

Network Programming I & Network Programming II

侯旭森 贾博暄 许珈铭

2023.12.11

Network Programming I

(CS:APP Ch. 11.1-11.4.6)

侯旭森

客户端-服务器编程模型

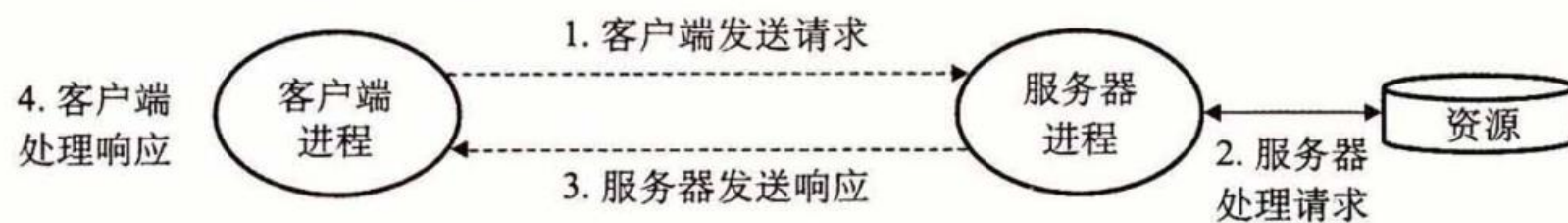
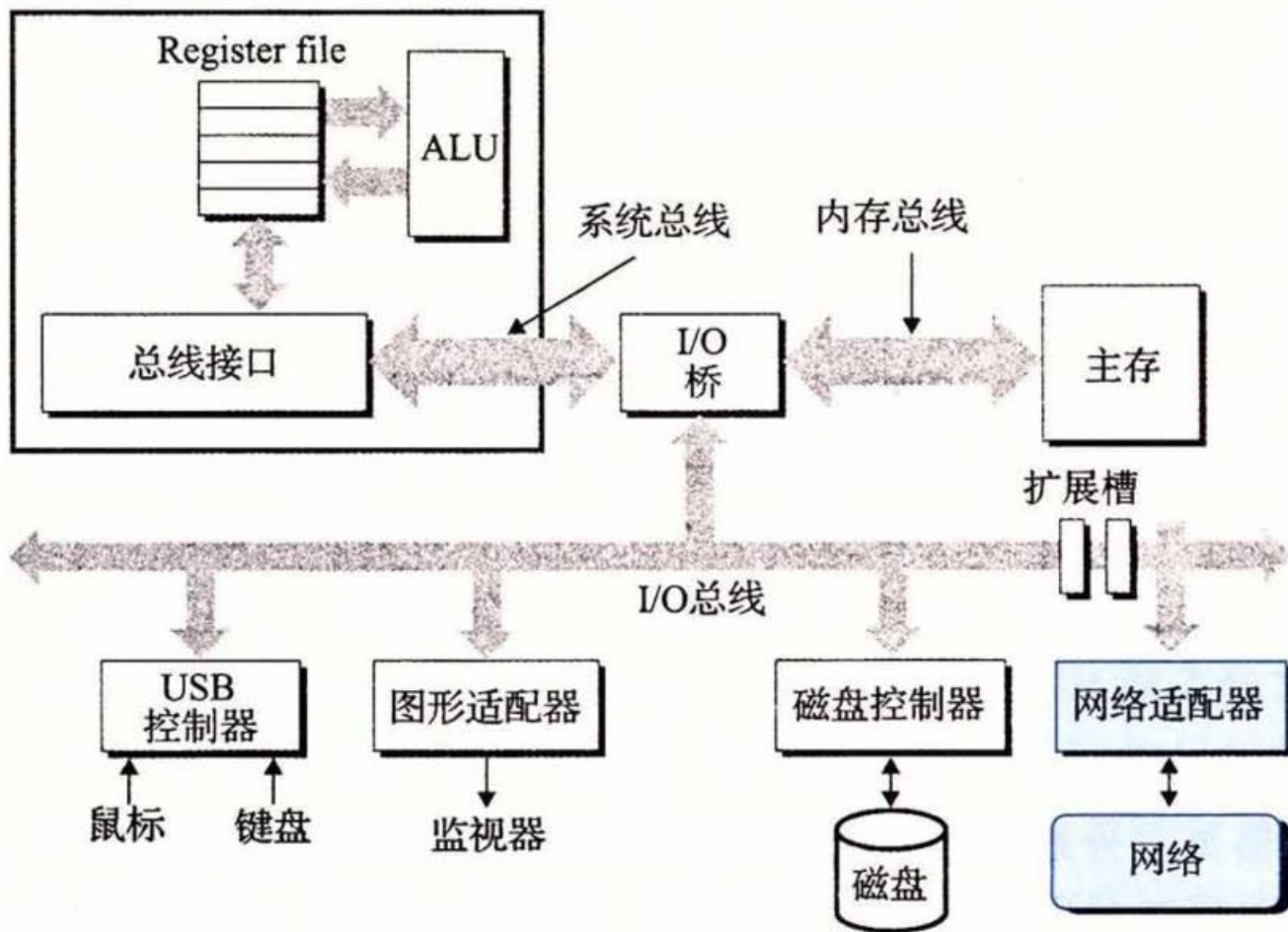


