Network Programming I & Network Programming II

侯旭森 贾博暄 许珈铭

2023.12.11

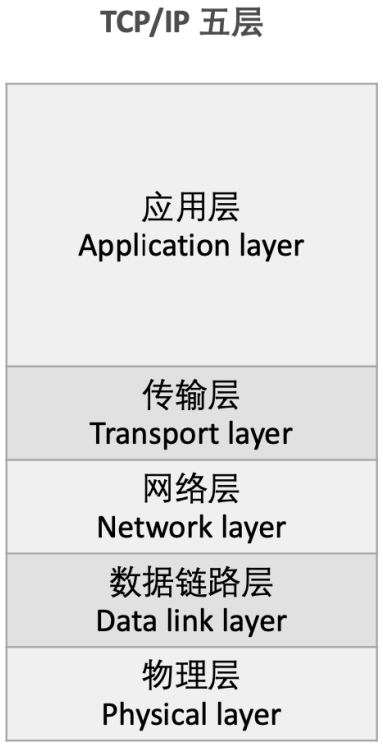
Preface

许珈铭

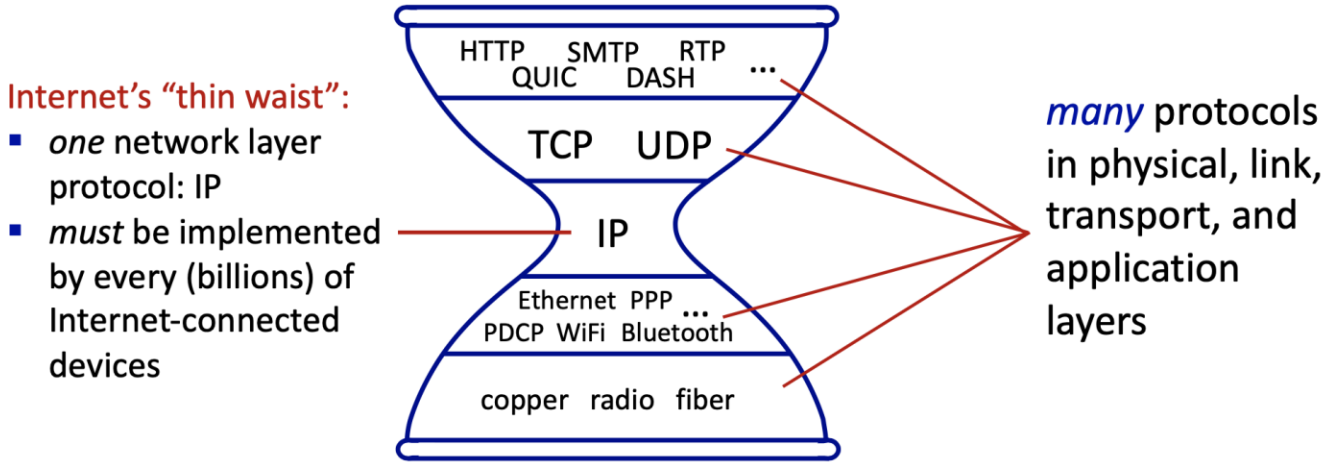
A Network Tour: Preliminaries
网络漫游

Before we start…

You should know of the concept of **protocol stack**, and the thin waist of it.



- Network functionality is too complex to implement all at once.
- The industry takes the classical **layered** approach.



Network Programming I (CS:APP Ch. 11.1-11.4.6)

侯旭森

客户端-服务器编程模型

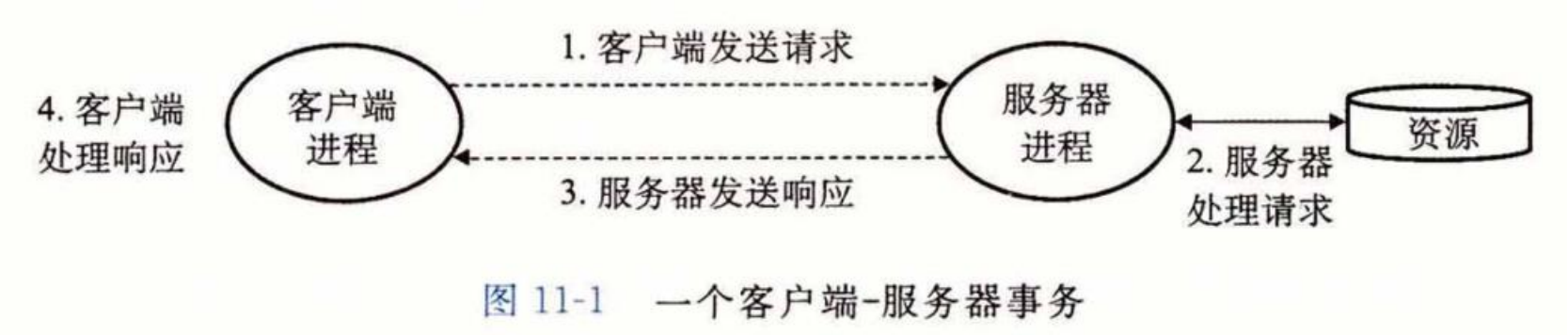
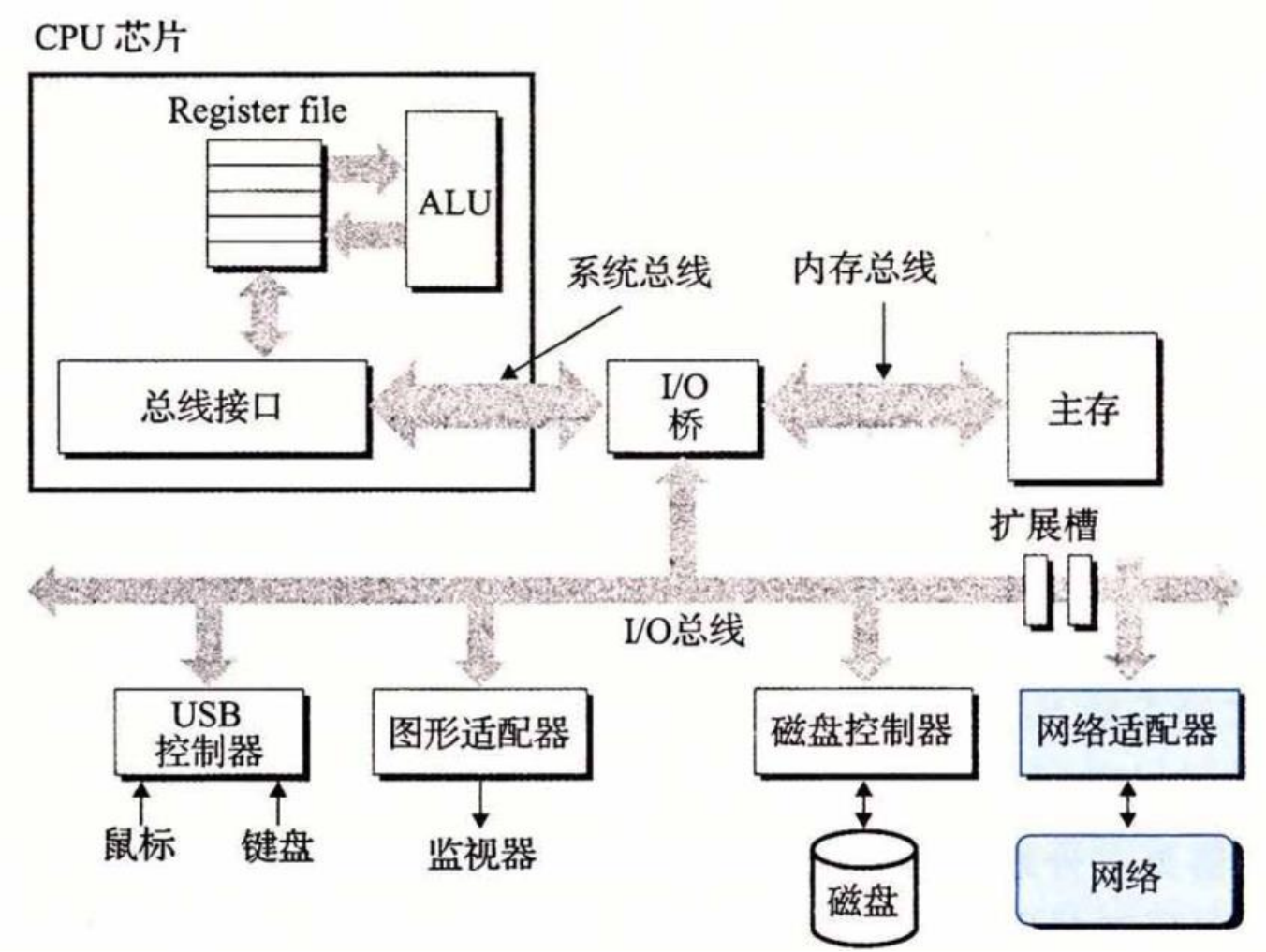


图 11-1　一个客户端-服务器事务

客户端和服务器是进程

网络



- 对主机而言，网络只是又一种I/O设备。=> 描述符、socket等抽象；
- 从网络上接受的数据通常通过DMA(直接内存访问,P413)传送到内存 。

LAN(Local Area Network,局域网)

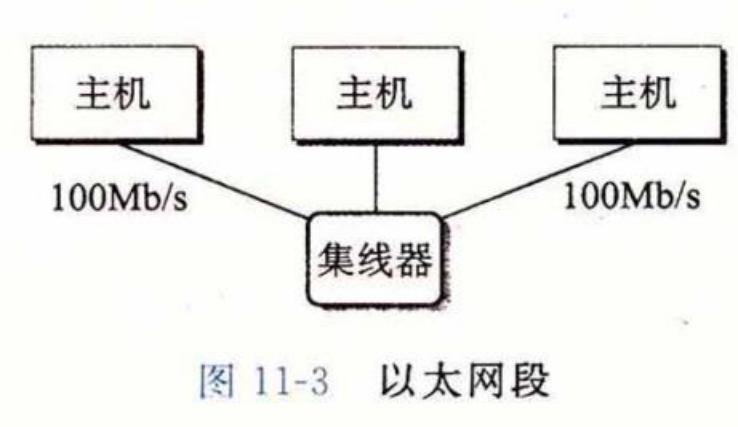
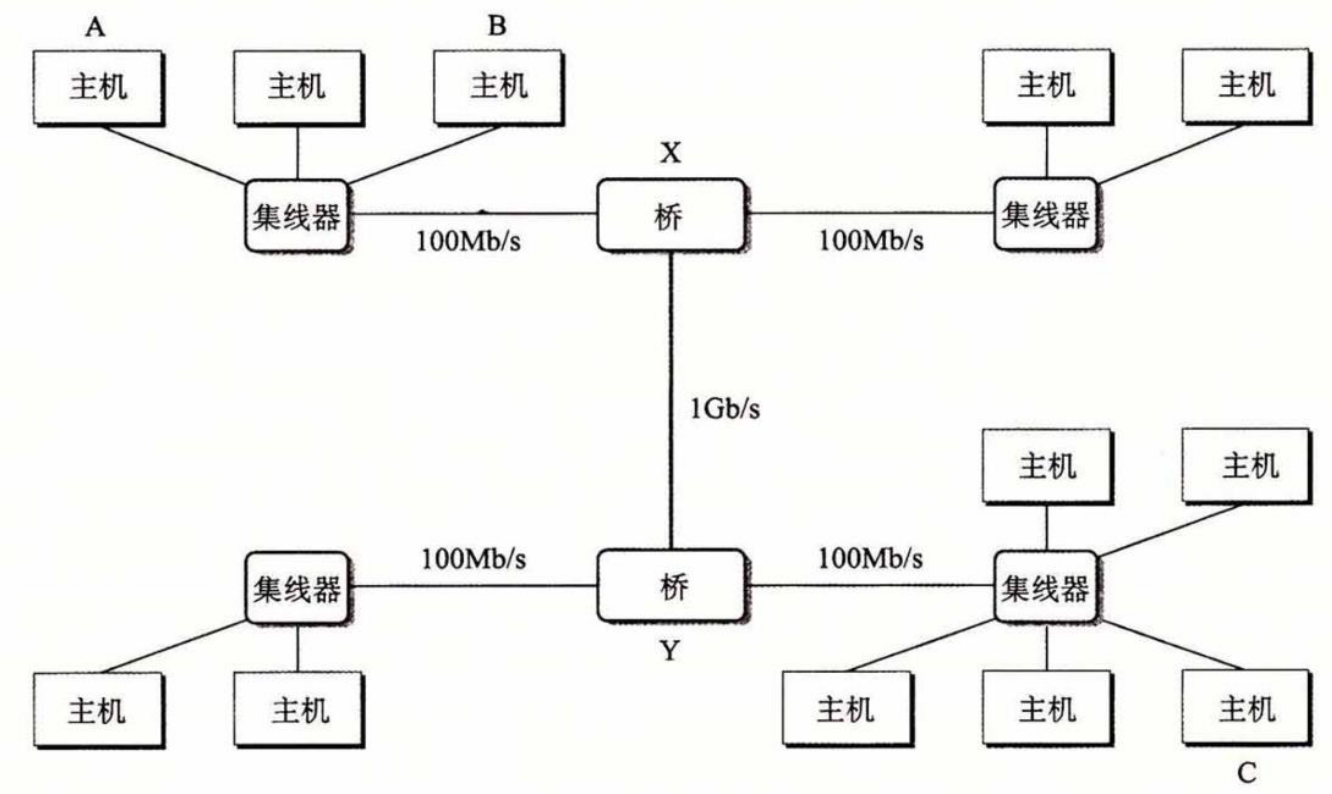


图 11-3　以太网段

- 最底层;
- 以太网(Ethernet)是目前最流行的局域网;
- 集线器不加分辨地将收到的每个位复制到其他所有端口上;
- 每台主机都可以看到每个位;
- 以太网段(Ethernet segment)。

桥接以太网(bridged Ethernet)



