Socket Programming

❑ What is a socket?

❑ Using sockets
  ○ Types (Protocols)
  ○ Associated functions
  ○ Styles
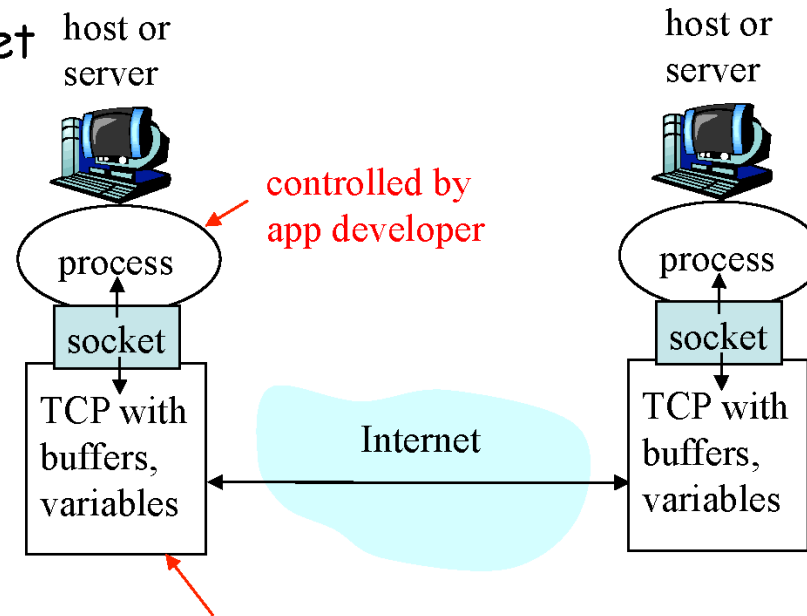
  ○ Socket programming reference:
    • TCP/IP 소켓 프로그래밍 – C버전, Michael J. Donahoo, (박준철 번역), 사이텍미디어

What is a socket?

UPP 선택
TCP 소켓

- ❑ **An interface between application and network**
  - ○ The application creates a socket
  - ○ The socket *type* dictates the style of communication
    - • reliable vs. best effort
    - • connection-oriented vs. connectionless
- ❑ **Once configured, the application can**
  - ○ pass data to the socket for network transmission
  - ○ receive data from the socket (transmitted through the network by some other host)

host or server

controlled by app developer

process

socket

TCP with buffers, variables

Internet

host or server

process

socket

TCP with buffers, variables
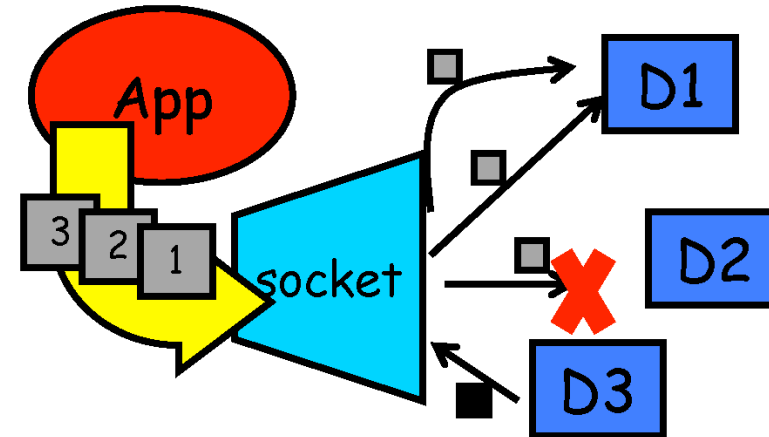
Two essential types of sockets