图 11-1 一个客户端-服务器事务

客户端和服务端是进程

网络

CPU 芯片



- 对主机而言，网络只是又一种I/O设备。
- 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)

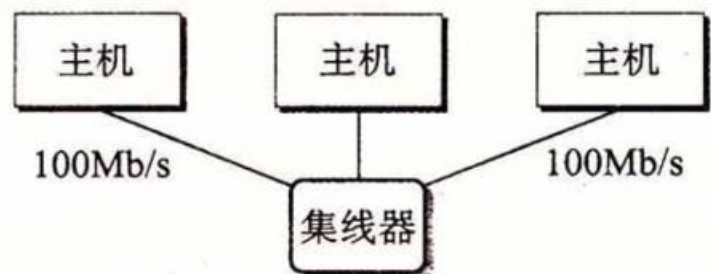
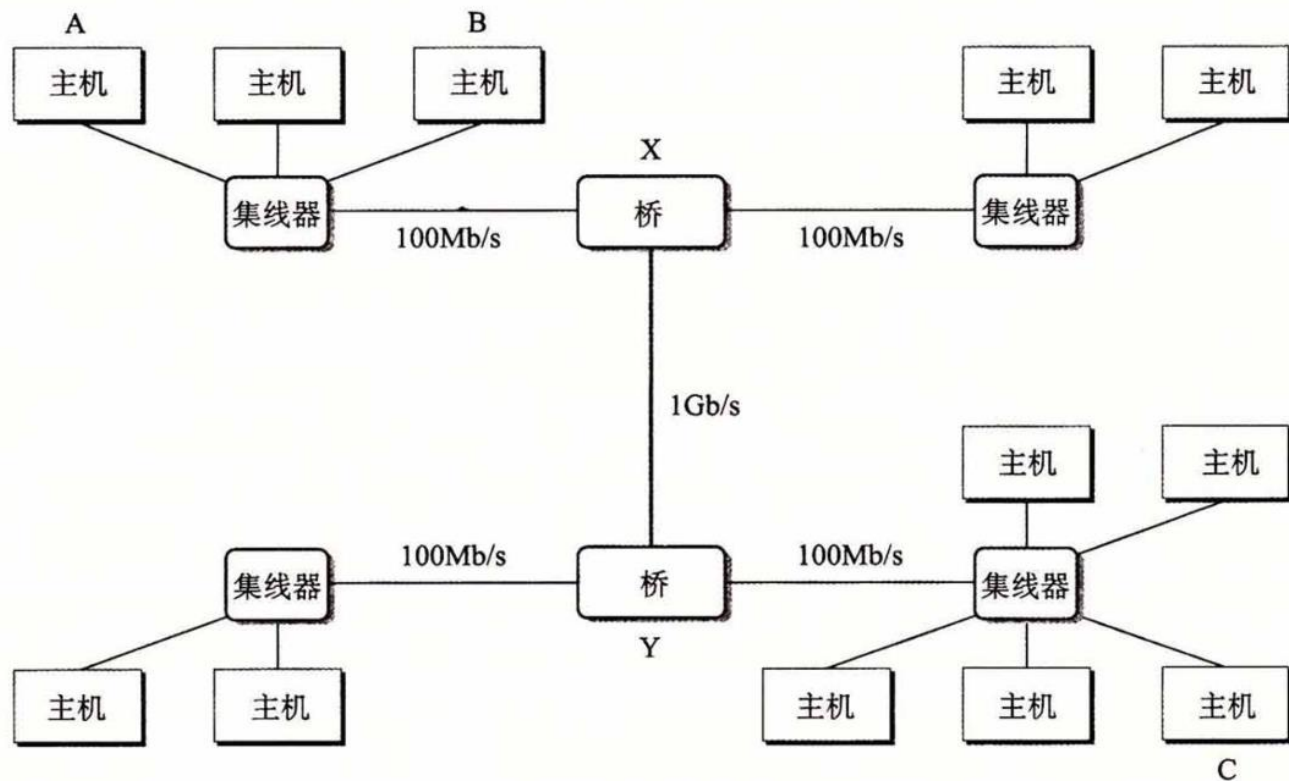