- 利用一种聪明的分配算法，自动学习哪个主机可以通过哪个端口到达；
- 局域网表示的简化；

图 11-5　局域网的概念视图

internet互联网络

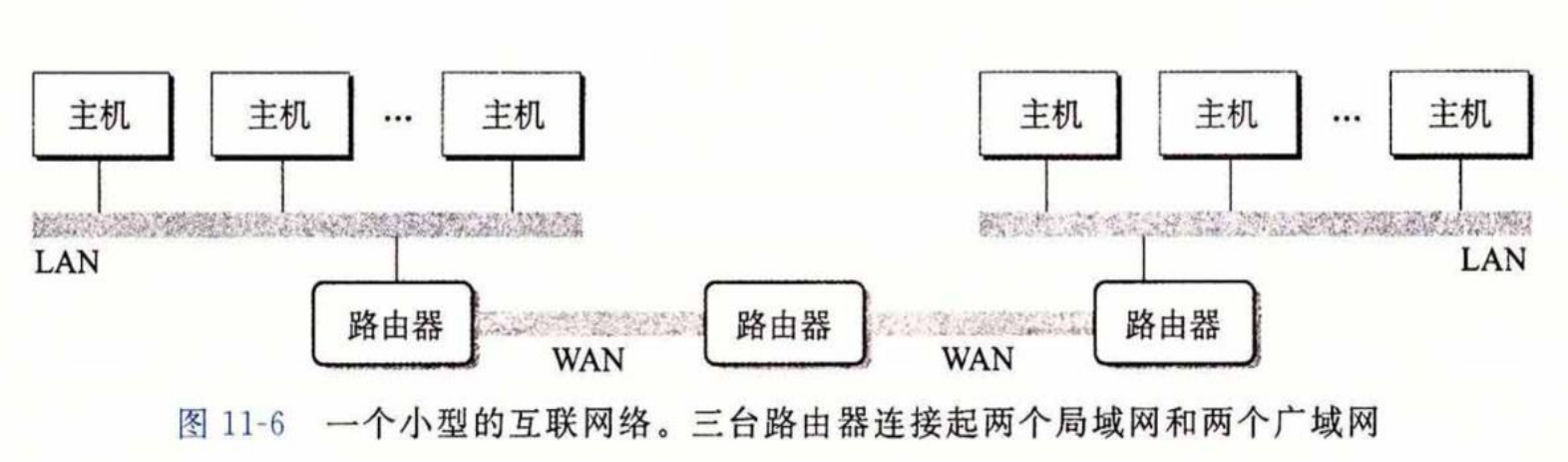
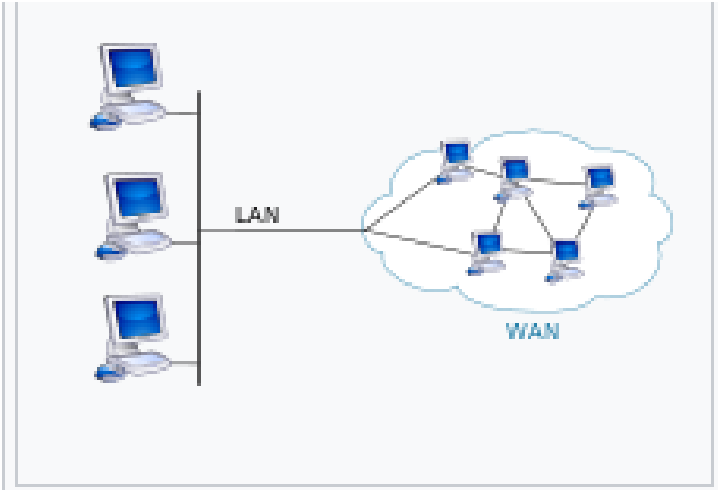


图 11-6　一个小型的互联网络。三台路由器连接起两个局域网和两个广域网



A local area network (LAN) with connection to a wide area network (WAN)

- 多个不兼容的局域网可以通过路由器(router)连接起来，组成一个internet(互联网络)；
- 路由器点对点连接形成广域网(WAN,Wide-Area Network);路由器可以用来由各种局域网和广域网构建互联网络；
- internet描述一般概念，而Internet描述一种具体的实现——全球IP因特网。

封装是关键

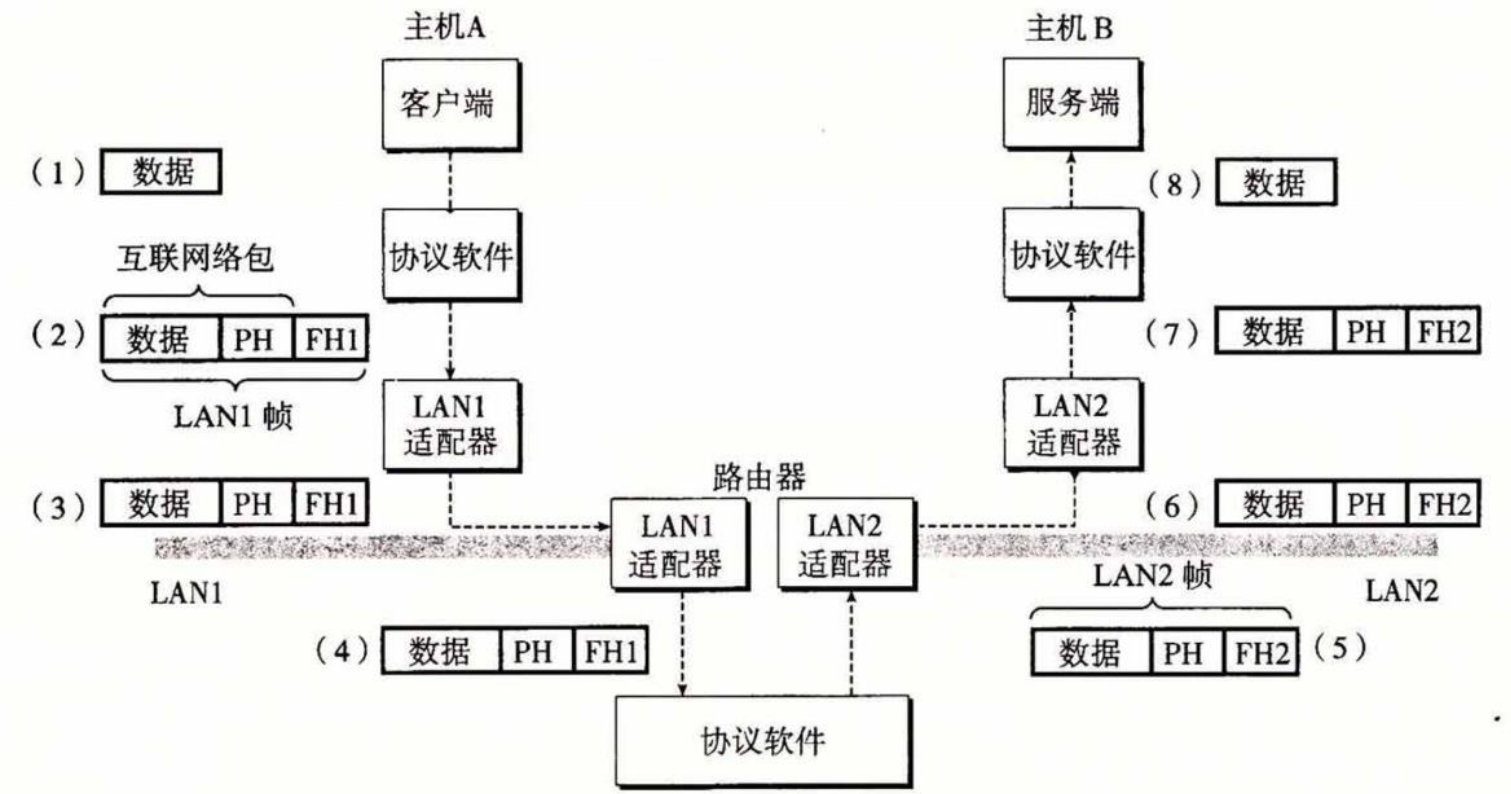
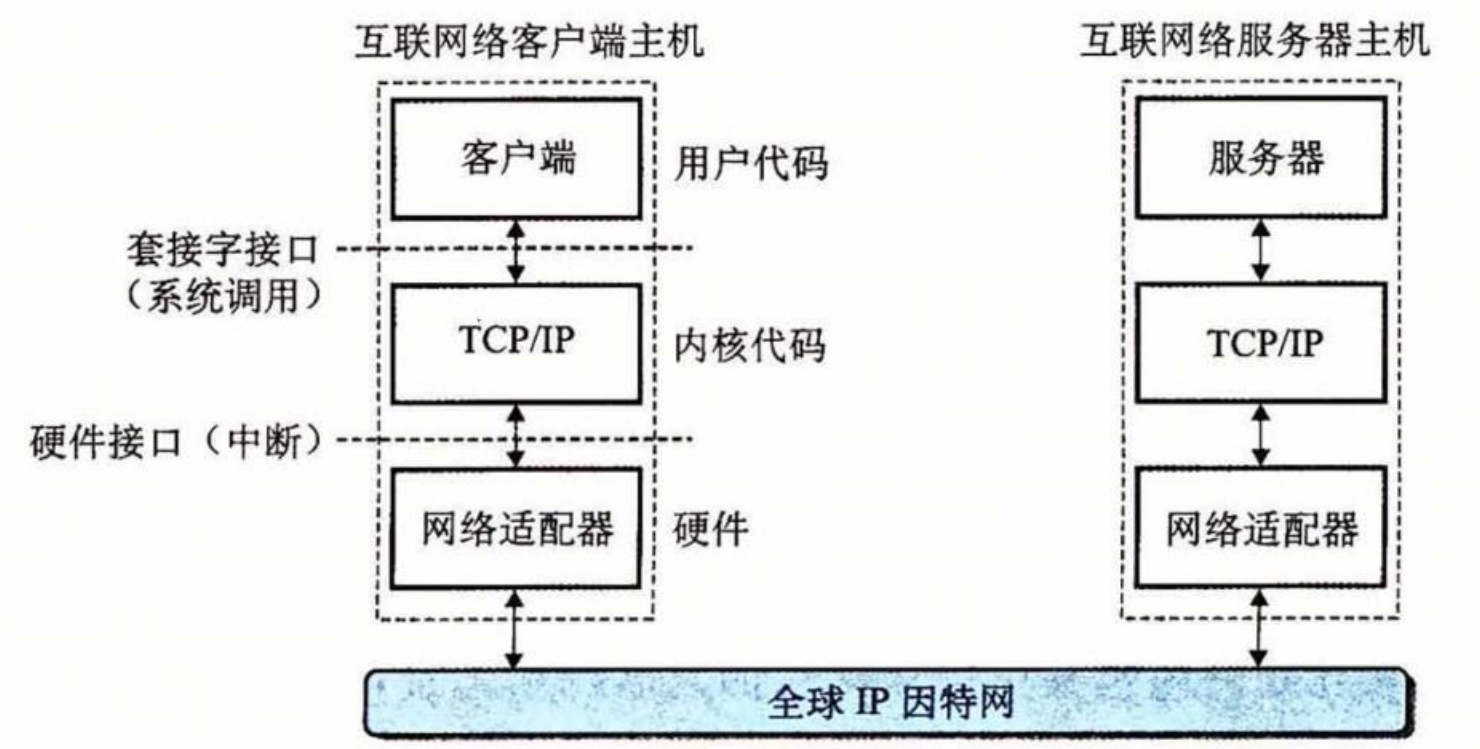


图 11-7 在互联网络上，数据是如何从一台主机传送到另一台主机的(PH：互联网络包头；FH1：LAN1 的帧头；FH2：LAN2 的帧头)

- 网段通过帧（头+有效载荷）封装；
- 广域网互联网包头寻址到互联网络主机B，LAN1帧头寻址到路由器。

全球IP因特网


互联网络客户端主机
互联网络服务器主机
客户端
用户代码
服务器
套接字接口
（系统调用）
TCP/IP
内核代码
TCP/IP
硬件接口（中断）
网络适配器
硬件
网络适配器
全球 IP 因特网

协议

协议层次	协议名称	是面向连接的吗？
网络层	IP	否
应用层	HTTP	否
传输层	TCP	是

- IP:Internet Protocol,互联网络协议;
- TCP:Transmission Control Protocol,传输控制协议;
- HTTP:Hypertext Transfer Protocol,超文本传输协议,一个基于文本的应用级协议。
- 面向连接的协议保障数据按照发送时的顺序被接收。
- TCP是一个构建在IP之上的复杂协议，提供了进程间可靠的全双工（双向的）连接。
- ICP/IP是一个协议群，我们将其看作一个单独的整体协议。

IP地址

code/netp/netpfragments.c

```
/* IP address structure */
struct in_addr {
    uint32_t  s_addr; /* Address in network byte order (big-endian) */
};
```

code/netp/netpfragments.c

图 11-9　IP 地址结构

- IPv4\IPv6;
- 网络字节顺序大端法(TCP/IP定义的);
- 点分十进制表示法;
- 域名(domain name);
- 域名集合和IP地址集合之间的映射通过DNS(Domain Name System,域名系统)维护。
- 域名与IP地址之间一对一、一对多、多对一、多对多都有。
- 本地域名localhost->回送地址(127.0.0.1)方便调试

IP地址表示方式的转换函数们

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
```

返回：按照网络字节顺序的值。

```
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(unit16_t netshort);
```

返回：按照主机字节顺序的值。

```
#include <arpa/inet.h>

int inet_pton(AF_INET, const char *src, void *dst);
```

返回：若成功则为1，若src为非法点分十进制地址则为0，若出错则为−1。

```
const char *inet_ntop(AF_INET, const void *src, char *dst,
                      socklen_t size);
```

返回：若成功则指向点分十进制字符串的指针，若出错则为NULL。

```
linux> nslookup www.twitter.com
Address: 199.16.156.6
Address: 199.16.156.70
Address: 199.16.156.102
Address: 199.16.156.230
```

因特网连接

- 一个套接字是连接的一个端点；
- 每个套接字有相应的套接字地址，用“地址:端口”表示，端口大小为16位；
- 客户端的端口是内核自动分配的临时端口(ephemeral port),服务器端口通常是知名端口，这样是为了方便，其实是可以改的。
- 一个连接由两端套接字地址（socket pair,套接字对）唯一确定的。
- (cliaddr:cliport, servaddr:servport)

有特定意义的IP和端口号

0.0.0.0：0.0.0.0地址被用于表示一个无效的，未知的或者不可用的目标。

x.x.x.0:当一个ip地址的主机部分全为0时，其代表一个网络地址。

255.255.255.255：受限的广播地址，该地址用于主机配置过程中IP数据报的目的地址。

x.x.x.255:当一个ip地址的主机部分全为1时，其代表一个广播地址。

127.0.0.1：A类网络地址127.X.X.X被用作环回地址。习惯上采用127.0.0.1作为环回地址，命名为localhost。

私网地址的范围：A类地址：10.0.0.0～10.255.255.255，B类地址：172.16.0.0 ～172.31.255.255，C类地址：192.168.0.0～192.168.255.255。所以根据局域网的ip大致可以猜出，这个局域网内的主机个数不超过多少个。

公网地址：剩下的基本就是公网地址了，私网能够上网就是通过公网nat技术实现接入互联网的，也能通过ip定位对方的位置。所以少在网上骂人骗人，真的可以顺着网线来找你。

课本：
Web服务器80
电子邮件服务器25
更多请访问：
https://zhuanlan.zhihu.com/p/420272619

名词回顾

- TCP/IP/HTTP/DNS/LAN/WAN/internet/Internet/...
- 以太网、以太网端、网桥、桥接以太网、套接字、IP地址、域名、套接字地址、套接字对...
- htonl/htons/ntohl/ntohs/inet_pton/inet_ntop/nslookup...

套接字接口

code/netp/netpfragments.c