□ SOCK_STREAM
  ○ a.k.a. TCP
  ○ reliable delivery
  ○ in-order guaranteed
  ○ connection-oriented
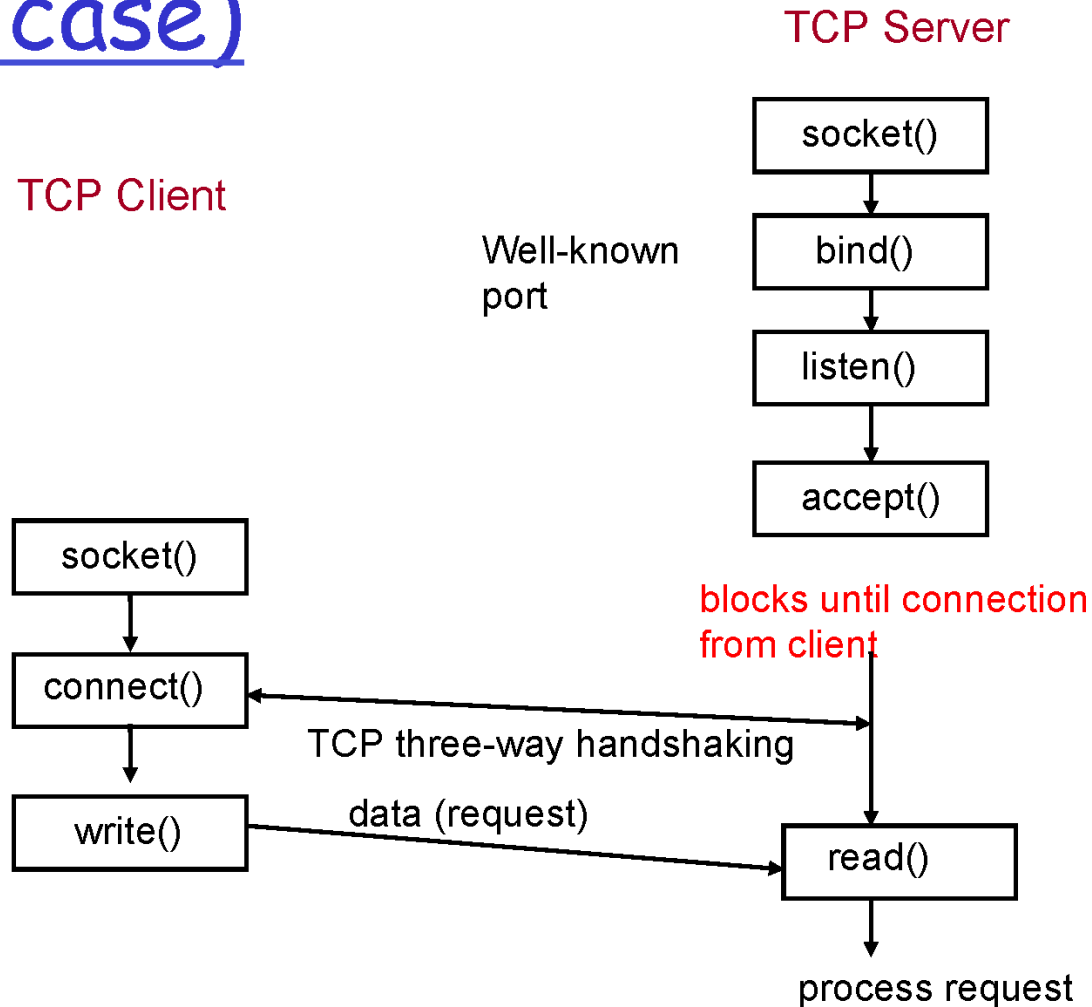 ○ bidirectional
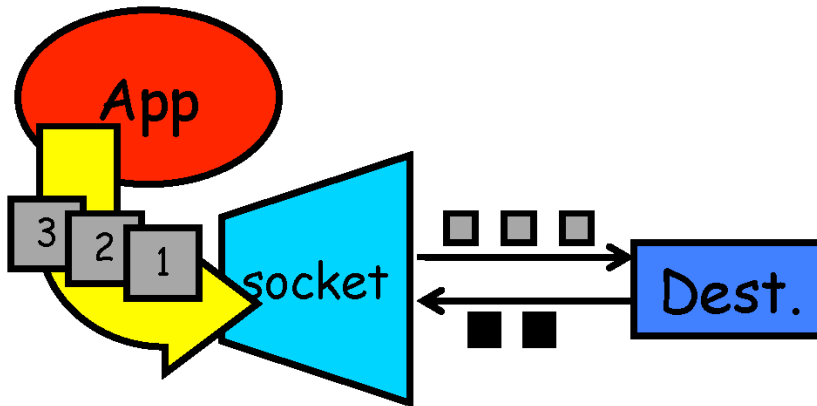
□ SOCK_DGRAM
  ○ a.k.a. UDP
  ○ unreliable delivery
  ○ no order guarantees
  ○ no notion of "connection" – app indicates dest. for each packet
  ○ can send or receive

Sockets API

❐ Creation and Setup
❐ Establishing a Connection (<u>TCP</u>)
❐ Sending and Receiving Data
❐ Tearing Down a Connection (<u>TCP</u>)

4

Big picture: Socket Functions (TCP case)

TCP Server

TCP Client

Well-known port

API

특정 포트에 바인드

클라이언트로 부터 받을 준비 됨!

blocking.

제녁선 들어올 때 까지

```
socket()
   |
bind()
   |
listen()
   |
accept()
```

blocks until connection from client

```
socket()
   |
connect()
   |
write()
```

TCP three-way handshaking

data (request)

```
read()
   |
process request
```

5

Big picture: Socket Functions (TCP case) cont.

TCP Client

```
write()  ──data (request)──►  read()
  │                             │
  │                             ▼
  │                        process request
  │                             │
  │                             ▼
  ▼       ◄──data (reply)──   write()
read()                          │
  │                             ▼
  ▼                           read()
close() ───────────────────►    │
                                ▼
                             close()
```

