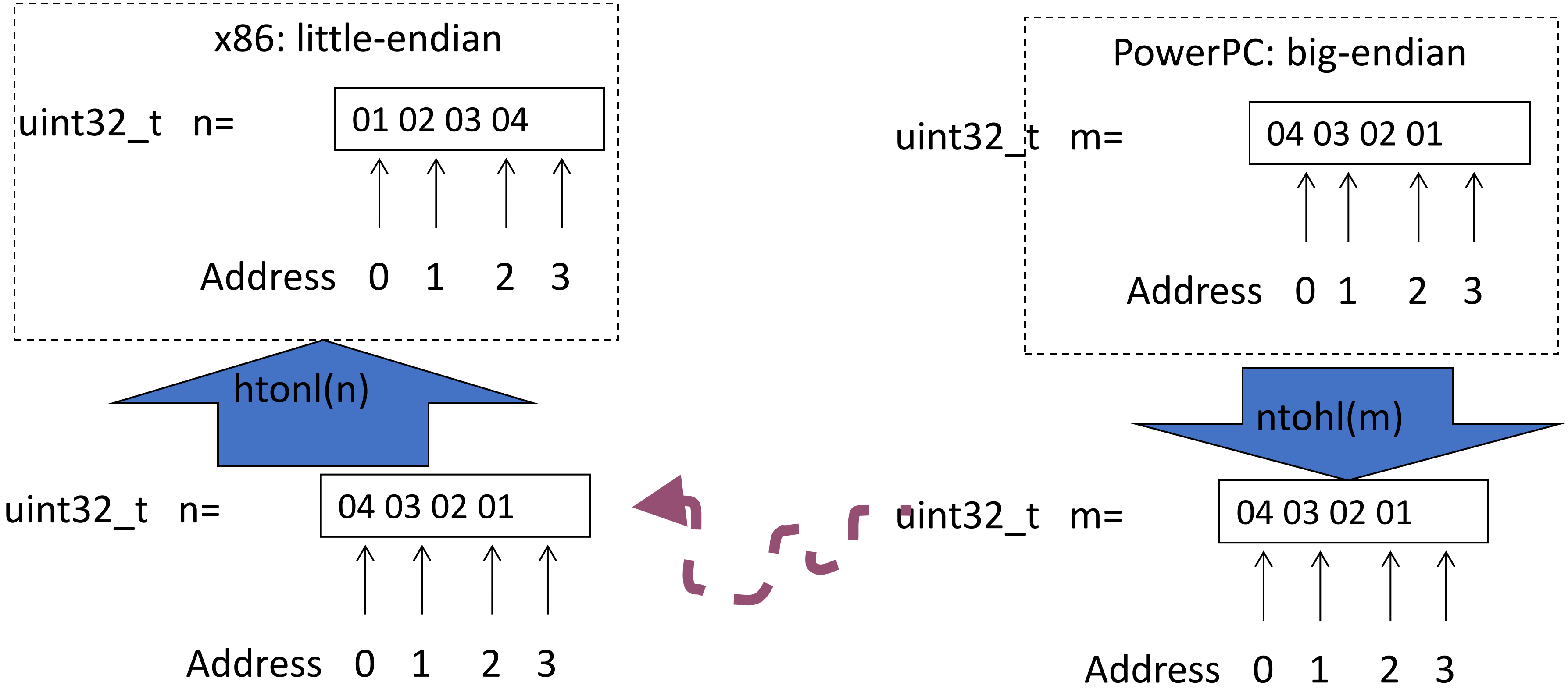
ntohl(m)

uint32_t m= 04 03 02 01

Address 0 1 2 3

Network Order

- Integer to transfer: 0x04030201



x86: little-endian

uint32_t n= 01 02 03 04

Address 0 1 2 3

htonl(n)

uint32_t n= 04 03 02 01

Address 0 1 2 3

PowerPC: big-endian

uint32_t m= 04 03 02 01

Address 0 1 2 3

ntohl(m)

uint32_t m= 04 03 02 01

Address 0 1 2 3