图 11-3 以太网段

- 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

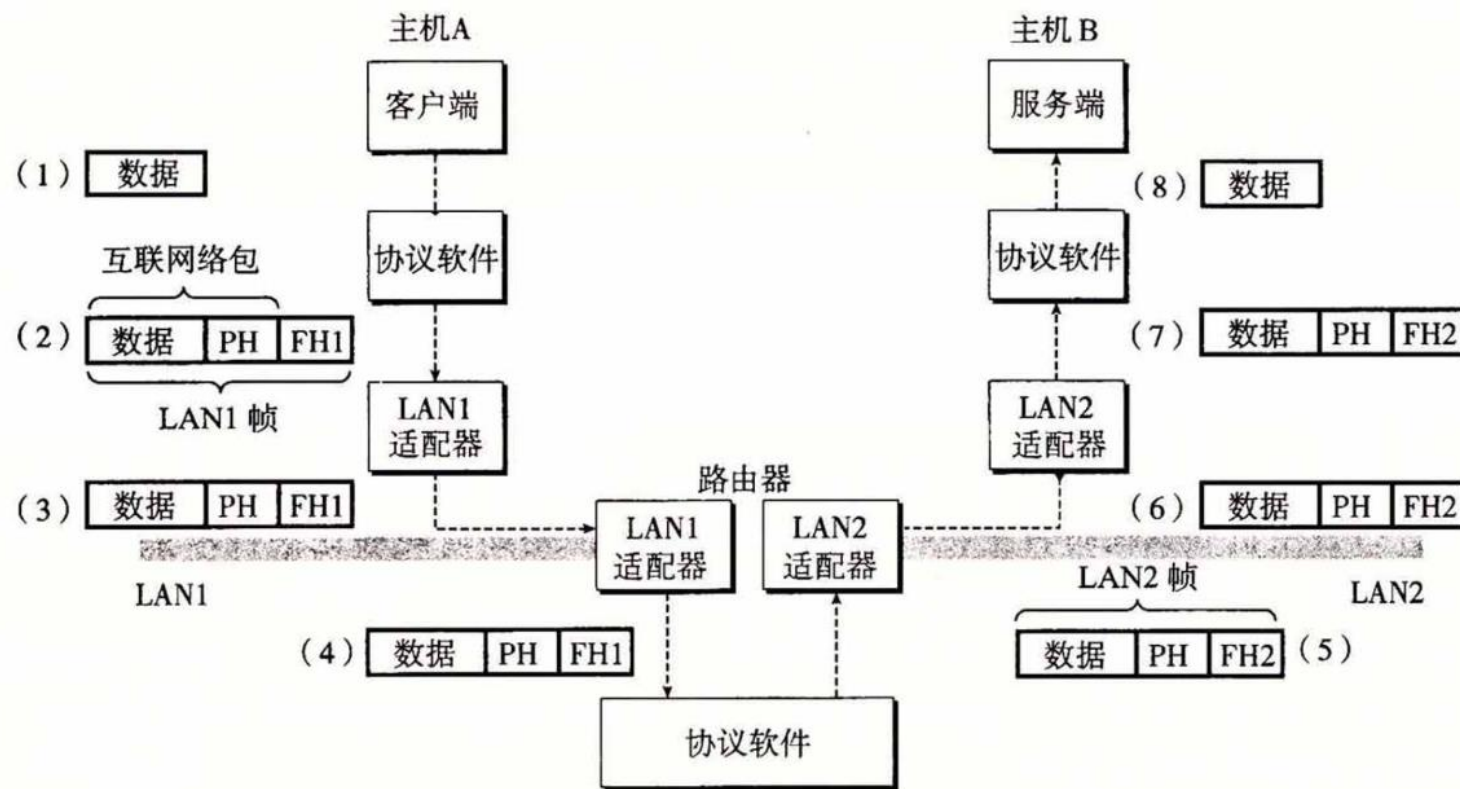
WAN(Wide-Area Network,广域网)