6
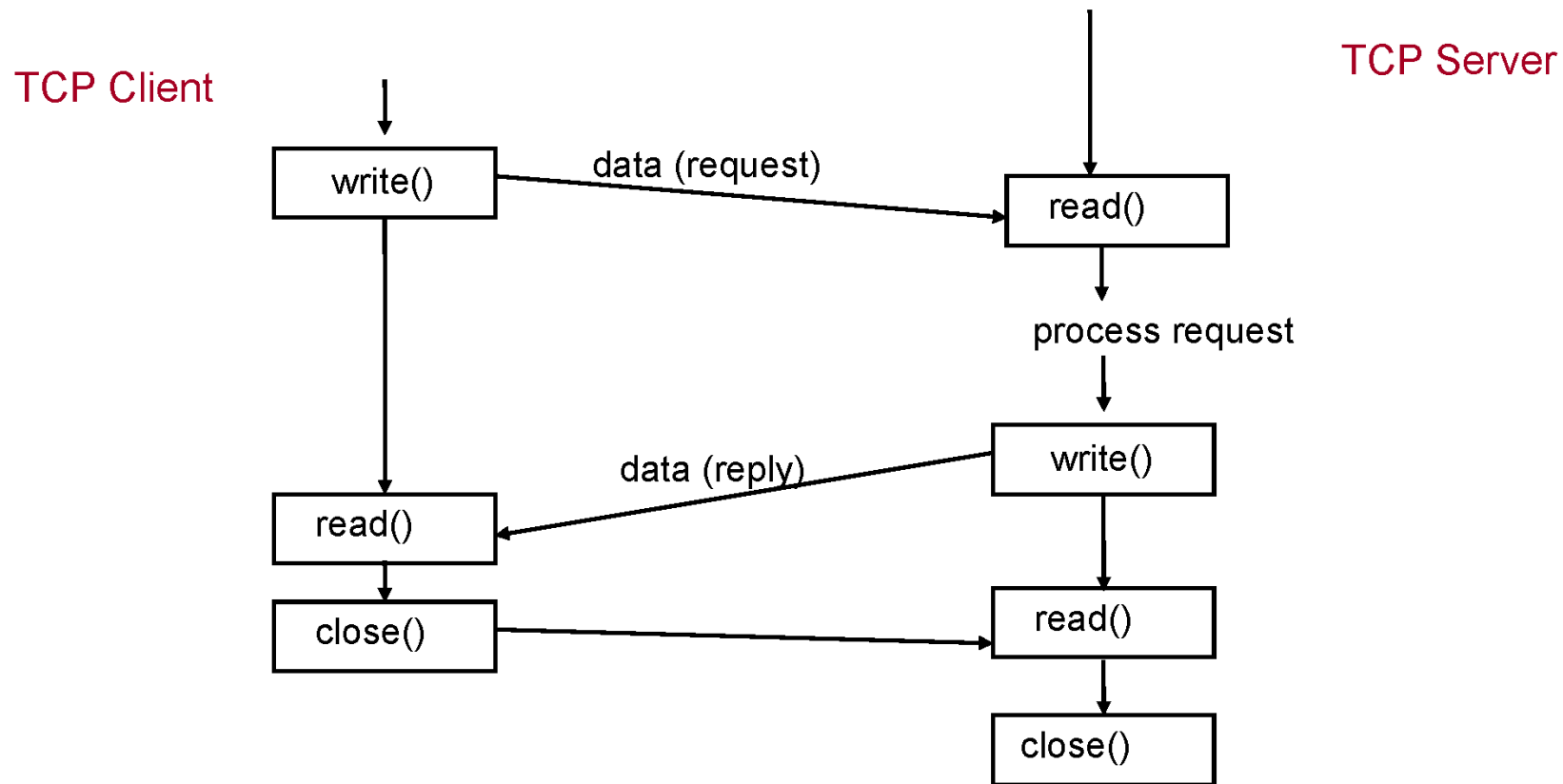
Socket Creation and Setup

❑ Include file `<sys/socket.h>`

❑ Create a socket
  – `int socket (int domain, int type, int protocol);`
  ○ Returns file descriptor or -1.

❑ Bind a socket to a local IP address and port number
  – `int bind (int sockfd, struct sockaddr* myaddr, int addrlen);`

❑ Put socket into passive state (wait for connections rather than initiate a connection).
  – `int listen (int sockfd, int backlog);`

❑ Accept connections
  – `int accept (int sockfd, struct sockaddr* cliaddr, int* addrlen);`
  ○ Returns file descriptor or -1.

8

Function: socket

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

UDP, TCP 멀저 결정

□ **Create a socket.**
  ○ Returns file descriptor or -1. Also sets `errno` on failure.   선택 포x값 return
  – <u>`domain`</u>: protocol family (same as address family)
    • `PF_INET` for IPv4 (typicall used)
    • other possibilities: `PF_INET6` (IPv6), `PF_UNIX` or `PF_LOCAL` (Unix socket), `PF_ROUTE` (routing)
  – <u>`type`</u>: style of communication
    • `SOCK_STREAM` for TCP (with `PF_INET`)
    • `SOCK_DGRAM` for UDP (with `PF_INET`)
  – <u>`protocol`</u>: protocol within family
    • Typically set to 0
    • getprotobyname(), /etc/protocols for list of protocols

9

Function: bind

소켓 끼 활용 특정 address 에 (port)

bind

```
int bind (int sockfd, struct sockaddr*
    myaddr, int addrlen);
```

❐ Bind a socket to a local IP address and port number.
  ○ Returns 0 on success, -1 and sets `errno` on failure.
  – `sockfd`: socket file descriptor (returned from `socket`)
  – `myaddr`: includes IP address and port number
    • IP address: set by kernel if value passed is `INADDR_ANY`, else set by caller
    • port number: set by kernel if value passed is 0, else set by caller
  – `addrlen`: length of address structure
    • `= sizeof (struct sockaddr_in)`

10

Function: listen

```
int listen (int sockfd, int backlog);
```

□ Put socket into passive state (wait for connections rather than initiate a connection).

  ○ Returns 0 on success, -1 and sets `errno` on failure.

  – `sockfd`: socket file descriptor (returned from socket)

  – `backlog`: bound on length of unaccepted connection queue (connection backlog); kernel will cap, thus better to set high

  – `Listen is non-blocking`: returns immediately

Function: accept

```
int accept (int sockfd, struct sockaddr*
    cliaddr, int* addrlen);
```

❏ **Accept a new connection.**
  ○ Returns file descriptor or -1. Also sets `errno` on failure.
  – `sockfd`: socket file descriptor (returned from `socket`)
  – `cliaddr`: IP address and port number of client (returned from call)
  – `addrlen`: length of address structure = pointer to `int` set to `sizeof (struct sockaddr_in)`

❏ **Accept is <u>blocking</u>**
  ○ Waits for connection before returning

12

Sockets API

❑ Creation and Setup

❑ Establishing a Connection (TCP)