```
/* IP socket address structure */
struct sockaddr_in  {
    uint16_t        sin_family;  /* Protocol family (always AF_INET) */
    uint16_t        sin_port;    /* Port number in network byte order */
    struct in_addr  sin_addr;    /* IP address in network byte order */
    unsigned char   sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};

/* Generic socket address structure (for connect, bind, and accept) */
struct sockaddr {
    uint16_t  sa_family;    /* Protocol family */
    char      sa_data[14];  /* Address data  */
};
```

code/netp/netpfragments.c

```
typedef struct sockaddr SA;
```

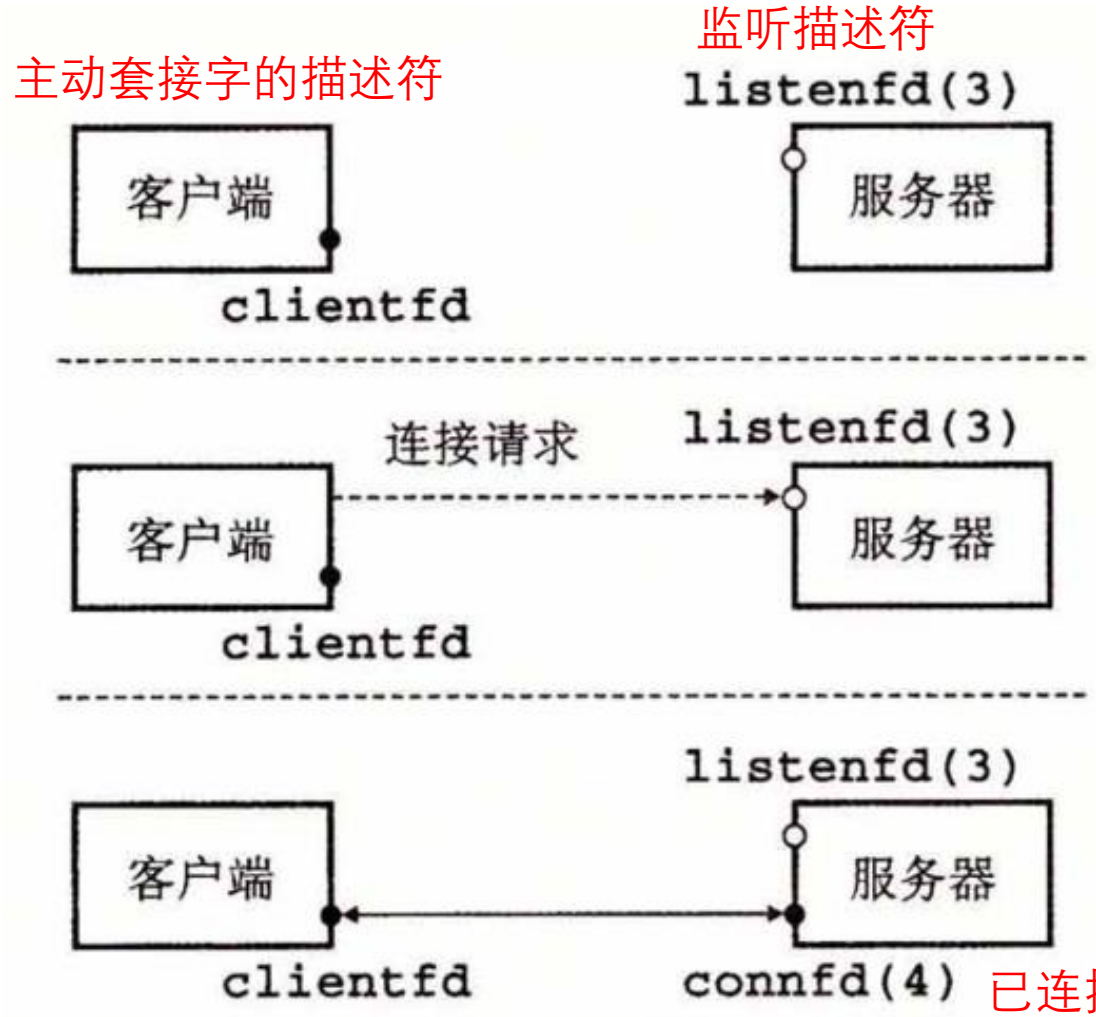
然后无论何时需要将 sockaddr_in 结构强制转换成通用 sockaddr 结构时，我们都使用这个类型。

套接字接口

- 从I/O抽象的角度，服务器对客户端而言是通过描述符读写的文件，客户端对服务器也是通过描述符读写的文件；
- 客户端使用的描述符应该指定特定的服务器（套接字地址），即描述符的创建过程中需要服务器的套接字地址；
- 服务器使用的描述符也需要指定特定的客户端（套接字地址）；
- 客户端的代码很容易指定特定的服务器，但是服务器怎么知道哪个客户端要访问？

套接字接口

注：监听描述符和已连接描述符共用一个套接字地址!



1. 服务器阻塞在 accept，等待监听描述符 listenfd 上的连接请求。

2. 客户端通过调用和阻塞在 connect，创建连接请求。

3. 服务器从 accept 返回 connfd。客户端从 connect 返回。现在在 clientfd 和 connfd 之间已经建立起了连接。

图 11-14　监听描述符和已连接描述符的角色

（1） 一个服务器拥有两个独立的固定 IP 地址，那么它在 web 应用端口 80 上最多可以监听多少个独立的 socket 连接？（2 分）

（1） 一个服务器拥有两个独立的固定 IP 地址，那么它在 web 应用端口 80 上最多可以监听多少个独立的 socket 连接？（2 分）

答案：$2*2^{48}$

服务器端	客户端	结果
2 个独立固定 IP	任意 32 位 IP 任意 16 位 port number	$2*2^{32+16}$

套接字接口

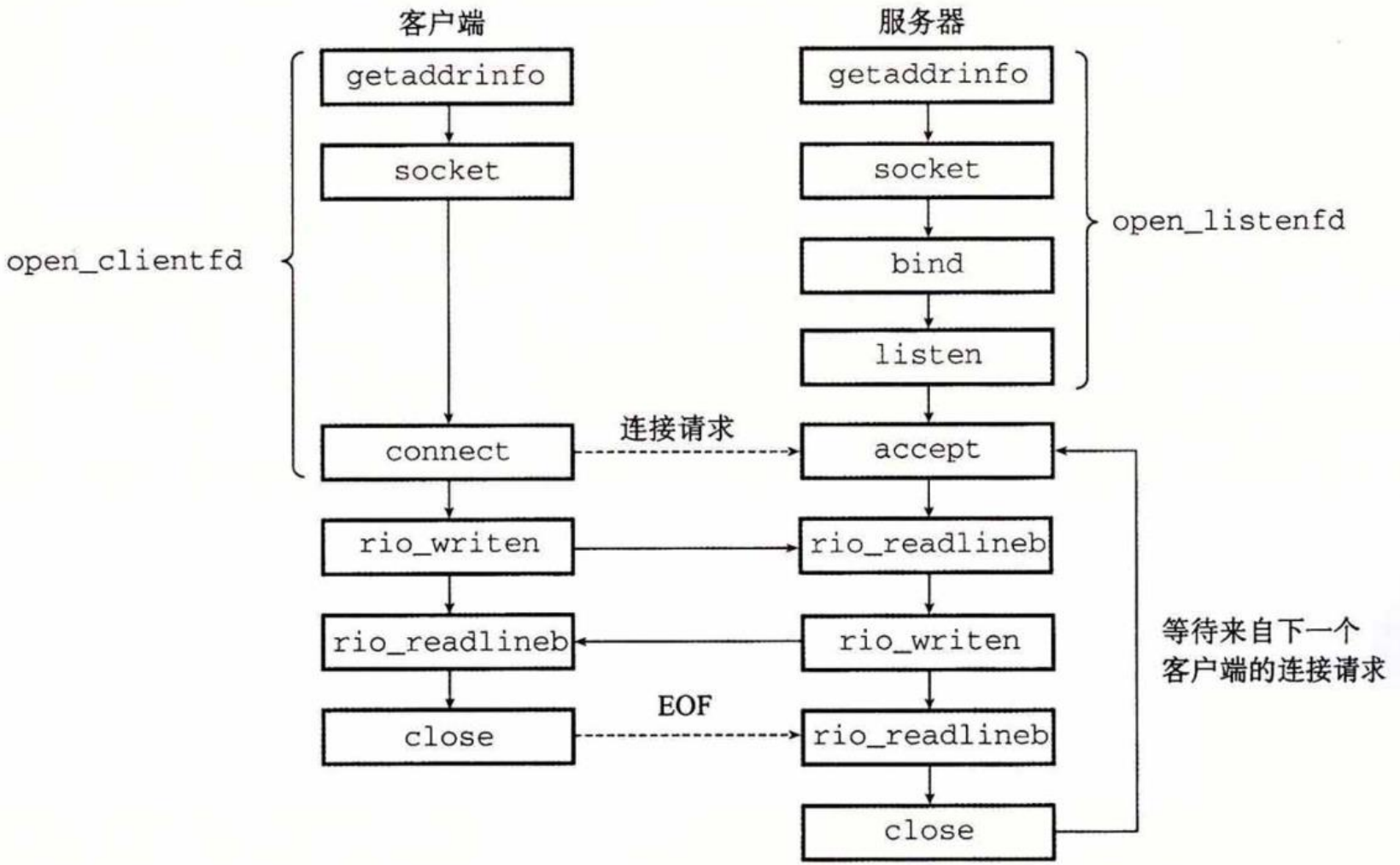
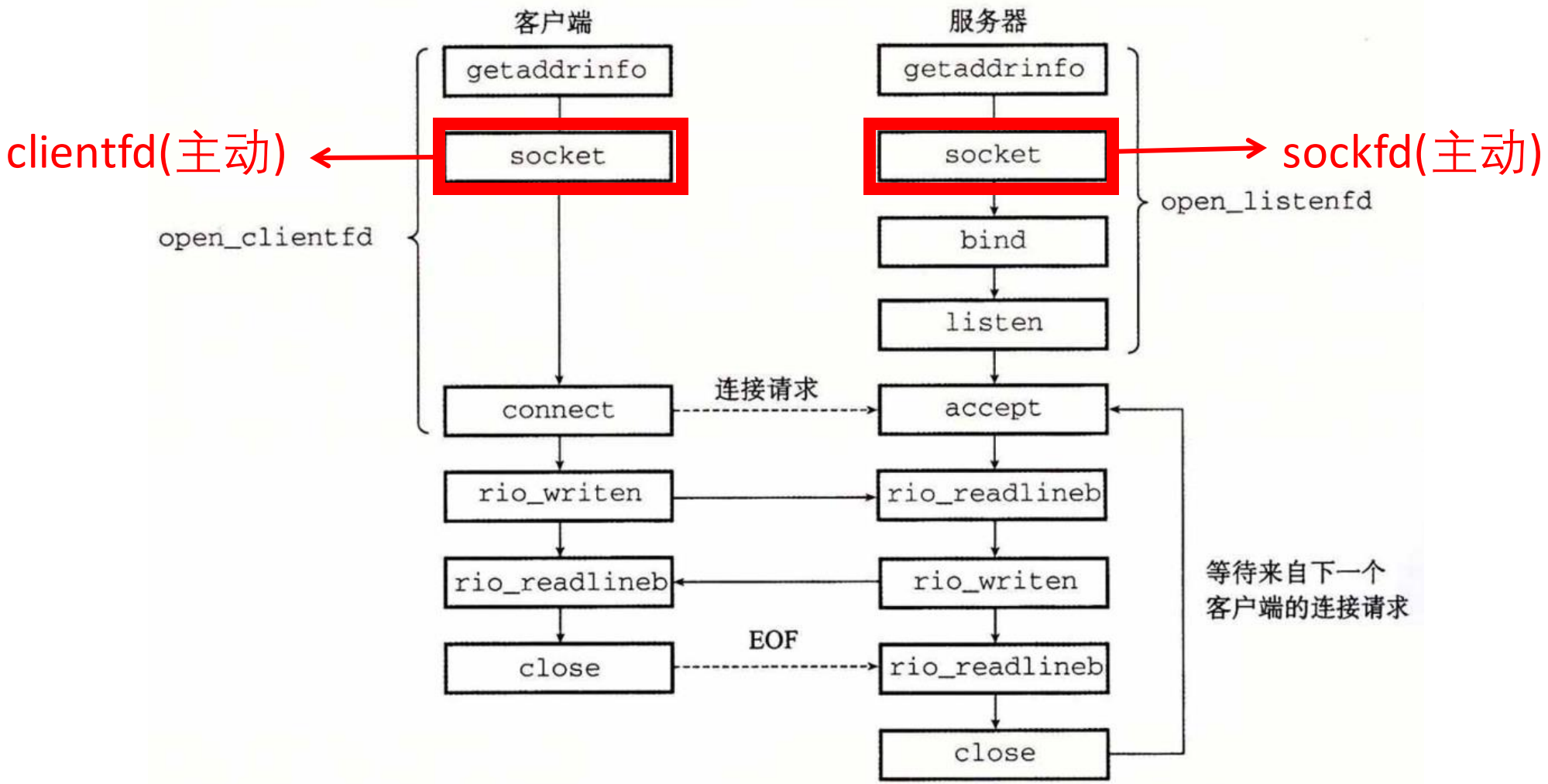


图 11-12　基于套接字接口的网络应用概述

- 主动套接字
- 监听描述符
- 已连接描述符



客户端和服务器使用 socket 函数来创建一个套接字描述符(socket descriptor)。