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

- 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);
- internet描述一般概念, 而Internet描述一种具体的实现——全球IP因特网。

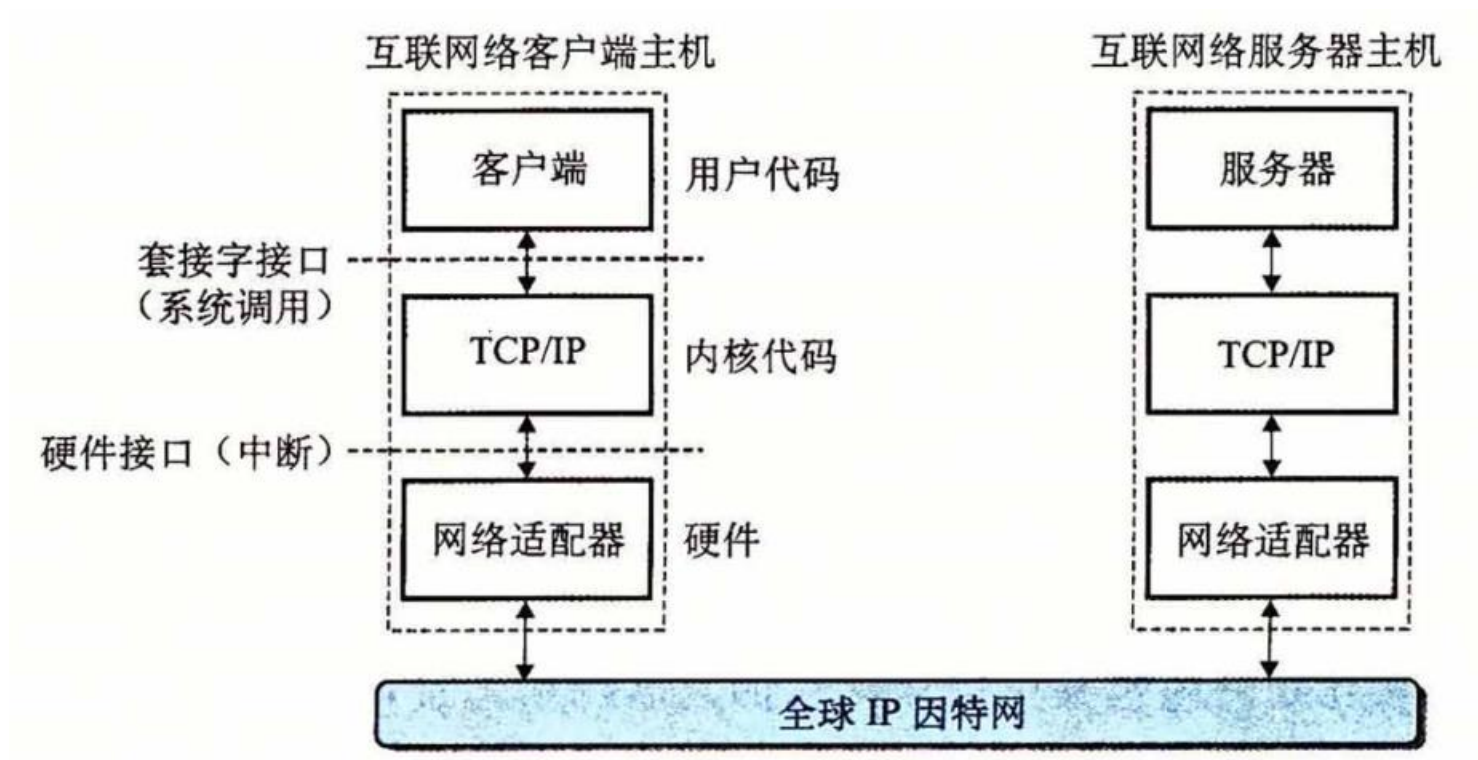
封装是关键



- 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主机B，LAN1帧头寻址到路由器。

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

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(uint16_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)