❑ Sending and Receiving Data

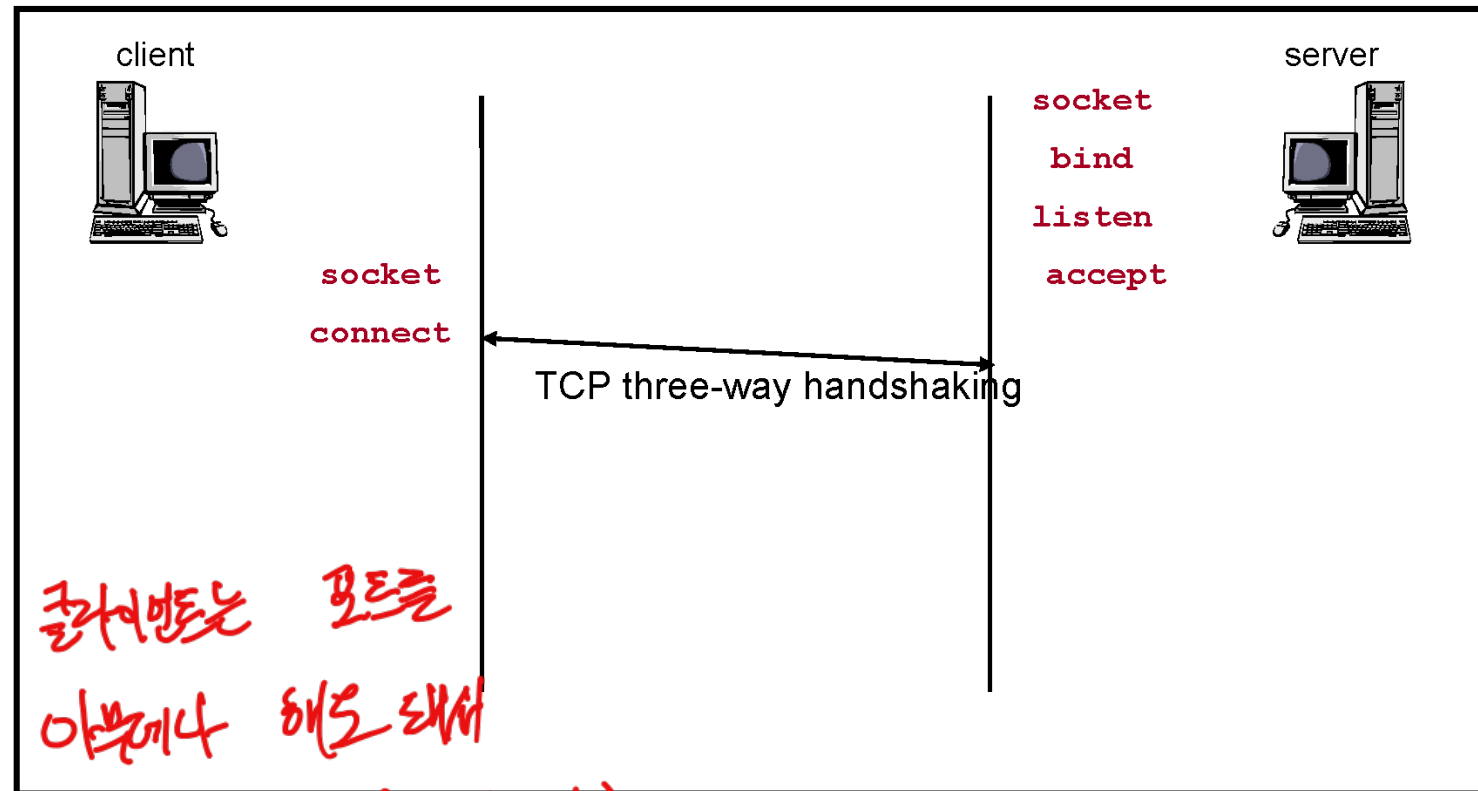❑ Tearing Down a Connection (TCP)

13

Function: connect

```
int connect (int sockfd, struct sockaddr*
  servaddr, int addrlen);
```

☐ Connect to another socket.
  ○ Returns 0 on success, -1 and sets `errno` on failure.
  – `sockfd`: socket file descriptor (returned from `socket`)
  – `servaddr`: IP address and port number of **server**
  – `addrlen`: length of address structure
    • = `sizeof (struct sockaddr_in)`

  – Connect is **blocking**

Recap: TCP socket connection setup

client   server

                                                            socket

                                                            bind

                                                            listen

                socket   **accept**

                connect

                              TCP three-way handshaking

클라이언트는 포트를

아무데나 해도 돼서

bind x (but 해도됨)

Sample code: server

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>
#define PORT 3490
#define BACKLOG 10          /* how many pending
                               connections queue
                               will hold */
```

최대 갯수

server

```
main()
{
    int sockfd, new_fd;       /* listen on sock_fd, new
                                 connection on new_fd */
    struct sockaddr_in my_addr;    /* my address */
    struct sockaddr_in their_addr; /* connector addr */
    int sin_size;

    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0))==-1){
        perror("socket");
        exit(1);
    }
```

17

server

❐ The Internet-specific:
  struct sockaddr_in {
      short sin_family;
      u_short sin_port;
      struct in_addr sin_addr;
      };
  ❍ sin_family = AF_INET
  ❍ sin_port: port # (0-65535)
  ❍ sin_addr: IP-address

```
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(MYPORT); /* short, network
                                     byte order */
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
/* INADDR_ANY allows clients to connect to any one of
the host's IP address */



if (bind(sockfd, (struct sockaddr *)&my_addr,
        sizeof(struct sockaddr)) == -1) {
    perror("bind");
    exit(1);
}
```

18

18

server

```c
if (listen(sockfd, BACKLOG) == -1) {
    perror("listen");
    exit(1);
}
while(1) {  /* main accept() loop */
    sin_size = sizeof(struct sockaddr_in);
    if ((new_fd = accept(sockfd, (struct sockaddr*)
                    &their_addr,&sin_size)) == -1) {
        perror("accept");
        continue;
    }
    printf("server: got connection from %s\n",
                    inet_ntoa(their_addr.sin_addr));
}
```

19

client

```c
if ((sockfd = socket (PF_INET, SOCK_STREAM, 0)) == -1) {
    perror ("socket");
    exit (1);
}


their_addr.sin_family = AF_INET;
their_addr.sin_port = htons (Server_Portnumber);
their_addr.sin_addr = htonl(Server_IP_address);


if (connect (sockfd, (struct sockaddr*)&their_addr,
             sizeof (struct sockaddr)) == -1) {
    perror ("connect");
    exit (1);
}
```

Sockets API

❐ Creation and Setup

❐ Establishing a Connection (TCP)

❐ Sending and Receiving Data

❐ Tearing Down a Connection (TCP)

21

Functions: write