```
#include <sys/types.h>
#include <sys/socket.h>

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

返回：若成功则为非负描述符，若出错则为−1。

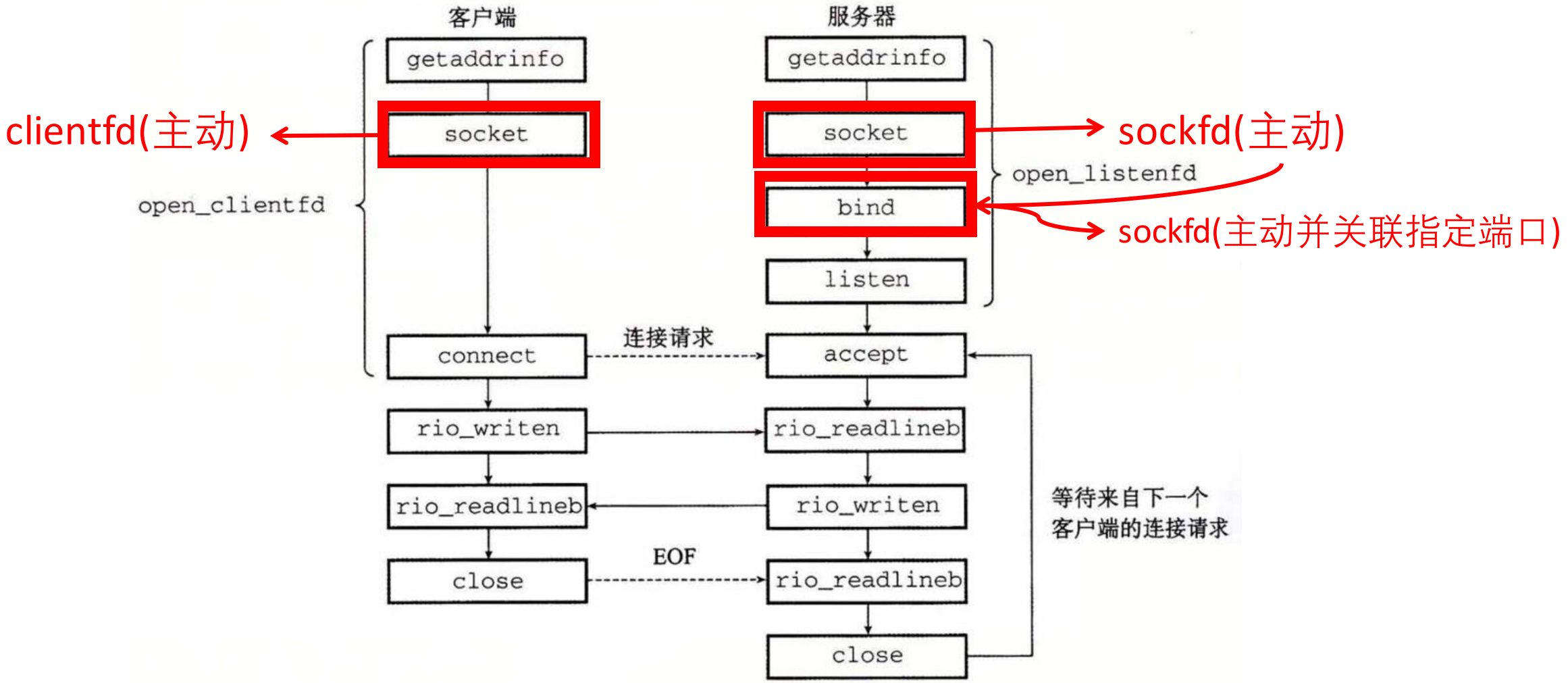


图 11-12　基于套接字接口的网络应用概述

```
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);
```

返回：若成功则为 0，若出错则为 −1。

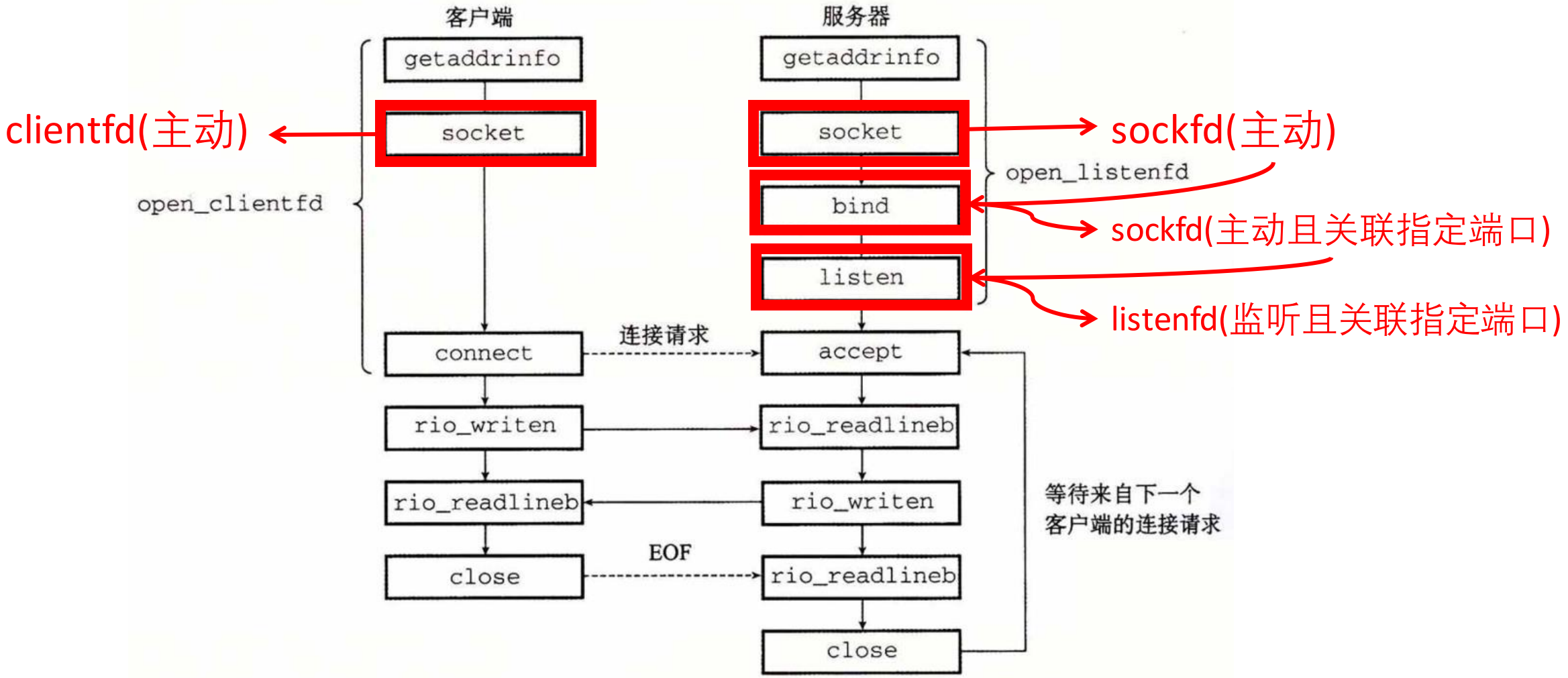
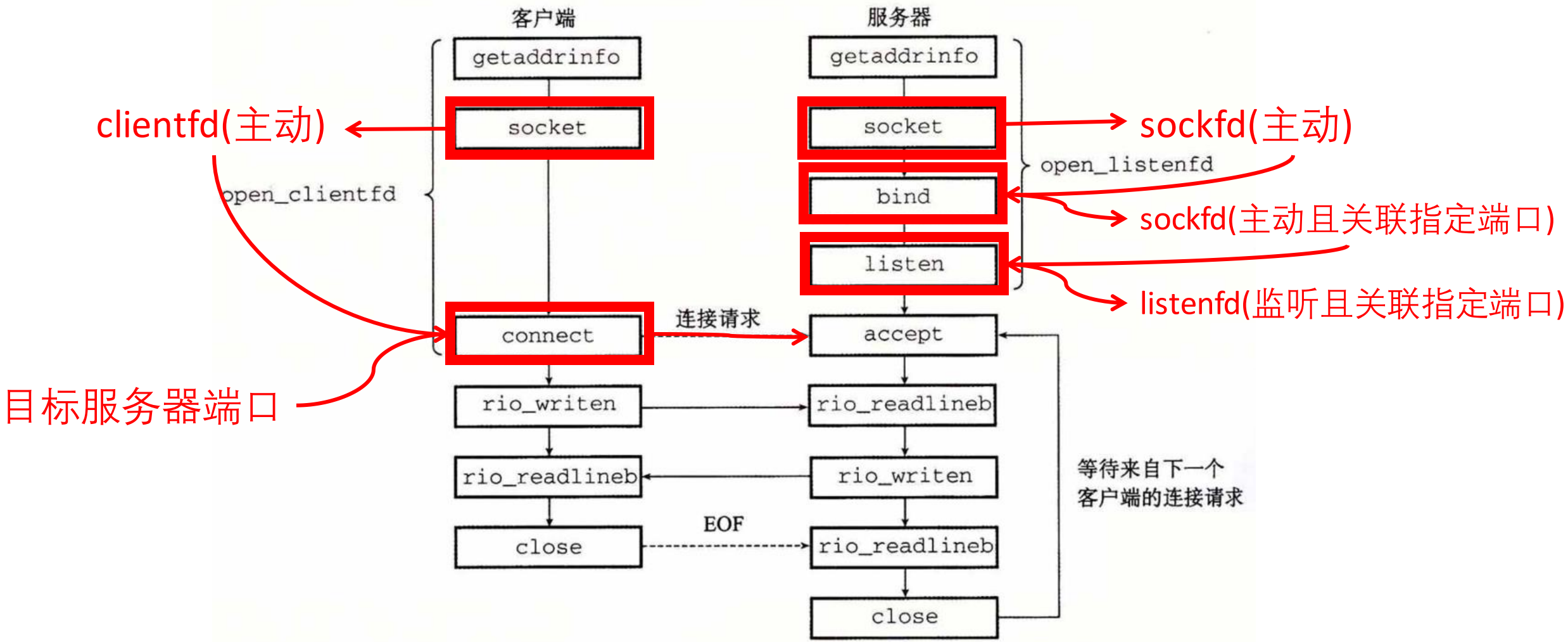


图 11-12　基于套接字接口的网络应用概述

```
#include <sys/socket.h>

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

返回：若成功则为 0，若出错则为 −1。



客户端通过调用 connect 函数来建立和服务器的连接。

```
#include <sys/socket.h>

int connect(int clientfd, const struct sockaddr *addr,
            socklen_t addrlen);
```

返回：若成功则为 0，若出错则为 −1。

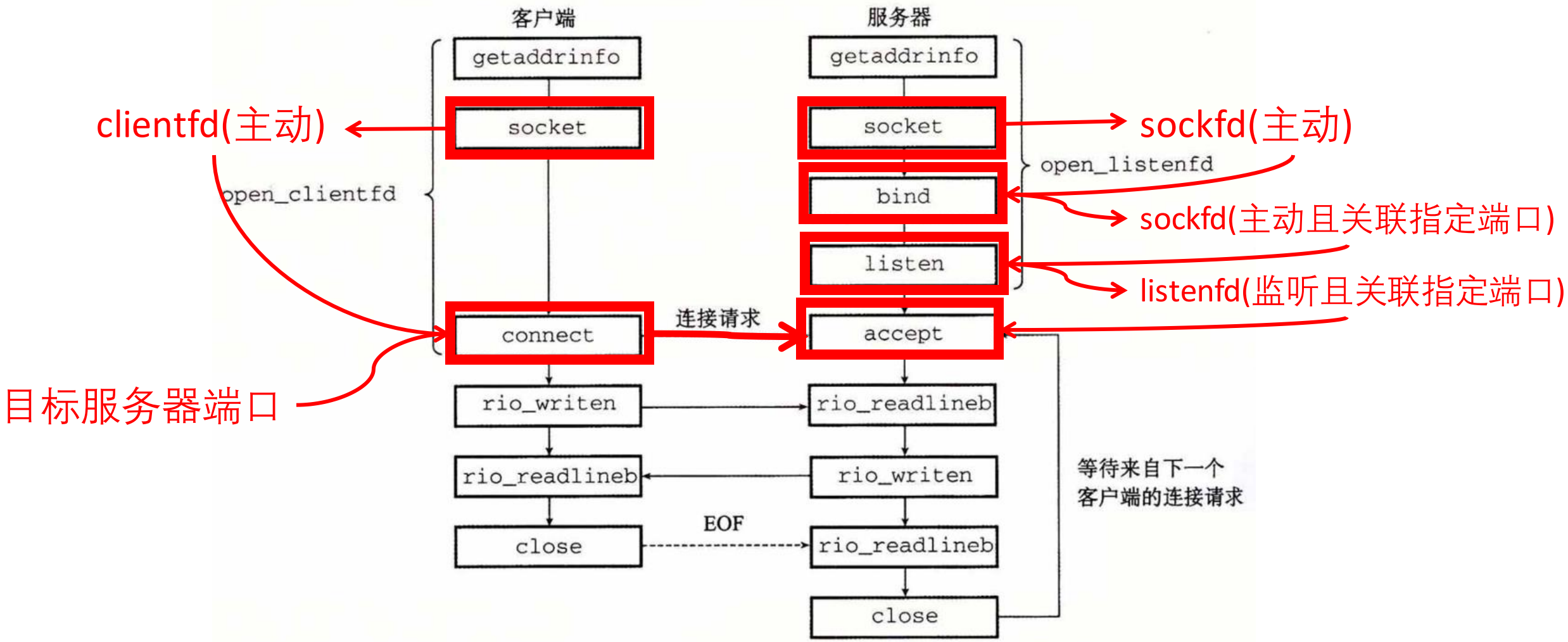


图 11-12　基于套接字接口的网络应用概述