(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）

(2) 该服务器在所有有 web 应用端口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	$2 \times 2^{32+16}$

(2) 该服务器在所有有 web 应用端口上最多可以监听多少个独立的 socket 连接？（2 分）

答案： 2×2^{64}

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

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

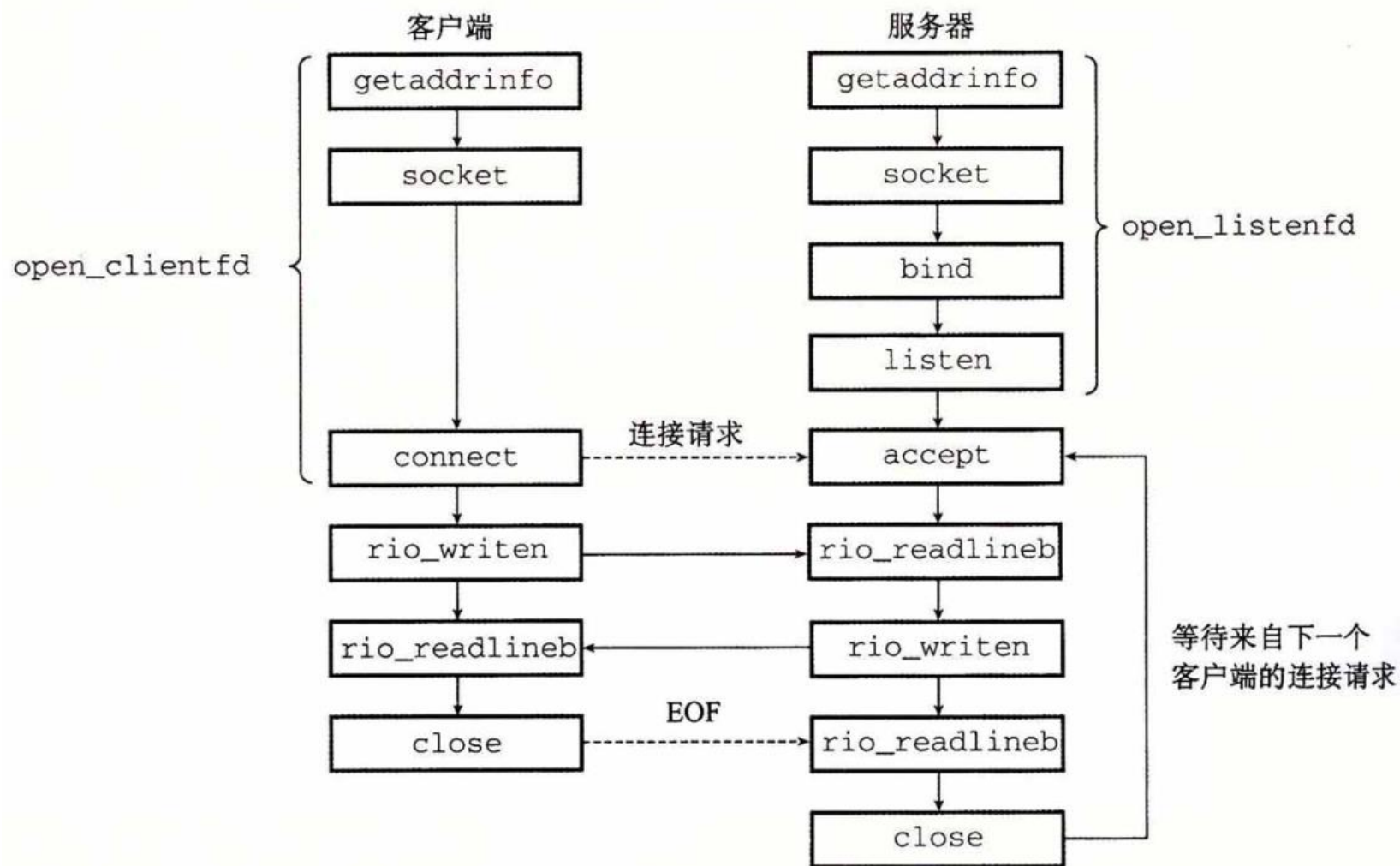