```
int write (int sockfd, char* buf, size_t
  nbytes);
```

❒ Write data to a stream (TCP).
   ○ Returns number of bytes written or -1. Also sets `errno` on failure.
   – `sockfd`: socket file descriptor (returned from `socket`)
   – `buf`: data buffer
   – `nbytes`: number of bytes to try to write

❒ write is **blocking**; returns only after data is sent

22

Functions: read

```
int read (int sockfd, char* buf, size_t
    nbytes);
```

☐ Read data from a stream (TCP).
- ○ Returns number of bytes read or -1. Also sets `errno` on failure.
- ○ Returns 0 if socket closed.
- − `sockfd`: socket file descriptor (returned from `socket`)
- − `buf`: data buffer
- − `nbytes`: number of bytes to try to read
- − read is **blocking**; returns only after data is received

23

Big picture: UDP Socket Functions
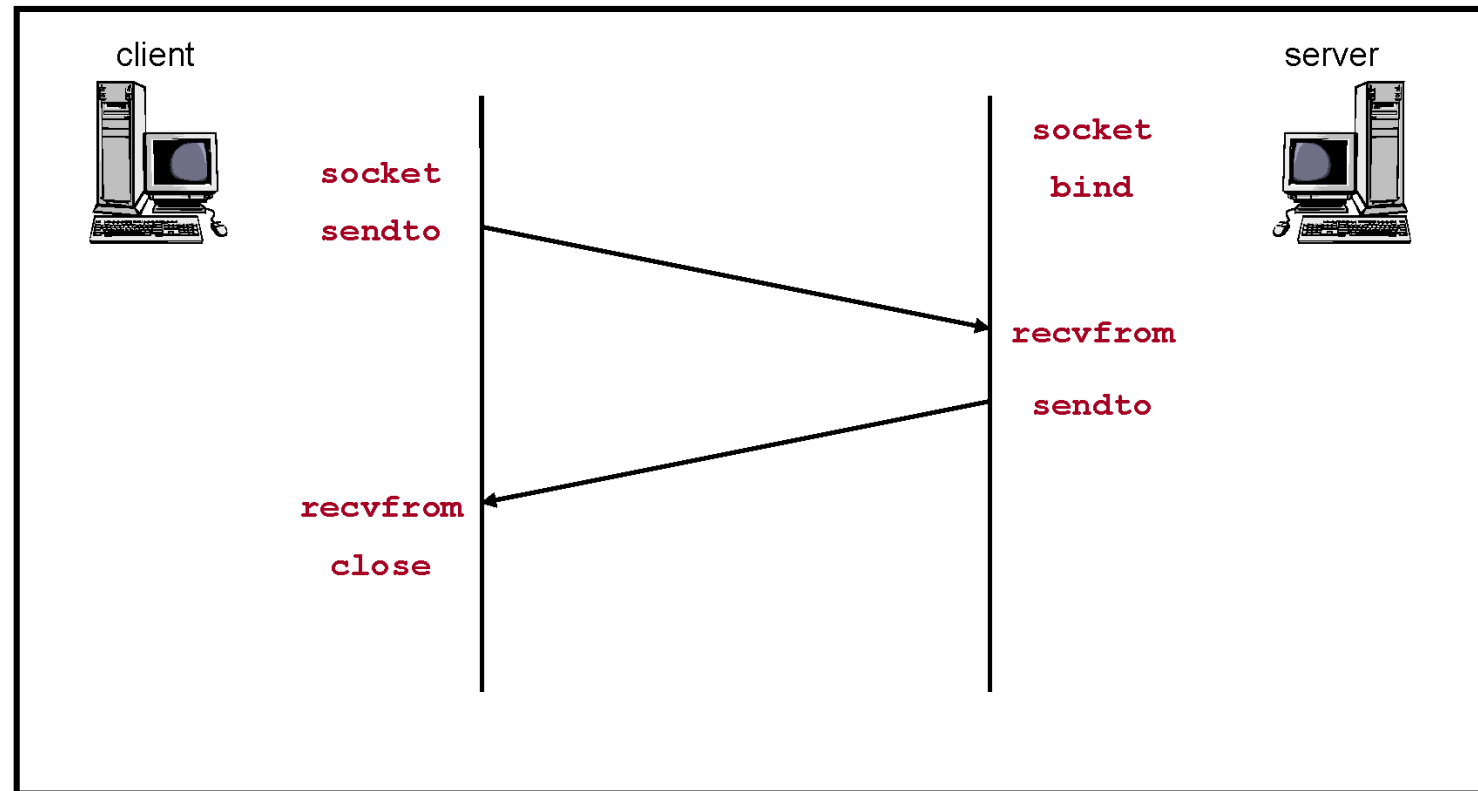
client

server

socket

socket

sendto

bind

recvfrom

sendto

recvfrom

close

Sockets API

❐ Creation and Setup

❐ Establishing a Connection (TCP)

❐ Sending and Receiving Data

❐ Tearing Down a Connection (TCP)

Function: close

```
int close (int sockfd);
```

❒ When finished using a socket, the socket should be closed:

- ❍ returns 0 if successful, -1 if error
- ❍ sockfd: the file descriptor (socket being closed)

❒ Closing a socket

- ❍ frees up the port used by the socket
- ❍ closes a connection (for SOCK_STREAM)

Tip: Release of ports

❐ Sometimes, a "rough" exit from a program (e.g., ctrl-c) does not properly free up a port
❐ Eventually (after a few minutes), the port will be freed
❐ To reduce the likelihood of this problem, include the following code:

```
#include <signal.h>
void cleanExit(){exit(0);}
```
○ in socket code:
```
signal(SIGTERM, cleanExit);
signal(SIGINT, cleanExit);
```

Chapter 3 outline

Multiplexing/demultiplexing
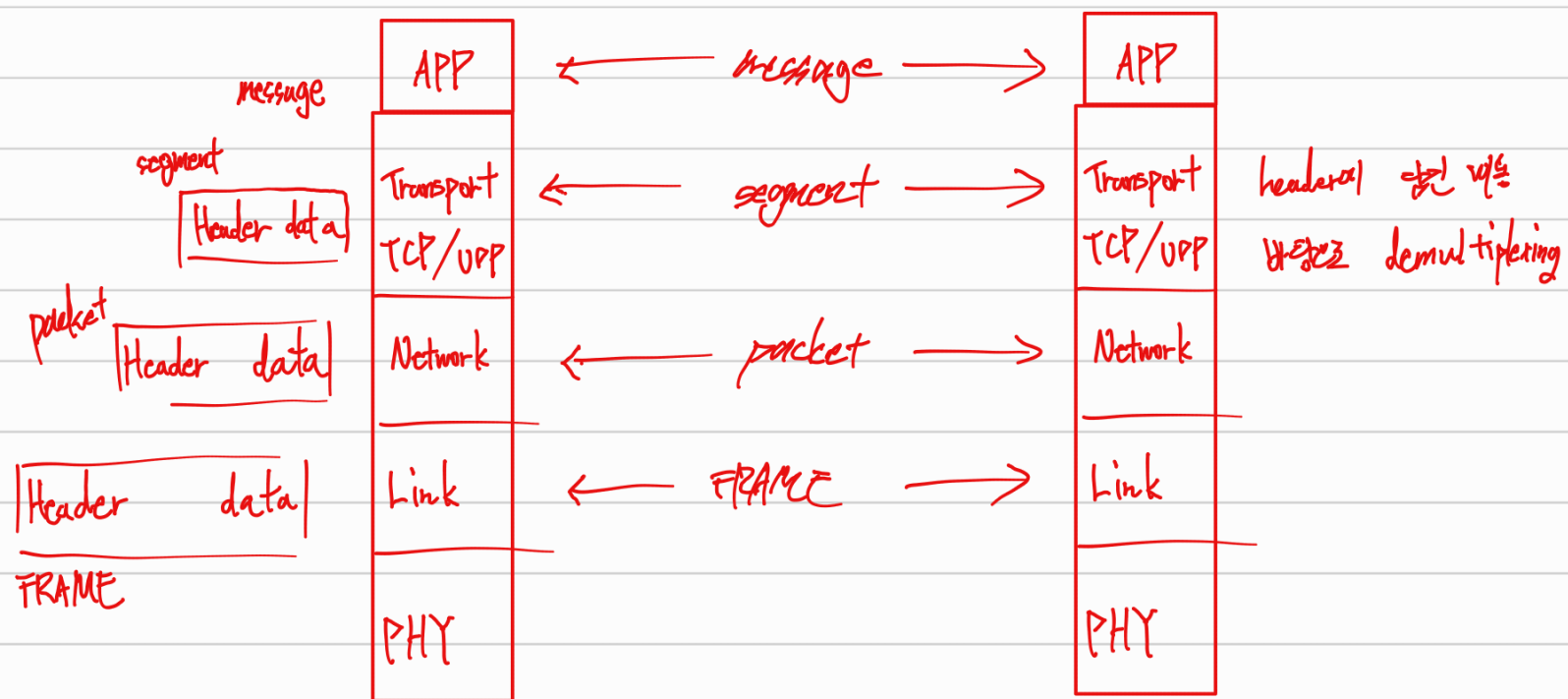
delivering received segments to correct socket

gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

▮ = socket   ⬭ = process

host 1   host 2   host 3

message

segment

Header	data

packet

Header	data

Header	data

FRAME

APP
Transport TCP/UDP
Network
Link
PHY

← message →

← segment →

← packet →

← FRAME →

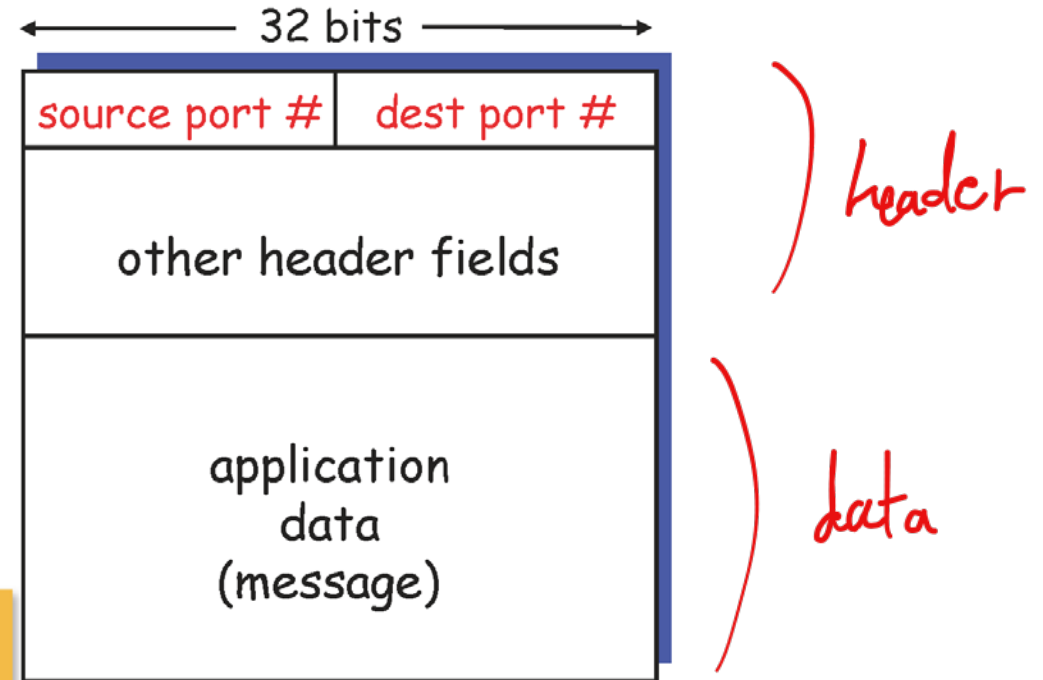| APP |
| --- |
| Transport TCP/UDP |
| Network |
| Link |
| PHY |

header의 정보 바탕
바탕으로 demultiplexing
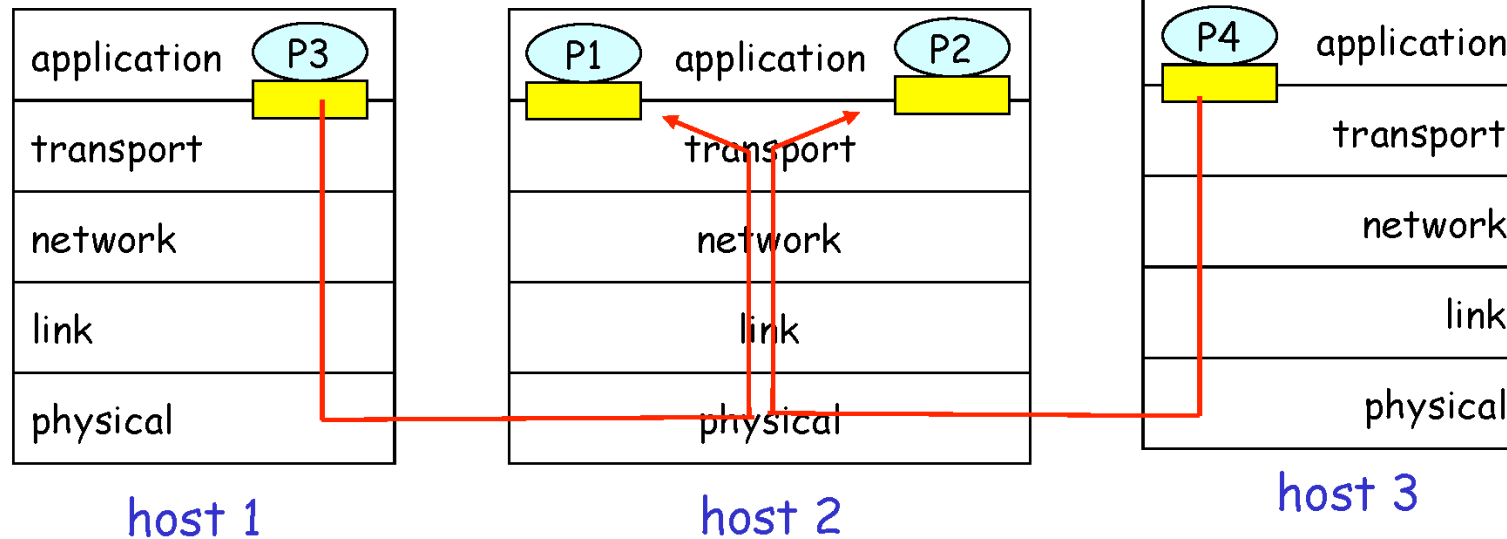
UDP/TCP 마다 *header의 field 다름

How demultiplexing works

❏ host receives IP datagrams
  ○ each datagram has source IP address, destination IP address
  ○ each datagram carries 1 transport-layer segment