```
#include <sys/socket.h>

int accept(int listenfd, struct sockaddr *addr, int *addrlen);
```

返回：若成功则为非负连接描述符，若出错则为−1。

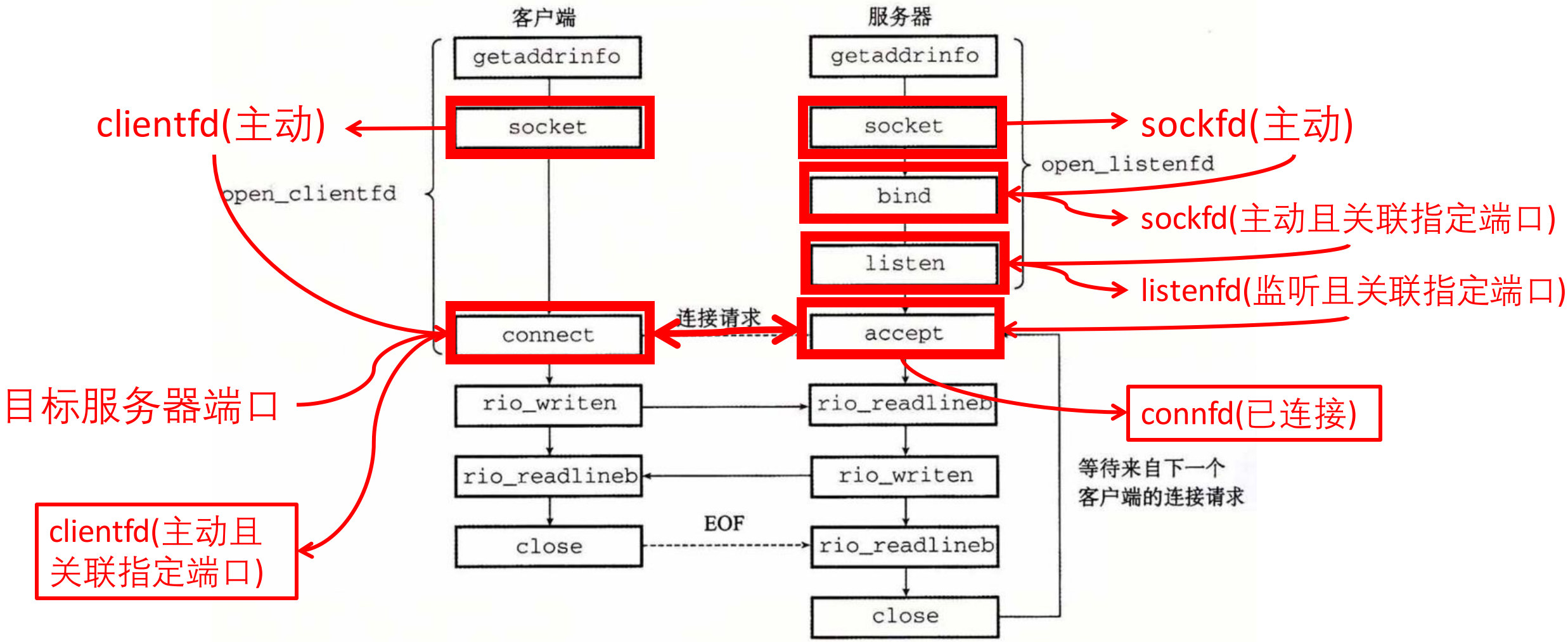


图 11-12　基于套接字接口的网络应用概述

```
#include <sys/socket.h>

int accept(int listenfd, struct sockaddr *addr, int *addrlen);
```

返回：若成功则为非负连接描述符，若出错则为−1。

Network Programming II (CS:APP Ch. 11.4.7-11.6)

贾博暄

Overview

- **Sockets interface revisited**
  - Wrappers
    - **`getaddrinfo`** and **`getnameinfo`**
    - **`open_clientfd`** and **`open_serverfd`**
  - Client / Server session: Reading & Writing
    - The echo server
- **HTTP**
  - Requests & Responses
  - Dynamic content: CGI
- **The TINY web server**

The best way to learn the sockets interface is to study example code.

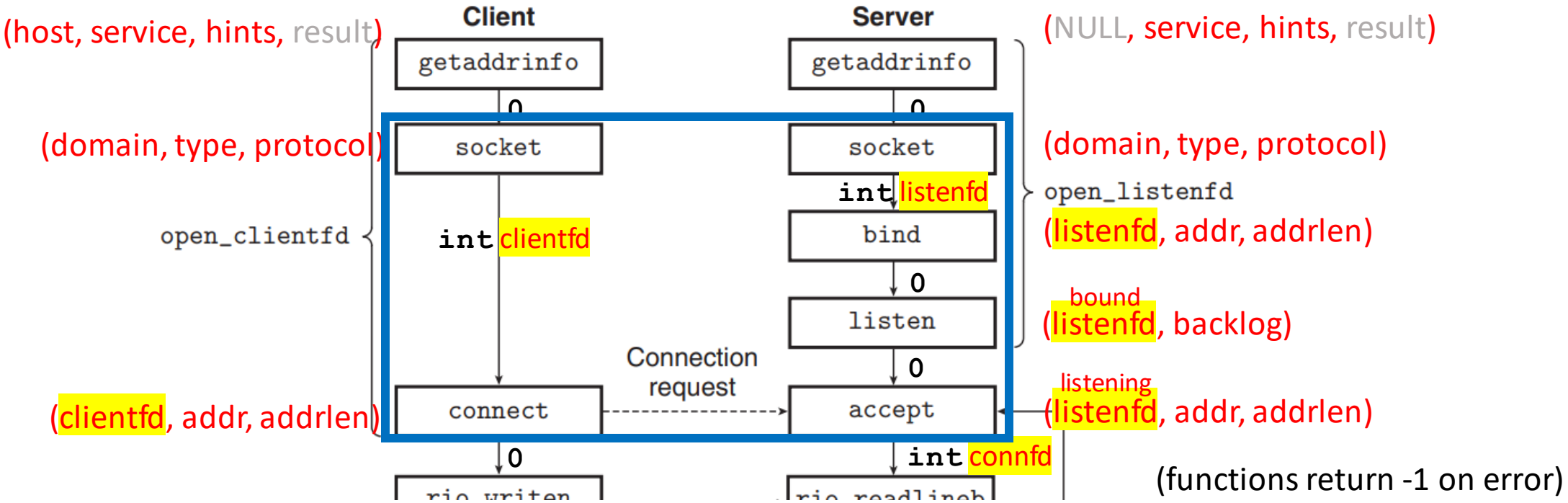
——CS:APP3e, Page 980

Today…

- **Sockets interface revisited**
  - Wrappers
    - `getaddrinfo` and `getnameinfo`
    - `open_clientfd` and `open_serverfd`
  - Client / Server session: Reading & Writing
    - The echo server
- HTTP
  - Requests & Responses
  - Dynamic content: CGI
- The TINY web server

Where we are

- We've introduced **these** functions…



- **A bit messy…**

Today…

- Sockets interface revisited
  - **Wrappers**
    - `open_clientfd` and `open_serverfd`
    - `getaddrinfo` and `getnameinfo`
  - Client / Server session: Reading & Writing
    - The echo server
- HTTP
  - Requests & Responses
  - Dynamic content: CGI
- The TINY web server

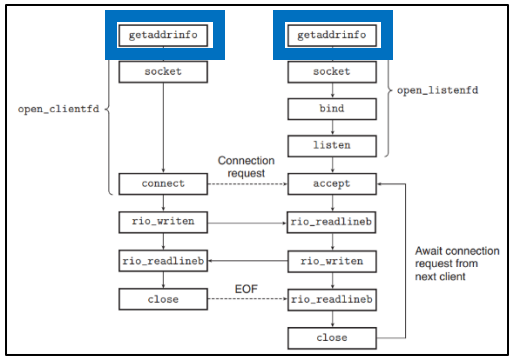
Wrappers

- Fortunately, CS:APP provides wrapper functions
  - **`open_clientfd`** and **`open_listenfd`**
- We try to motivate them in an intuitive way

Today…

- Sockets interface revisited
  - Wrappers
    - **`getaddrinfo`** and **`getnameinfo`**
    - **`open_clientfd`** and **`open_serverfd`**
  - Client / Server session: Reading & Writing
    - The echo server
- HTTP
  - Requests & Responses
  - Dynamic content: CGI
- The TINY web server

But before we dive in…



- **`getaddrinfo`**
  - A "better" **`inet_pton`**
  - Remember "multiple addresses" from minutes ago?

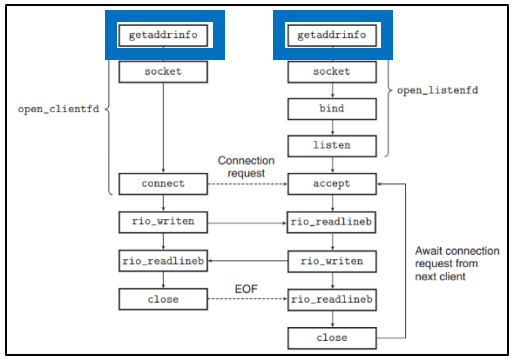
```
int getaddrinfo(const char *host, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **result);
                                  Returns: 0 if OK, nonzero error code on error
```

  - Argument **`hints`** offer customization
    - eg. Client or Server?
  - Stores **`addrinfo`** in result
    - Use **`freeaddrinfo`** to free

Recall:
`p` = presentation
(dotted decimal, e.g., `127.0.0.1`)
`n` = network
(e.g., `0x7f000001`)

```
linux> nslookup twitter.com
Address: 199.16.156.102
Address: 199.16.156.230
Address: 199.16.156.6
Address: 199.16.156.70
```

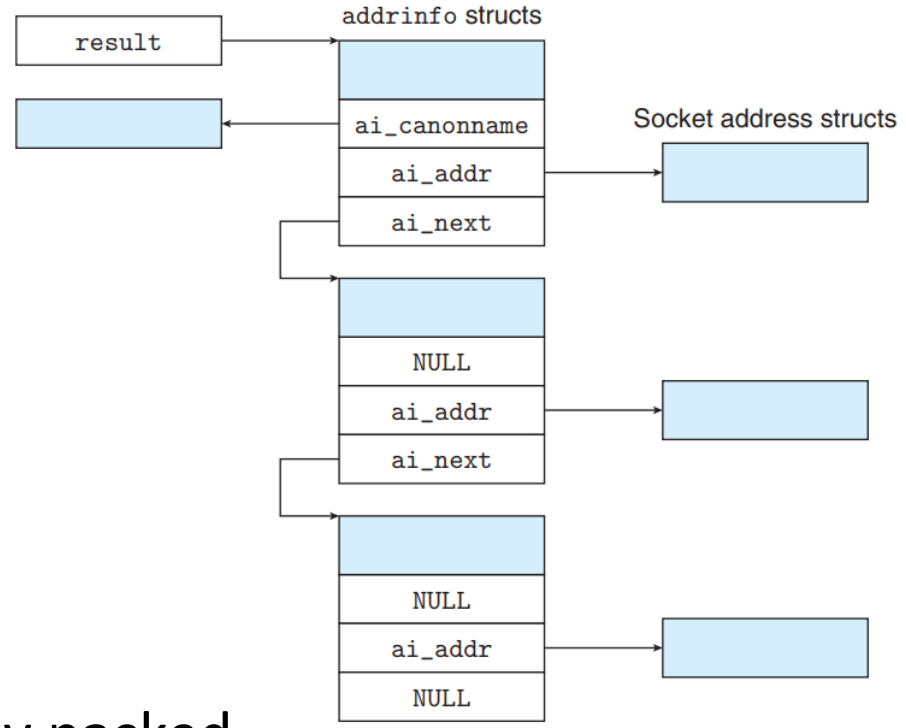
But before we dive in…



- The **`addrinfo`** struct

```
struct addrinfo {
    int              ai_flags;     /* Hints argument flags */
    int              ai_family;    /* First arg to socket function */
    int              ai_socktype;  /* Second arg to socket function */
    int              ai_protocol;  /* Third arg to socket function  */
    char            *ai_canonname; /* Canonical hostname */
    size_t           ai_addrlen;   /* Size of ai_addr struct */
    struct sockaddr *ai_addr;      /* Ptr to socket address structure */
    struct addrinfo *ai_next;      /* Ptr to next item in linked list */
};
```

  - Linked list
    - Remember “multiple addresses” from minutes ago?
    - Traverse for connection
  - Args for **`socket`**, **`connect`**, **`listen`** are neatly packed



But before we dive in…

- **`getnameinfo`**
  - A "better" **`inet_ntop`**

Recall:
`p` = presentation
(dotted decimal, e.g., `127.0.0.1`)
`n` = network
(e.g., `0x7f000001`)

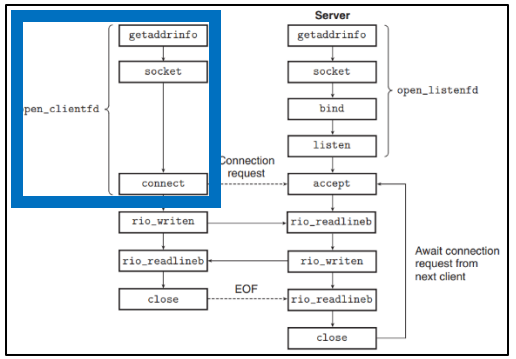
```
int getnameinfo(const struct sockaddr *sa, socklen_t salen,
                char *host, size_t hostlen,
                char *service, size_t servlen, int flags);
                                    Returns: 0 if OK, nonzero error code on error
```

  - Domain name in `host`
    - (can configure `FLAGS` to return numeric address string)
  - Service name in `service`
    - (can configure `FLAGS` to return port number)
  - (In CS:APP, used only when printing a decimal IP, e.g. HOSTINFO, TINY)

Today…

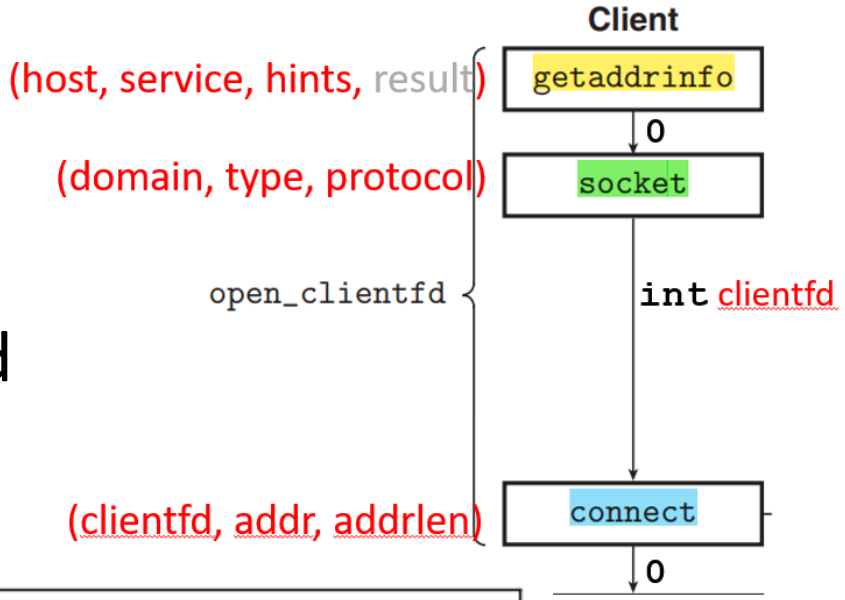
- Sockets interface revisited
  - Wrappers
    - **`getaddrinfo`** and **`getnameinfo`**
    - **`open_clientfd`** and **`open_serverfd`**
  - Client / Server session: Reading & Writing
    - The echo server
- HTTP
  - Requests & Responses
  - Dynamic content: CGI
- The TINY web server

`open_clientfd`: Motivation



In our case, port number ↓

- Simplicity
  - Taking in host and service should be enough!
- There should be 3 major calls
  - **`getaddrinfo`**, **`socket`**, **`connect`**
- **`clientfd`** (from **`socket`**) should be retained
  - This should be the return value



- So…