  ○ each segment has source, destination port number
❏ host uses IP addresses & port numbers to direct segment to appropriate socket
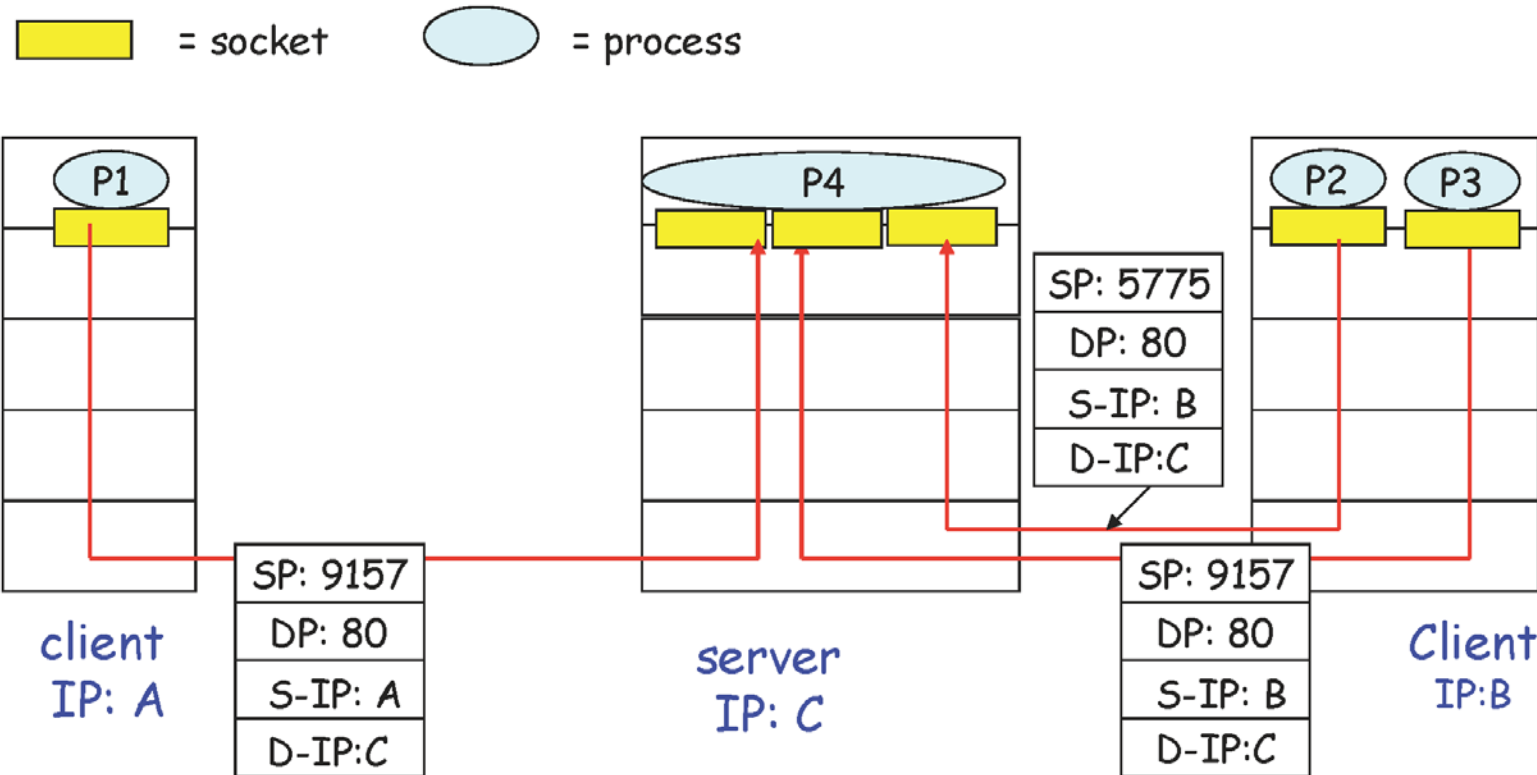
Analogous to airport shuttles

Shuttles MUX passengers and take them to downtown -- DeMUX at different locations

← 32 bits →

source port #	dest port #

other header fields

} header

application
data
(message)

} data

TCP/UDP segment format

Connection-oriented demux: Threaded Web Server

▯ = socket   ◯ = process

P1

P4

P2   P3

| SP: 5775 |
| DP: 80 |
| S-IP: B |
| D-IP:C |

| SP: 9157 |
| DP: 80 |
| S-IP: A |
| D-IP:C |

| SP: 9157 |
| DP: 80 |
| S-IP: B |
| D-IP:C |

client
IP: A

server
IP: C

Client
IP:B

Modify the airport shuttle analogy
to distinguish between UDP and TCP

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer

- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

UDP: User Datagram Protocol [RFC 768]

❐ "no frills," "bare bones" Internet transport protocol