```
#include "csapp.h"

int open_clientfd(char *hostname, char *port);
```

Returns: descriptor if OK, −1 on error

open_clientfd

Set customized hints

1. ONE **addrinfo** struct
2. Force port number (not service name)
3. Restrict returned addresses to family

listp is the result of **getaddrinfo**, an **addrinfo** struct

Avoid memory leak

Whoops, we walked all the way to NULL ptr

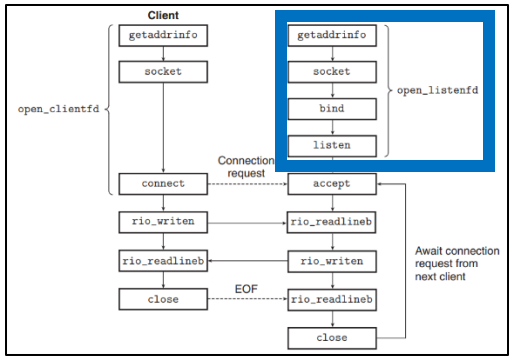
code/src/csapp.c

```
int open_clientfd(char *hostname, char *port) {
    int clientfd;
    struct addrinfo hints, *listp, *p;

    /* Get a list of potential server addresses */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_socktype = SOCK_STREAM;  /* Open a connection */
    hints.ai_flags = AI_NUMERICSERV;  /* ... using a numeric port arg. */
    hints.ai_flags |= AI_ADDRCONFIG;  /* Recommended for connections */
    Getaddrinfo(hostname, port, &hints, &listp);

    /* Walk the list for one that we can successfully connect to */
    for (p = listp; p; p = p->ai_next) {
        /* Create a socket descriptor */
        if ((clientfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) < 0)
            continue; /* Socket failed, try the next */

        /* Connect to the server */
        if (connect(clientfd, p->ai_addr, p->ai_addrlen) != -1)
            break; /* Success */
        Close(clientfd); /* Connect failed, try another */
    }

    /* Clean up */
    Freeaddrinfo(listp);
    if (!p) /* All connects failed */
        return -1;
    else    /* The last connect succeeded */
        return clientfd;
}
```

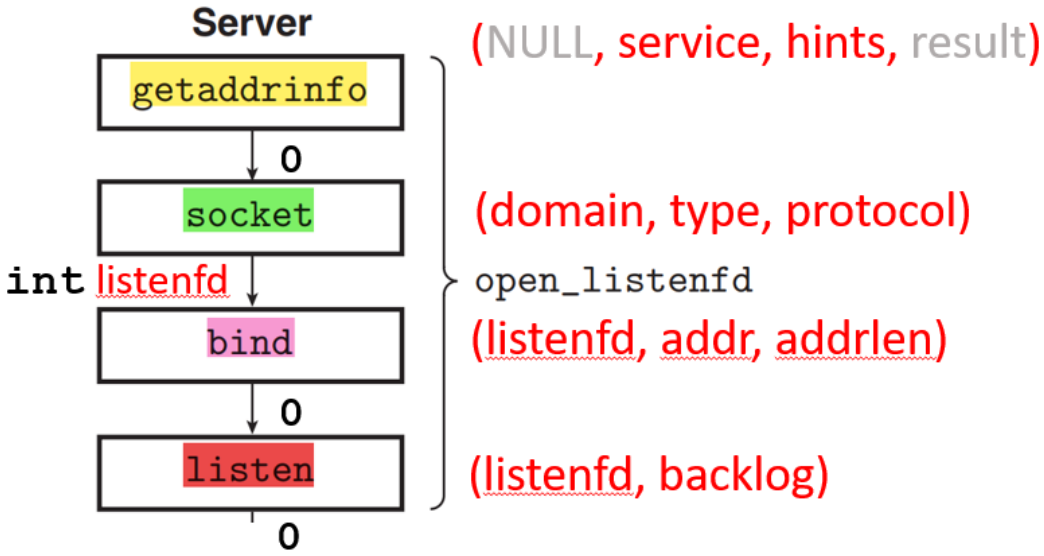
code/src/csapp.c

args are neatly packed in **addrinfo** struct

`open_listenfd`: Motivation



- Again, simplicity
  - Taking in port should suffice!
- 4 major calls this time
  - **`getaddrinfo`**, **`socket`**, **`bind`**, **`listen`**
- Should return **`listenfd`** (from **`socket`**)



- So…

```
#include "csapp.h"

int open_listenfd(char *port);
```

Returns: descriptor if OK, −1 on error

open_listenfd

Set customized hints

Almost same as open_clientfd, with the addition of
1. AI_PASSIVE (i.e., set as listening socket)

Configure to "eliminate cooldown"
(i.e., be terminated, be restarted, and begin accepting connection requests immediately)

- Compare with **open_clientfd**

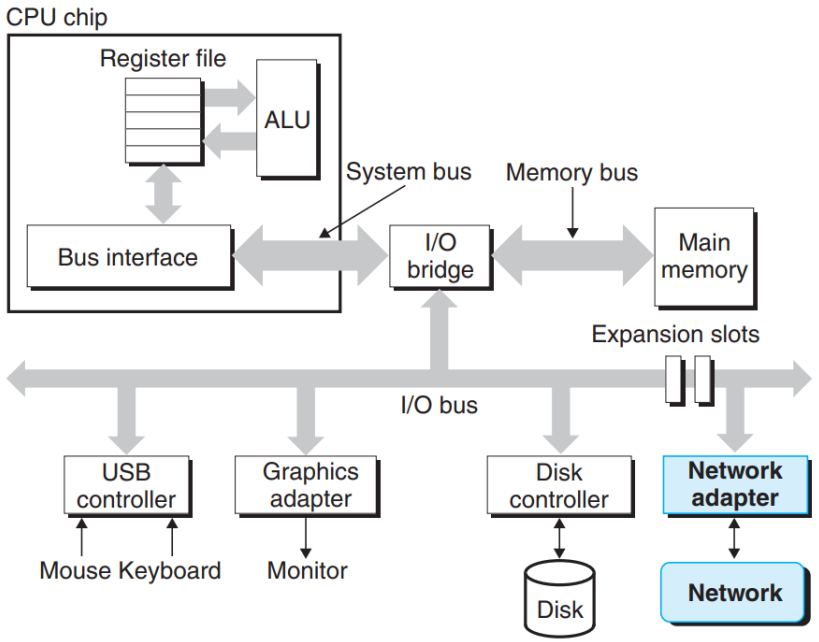
```
int open_listenfd(char *port)
{
    struct addrinfo hints, *listp, *p;
    int listenfd, optval=1;

    /* Get a list of potential server addresses */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_socktype = SOCK_STREAM;             /* Accept connections */
    hints.ai_flags = AI_PASSIVE | AI_ADDRCONFIG; /* ... on any IP address */
    hints.ai_flags |= AI_NUMERICSERV;            /* ... using port number */
    Getaddrinfo(NULL, port, &hints, &listp);

    /* Walk the list for one that we can bind to */
    for (p = listp; p; p = p->ai_next) {
        /* Create a socket descriptor */
        if ((listenfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) < 0)
            continue;  /* Socket failed, try the next */

        /* Eliminates "Address already in use" error from bind */
        Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
                   (const void *)&optval , sizeof(int));

        /* Bind the descriptor to the address */
        if (bind(listenfd, p->ai_addr, p->ai_addrlen) == 0)
            break; /* Success */
        Close(listenfd); /* Bind failed, try the next */
    }

    /* Clean up */
    Freeaddrinfo(listp);
    if (!p) /* No address worked */
        return -1;

    /* Make it a listening socket ready to accept connection requests */
    if (listen(listenfd, LISTENQ) < 0) {
        Close(listenfd);
        return -1;
    }
    return listenfd;
}
```

Today…

- Sockets interface revisited
  - Wrappers
    - `getaddrinfo` and `getnameinfo`
    - `open_clientfd` and `open_serverfd`
  - **Client / Server session: Reading & Writing**
    - The echo server
- HTTP
  - Requests & Responses
  - Dynamic content: CGI
- The TINY web server

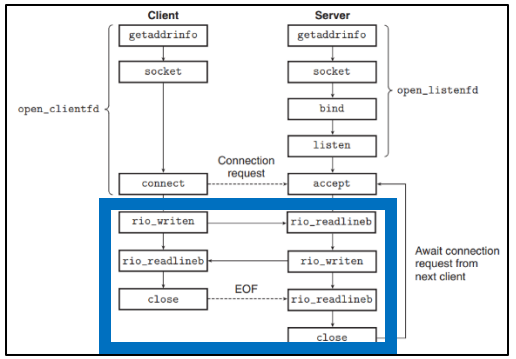
… Remember Chapter 10?

- Network socket is abstracted as a file!
- Recall the socket interface flowchart



- The top half involves multiple **`fd`**'s
  - Functions get abstracted into **`open_clientfd`** & **`open_listenfd`**
  - Remember the system call **`open()`**?
- The bottom half is full of **`read()`**, **`write()`**, **`close()`**
  - Again, system calls!

Reading & Writing



- Robust I/O (rio) from Chapter 10

```
ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
```

Returns: number of bytes read if OK, 0 on EOF, −1 on error

```
ssize_t rio_writen(int fd, void *usrbuf, size_t n);
```

Returns: number of bytes transferred if OK, 0 on EOF (`rio_readn` only), −1 on error

```
#include <unistd.h>

int close(int fd);
```

Returns: 0 if OK, −1 on error

- These are used by both clients and servers

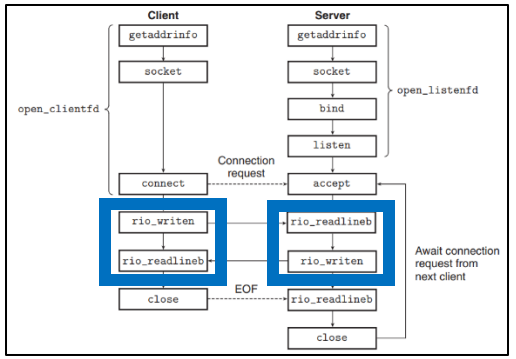
Recall:
`rio_readlineb`
= read next text line, buffered

`writen` = `n` bytes (handles short counts)

Today…

- Sockets interface revisited
  - Wrappers
    - `getaddrinfo` and `getnameinfo`
    - `open_clientfd` and `open_serverfd`
  - Client / Server session: Reading & Writing
    - **The echo server**
- HTTP
  - Requests & Responses
  - Dynamic content: CGI
- The TINY web server

Case: The Echo Server



Client main routine

- Input -> Write -> Read -> Print

`fgets` `rio_writen` `rio_readlineb` `fputs`

```
18
19      while (Fgets(buf, MAXLINE, stdin) != NULL) {
20          Rio_writen(clientfd, buf, strlen(buf));
21          Rio_readlineb(&rio, buf, MAXLINE);
22          Fputs(buf, stdout);
23      }
24      Close(clientfd);
25      exit(0);
26  }
```

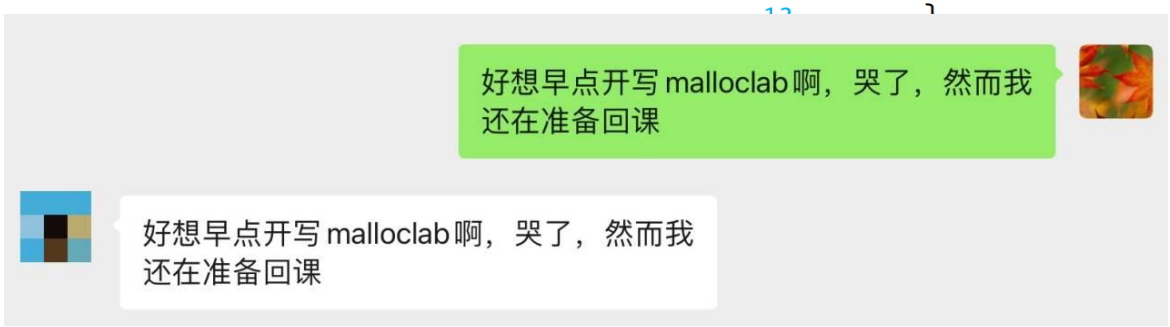
Server echo function

- Read -> Print Msg -> Write

`rio_readlineb` `printf` `rio_writen`