❐ "best effort" service, UDP segments may be:
  ○ lost
  ○ delivered out of order to app

❐ *connectionless:*
  ○ no handshaking between UDP sender, receiver
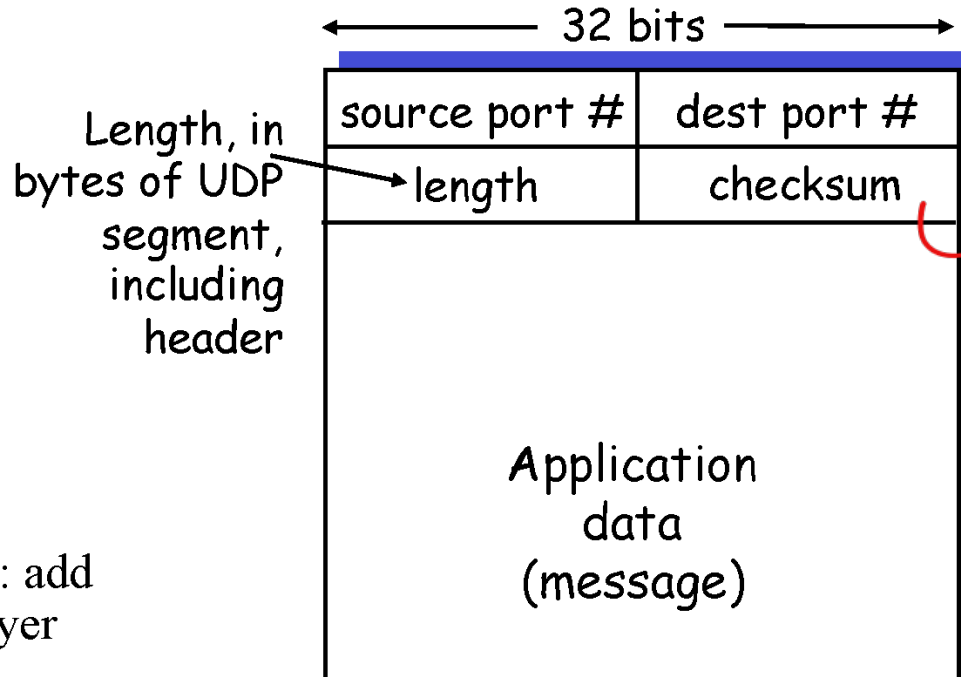 ○ each UDP segment handled independently of others

Why is there a UDP?
❐ no connection establishment (which can add delay)
❐ simple: no connection state at sender, receiver
❐ small segment header
❐ no congestion control: UDP can blast away as fast as desired

UDP: more

多重化 (multiplexing, demultiplexing, 에러 체크)

- ❐ often used for streaming multimedia apps
  - ○ loss tolerant
  - ○ rate sensitive

- ❐ **other UDP uses**
  - ○ DNS
  - ○ SNMP

- ❐ reliable transfer over UDP: add reliability at application layer
  - ○ application-specific error recovery!

Length, in bytes of UDP segment, including header

← 32 bits →	
source port #	dest port #
length	checksum
Application data (message)	

전송 과정에 에러 있는지 판단

UDP segment format