```
3   void echo(int connfd)
4   {
5       size_t n;
6       char buf[MAXLINE];
7       rio_t rio;
8
9       Rio_readinitb(&rio, connfd);
10      while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
11          printf("server received %d bytes\n", (int)n);
12          Rio_writen(connfd, buf, n);
```

Example



Today…

- Sockets interface revisited
  - Wrappers
    - **`getaddrinfo`** and **`getnameinfo`**
    - **`open_clientfd`** and **`open_serverfd`**
  - Client / Server session: Reading & Writing
    - The echo server
- **HTTP**
  - Requests & Responses
  - Dynamic content: CGI
- The TINY web server

Web Basics

- HTT**P** (hypertext transfer **protocol**)
  - Client: Browser
  - Server: Web server
- HTM**L** (hypertext markup **language**)
  - Language enabling display of webpage content
- MIME type
  - Extension associated with content, e.g., **`text/html`**, **`image/png`**

Today…

- Sockets interface revisited
  - Wrappers
    - **`getaddrinfo`** and **`getnameinfo`**
    - **`open_clientfd`** and **`open_serverfd`**
  - Client / Server session: Reading & Writing
    - The echo server
- HTTP
  - **Requests & Responses**
  - Dynamic content: CGI
- The TINY web server

HTTP Requests

- 1. Request line (location of resource)
  - *<method> <URI> <version>*
    - e.g. `GET / HTTP/1.1`
    - e.g. `GET /path/targetpage.html HTTP/1.0`
- 2. Request headers (info related to request, >=0)
  - *<header-name>: <header data>*
    - e.g. `Host: www.pku.edu.cn`
    - (Host is required for HTTP/1.1)
- 3. Request body (doesn't exist for HTTP GET)

HTTP Responses

- 1. Response line (outcome of response)
  - *<version> <status code> <status message>*
    - e.g. `HTTP/1.0 200 OK`
    - e.g. `HTTP/1.0 404 Not Found`
- 2. Response header (additional info, >=0)
  - *<header-name>: <header data>*
    - e.g. `Content-Type: text/html`
    - `Content-Length: 42092`

  Blank line (`"\r\n"`)
- 3. Content

HTTP status codes.
src: https://http.cat/

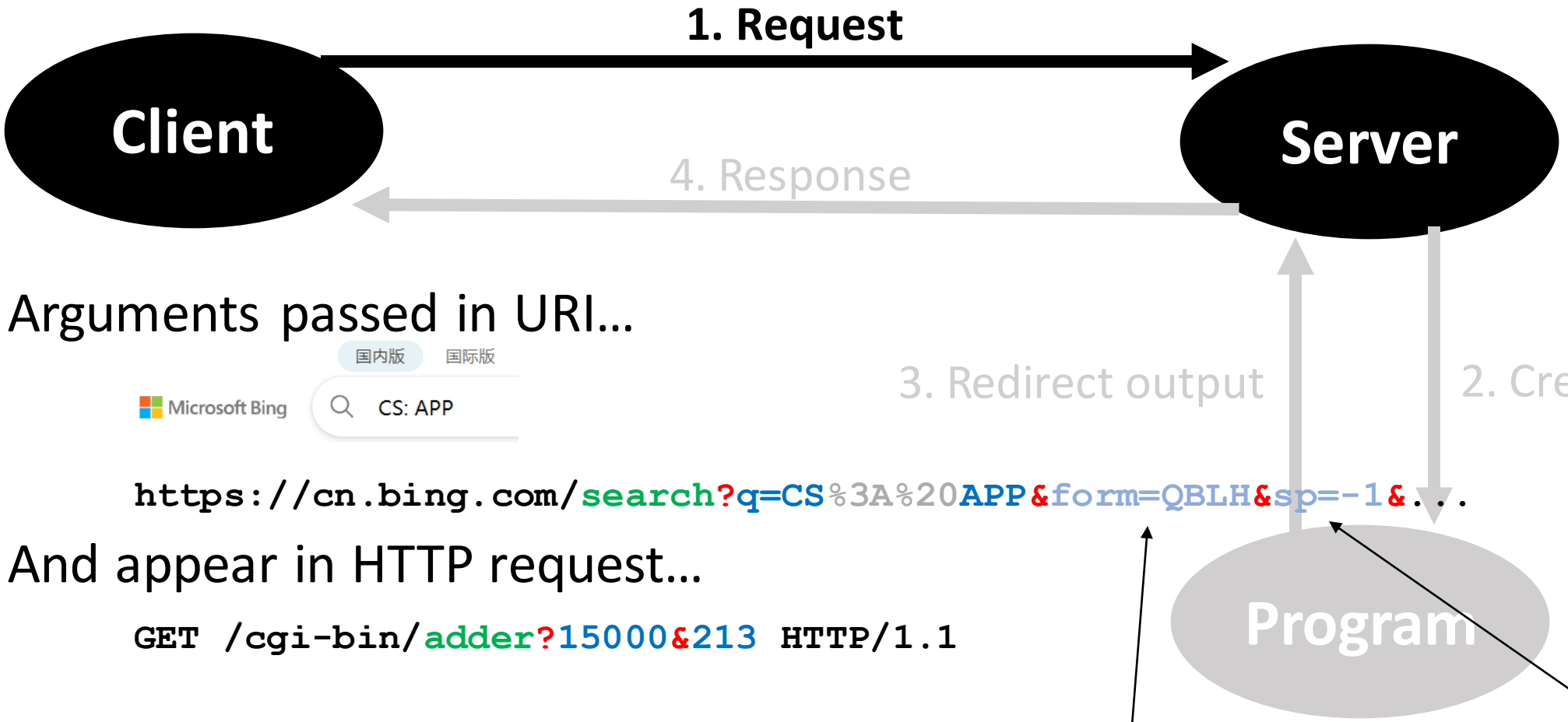
URL Parsing

- URL (universal resource locator)
  - **`http://www.google.com:80/index.html`**
  - **protocol://server:port/suffix**
    - **Suffix / is NOT root directory!**
      - **It gets expanded into default, (usually /index.html)**
    - This is why simply “www.google.com” works as well
- Static Content
  - Fetch a disk file
  - Return its contents
- Dynamic Content
  - Run an executable file (one (old) convention is that they are in `/cgi-bin/`)
  - return its output

Today…

- Sockets interface revisited
  - Wrappers
    - **`getaddrinfo`** and **`getnameinfo`**
    - **`open_clientfd`** and **`open_serverfd`**
  - Client / Server session: Reading & Writing
    - The echo server
- HTTP
  - Requests & Responses
  - **Dynamic content: CGI**
- The TINY web server

Dynamic Content: CGI



- Arguments passed in URI…

- And appear in HTTP request…

```
GET /cgi-bin/adder?15000&213 HTTP/1.1
```

Case: **PKU** Lecture Recordings

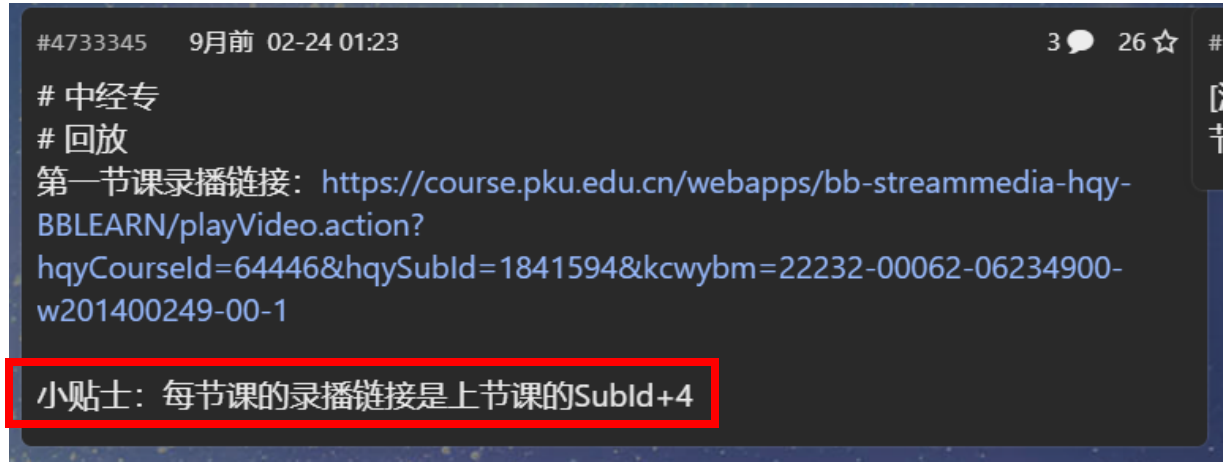
课程ID　　Subsection ID(?)　　课程唯一编码(?)

.../playVideo.action?hqyCourseId=78706&hqySubId=2521394&kcwybm=23241...
计算机系统导论　陈向群　北京大学本部一教-101　(2023/9/11 13:00)

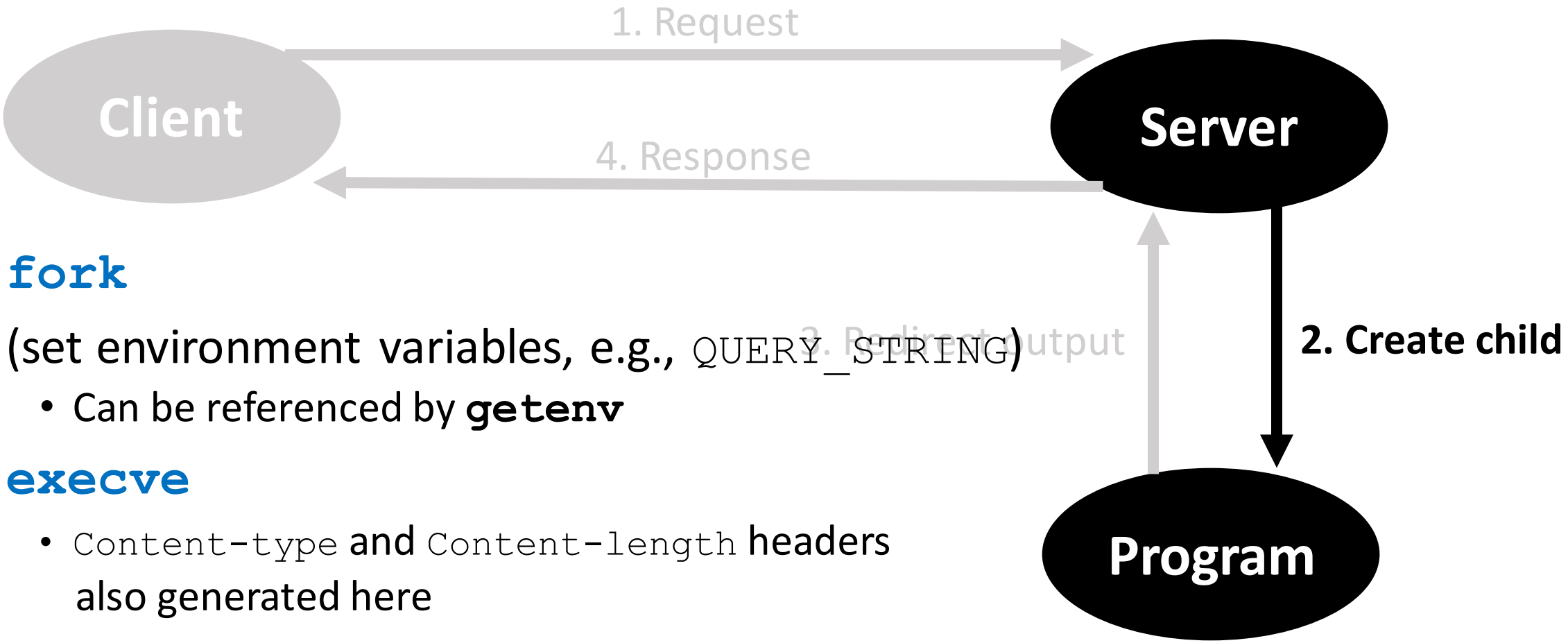
.../playVideo.action?hqyCourseId=78706&hqySubId=2521458&kcwybm=23241...
计算机系统导论　陈向群　北京大学本部一教-101　(2023/9/13 13:00)

.../playVideo.action?hqyCourseId=78706&hqySubId=2521398&kcwybm=23241...
计算机系统导论　陈向群　北京大学本部一教-101　(2023/9/18 13:00)

.../playVideo.action?hqyCourseId=78706&hqySubId=2521462&kcwybm=23241...
计算机系统导论　陈向群　北京大学本部一教-101　(2023/9/20 13:00)

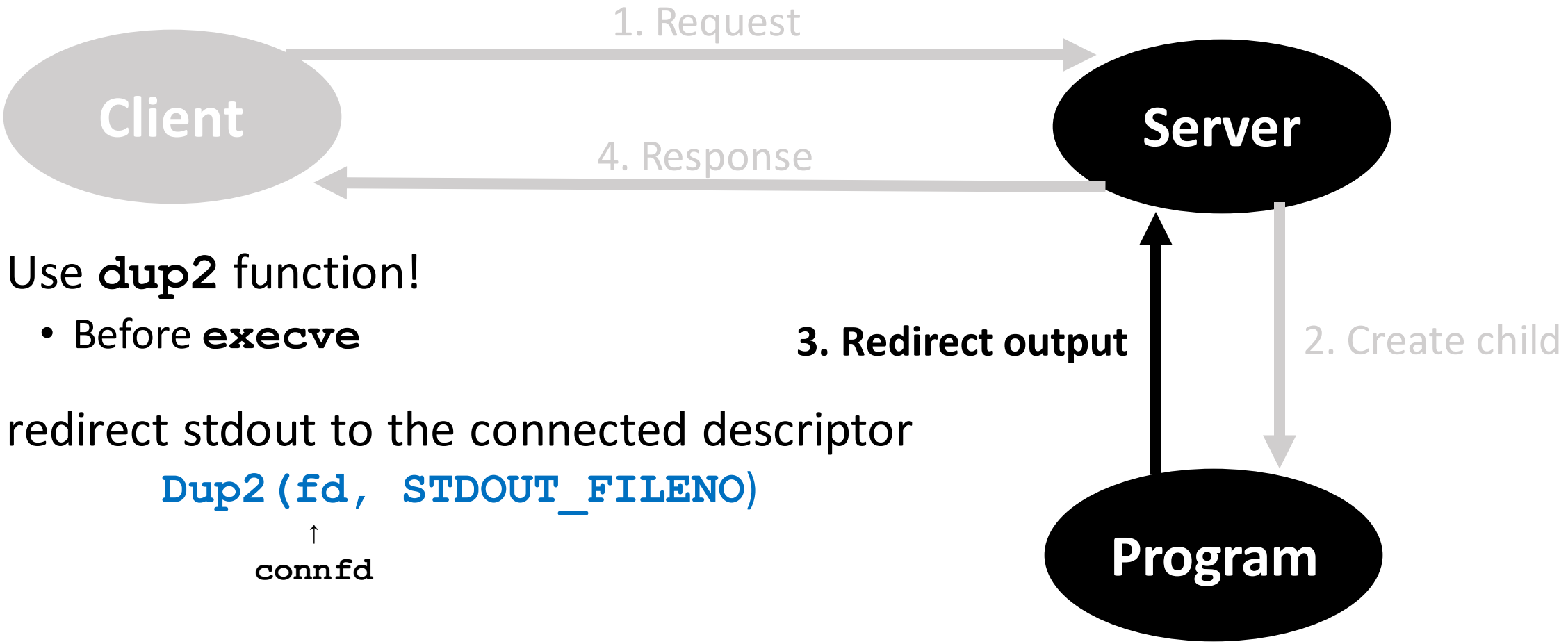


Dynamic Content: CGI



- **`fork`**
- (set environment variables, e.g., `QUERY_STRING`)
  - Can be referenced by **`getenv`**
- **`execve`**
  - `Content-type` and `Content-length` headers also generated here

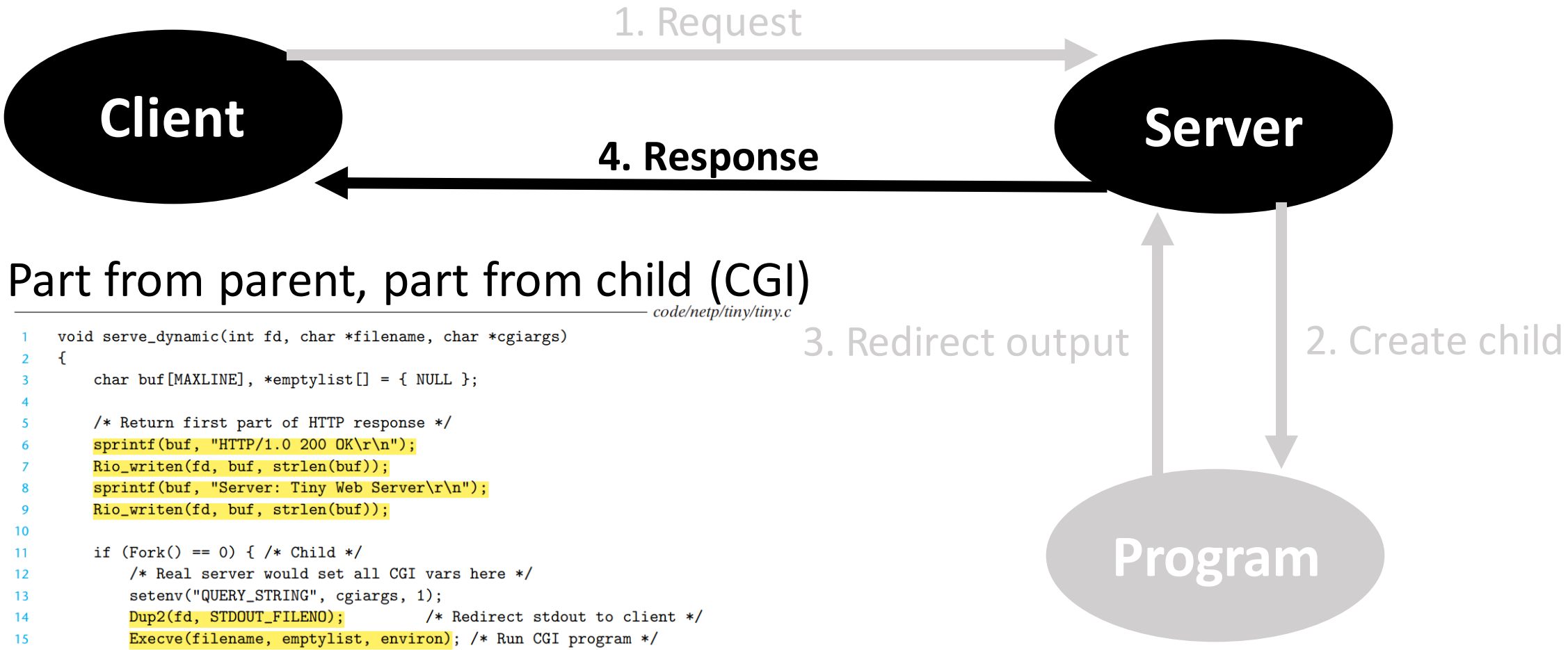
Dynamic Content: CGI



- Use **dup2** function!
  - Before **execve**
- redirect stdout to the connected descriptor

```
Dup2(fd, STDOUT_FILENO)
      ↑
    connfd
```

Dynamic Content: CGI



- Part from parent, part from child (CGI)

code/netp/tiny/tiny.c

```
1  void serve_dynamic(int fd, char *filename, char *cgiargs)
2  {
3      char buf[MAXLINE], *emptylist[] = { NULL };
4
5      /* Return first part of HTTP response */
6      sprintf(buf, "HTTP/1.0 200 OK\r\n");
7      Rio_writen(fd, buf, strlen(buf));
8      sprintf(buf, "Server: Tiny Web Server\r\n");
9      Rio_writen(fd, buf, strlen(buf));
10
11     if (Fork() == 0) { /* Child */
12         /* Real server would set all CGI vars here */
13         setenv("QUERY_STRING", cgiargs, 1);
14         Dup2(fd, STDOUT_FILENO);         /* Redirect stdout to client */
15         Execve(filename, emptylist, environ); /* Run CGI program */
16     }
17     Wait(NULL); /* Parent waits for and reaps child */
18 }
```

code/netp/tiny/tiny.c

Today…

- Sockets interface revisited
  - Wrappers
    - **`getaddrinfo`** and **`getnameinfo`**
    - **`open_clientfd`** and **`open_serverfd`**
  - Client / Server session: Reading & Writing
    - The echo server
- HTTP
  - Requests & Responses
  - Dynamic content: CGI
- **The TINY web server**

The TINY web server

```c
int main(...)
{
    /* Check command line */
    ...

    listenfd = Open_listenfd(port);
    while(1) {
        connfd = Accept(listenfd, ...);
        Getnameinfo(...);
        printf("Accepted connection");

        doit(connfd);
        Close(connfd);
    }
}
```

```c
void doit(int fd)
{
    read_requesthdrs(...);

    is_static = parse_uri(...);

    if (is_static) { /* Serve static content */
        serve_static(fd, ...);
    }
    else { /* Serve dynamic content */
        serve_dynamic(fd, ...);
    }
}
```

The TINY web server

```c
void doit(int fd)
{
    read_requesthdrs(...);

    is_static = parse_uri(...);

    if (is_static) { /* Serve static content */
        serve_static(fd, ...);
    }
    else { /* Serve dynamic content */
        serve_dynamic(fd, ...);
    }
}
```

```c
void serve_static(int fd, ...)
{
    /* Send response headers */
    ...

    /* Send response body */
    ...
}
```

```c
void serve_dynamic(int fd, ...)
{
    /* Send first part of response */
    Rio_writen(fd, ...);

    if (Fork() == 0) { /* Child*/
        setenv("QUERY_STRING", ...);
        Dup2(fd, STDOUT_FILENO);    /* Redirect stdout */
        Execve(...)                 /* Run CGI */
    }
    Wait(NULL); /* Parent waits for child */
}
```

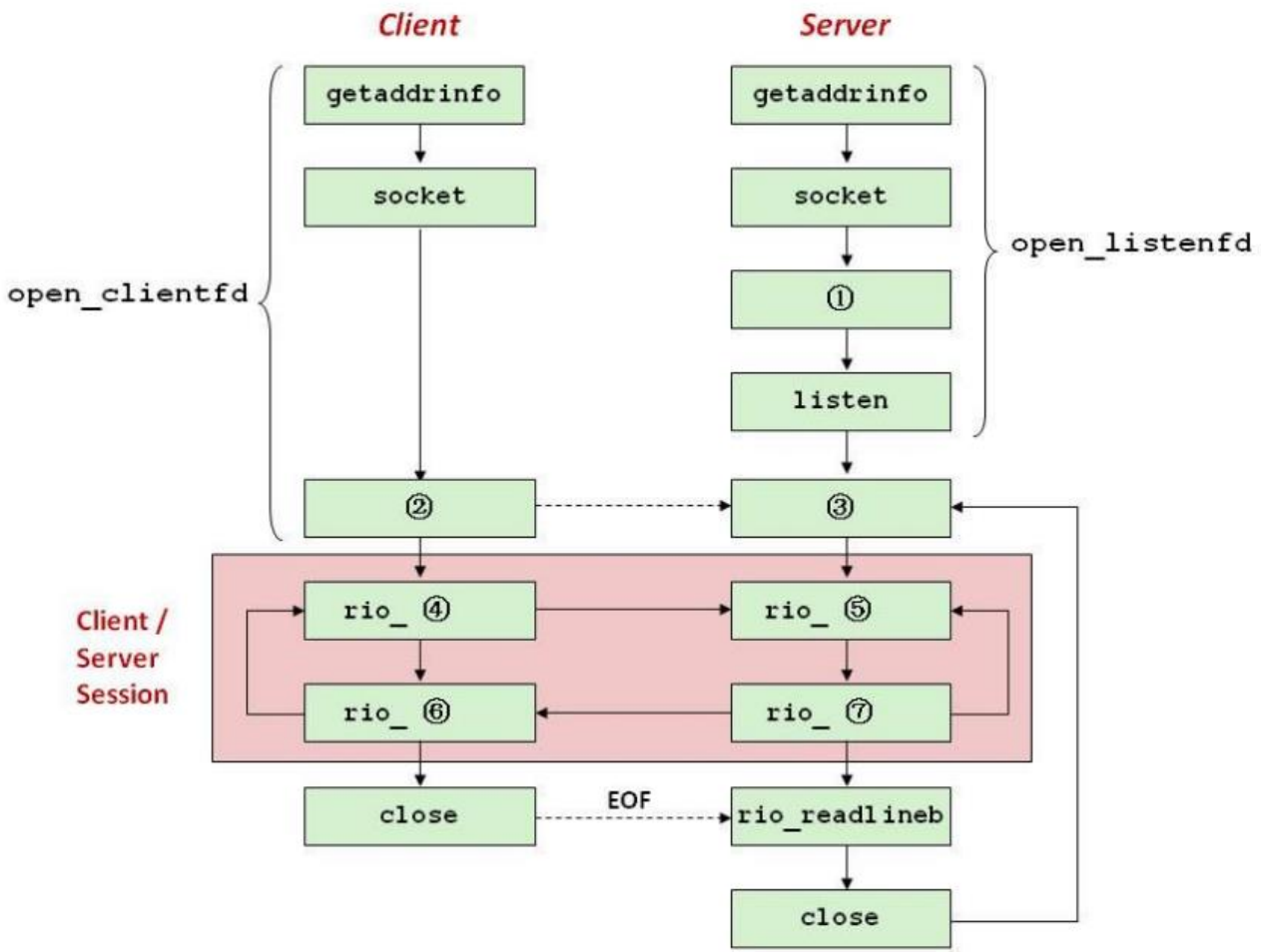
Practice

贾博暄

第六题（10 分）

下图是一个基于 echo 服务器的 client-server 框架

（1） 请给图中的编号填写相应的函数名。



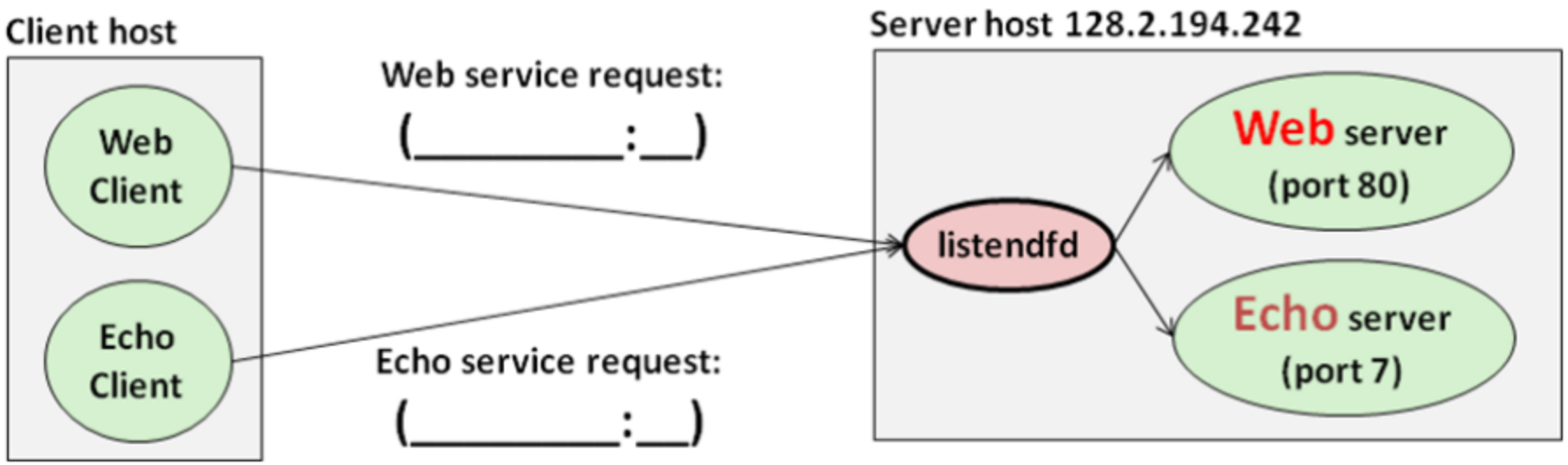
Source: 2019fa

（2） 请补全下面 server 端 open_listenfd 函数中缺失的操作（Line 21, Line 26 和 Line 34）

```
Line 1:  int open_listenfd(char *port)
Line 2:  {
Line 3:     struct addrinfo hints, *listp, *p;
Line 4:     int listenfd, optval=1;
Line 5:     /* Get a list of potential server addresses */
Line 6:     memset(&hints, 0, sizeof(struct addrinfo));
Line 7:     hints.ai_socktype = SOCK_STREAM;
Line 8:     hints.ai_flags = AI_PASSIVE | AI_ADDRCONFIG;
Line 9:     hints.ai_flags |= AI_NUMERICSERV;
Line 10:    Getaddrinfo(NULL, port, &hints, &listp);
Line 11:
Line 12:     for (p = listp; p; p = p->ai_next) {
Line 13:        /* Create a socket descriptor */
Line 14:       if ((listenfd = socket(p->ai_family, p->ai_socktype,
Line 15:                              p->ai_protocol)) < 0)
Line 16:         continue;  /* Socket failed, try the next */
Line 17:        /* Eliminates "Address already in use" error from
bind */
Line 18:       Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
Line 19:                  (const void *)&optval , sizeof(int));
Line 20:
Line 21:       if (_⑧_(listenfd, p->ai_addr, p->ai_addrlen) == 0)
Line 22:         break;  /* Success */
Line 23:       Close(listenfd);
Line 24:    }
Line 25:    /* Clean up */
Line 26:    Freeaddrinfo(___⑨___);
Line 27:    if (!p) /* No address worked */
Line 28:       return -1;
Line 29:
Line 30:    if (listen(listenfd, LISTENQ) < 0) {
Line 31:       Close(listenfd);
Line 32:       return -1;
            }
            return __⑩___;
         }
```

Source: 2019fa

（3） 在下图中连线上填入正确的目标服务器的 socket 标识符（2 分）



Source: 2013fa

4）在 Echo server 范例中，server 端通过 accept 函数接受了一个 client 的连接请求，从而将网络描述符与该网络连接、socket 绑定，然后进行网络数据传输。在下面的空格处填写正确的网络描述符，每个空填写 **listenfd** 或 **connfd**（共 4 分，每空 1 分）。

```
int main(int argc, char **argv) {
    int listenfd, connfd, port, clientlen;
    struct sockaddr_in clientaddr;
    struct hostent *hp;
    char *haddrp;
    unsigned short client_port;
      ……
      while (1) {
          clientlen = sizeof(clientaddr);
          _________ = Accept(_________, (SA *)&clientaddr,
                      &clientlen);
          hp = Gethostbyaddr((const char*)
               &clientaddr.sin_addr.s_addr,
               sizeof(clientaddr.sin_addr.s_addr), AF_INET);
          haddrp = inet_ntoa(clientaddr.sin_addr);
          client_port = ntohs(clientaddr.sin_port);
          printf("server connected to %s (%s), port %u\n",
                    hp->h_name, haddrp, client_port);
          echo(_________);
          Close(_________);
          }
      }
```

Source: 2013fa

The End