图 11-12 基于套接字接口的网络应用概述

- 主动套接字 (的描述符)
- 监听描述符
- 已连接描述符

套接字接口

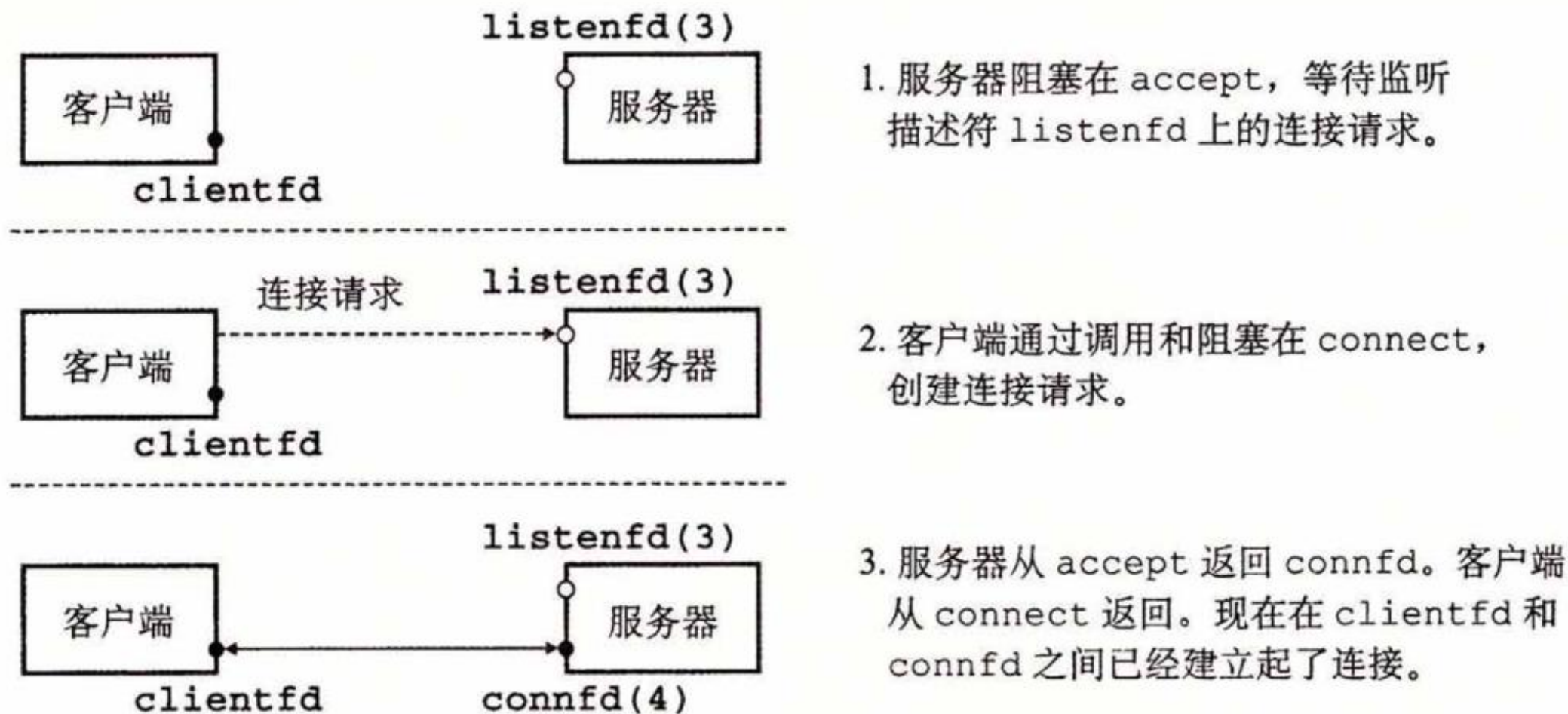


图 11-14 监听描述符和已连接描述符的角色

套接字接口

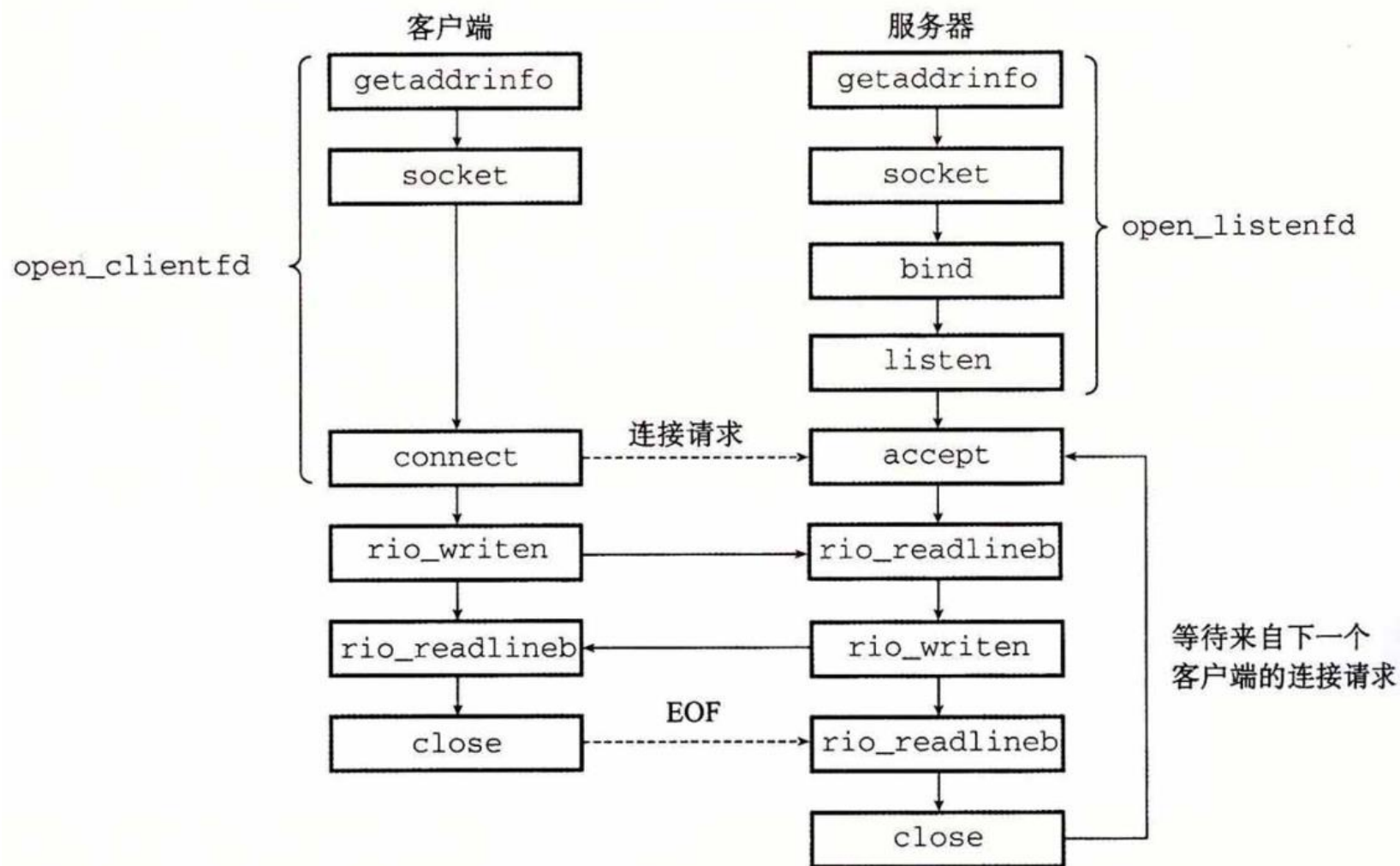
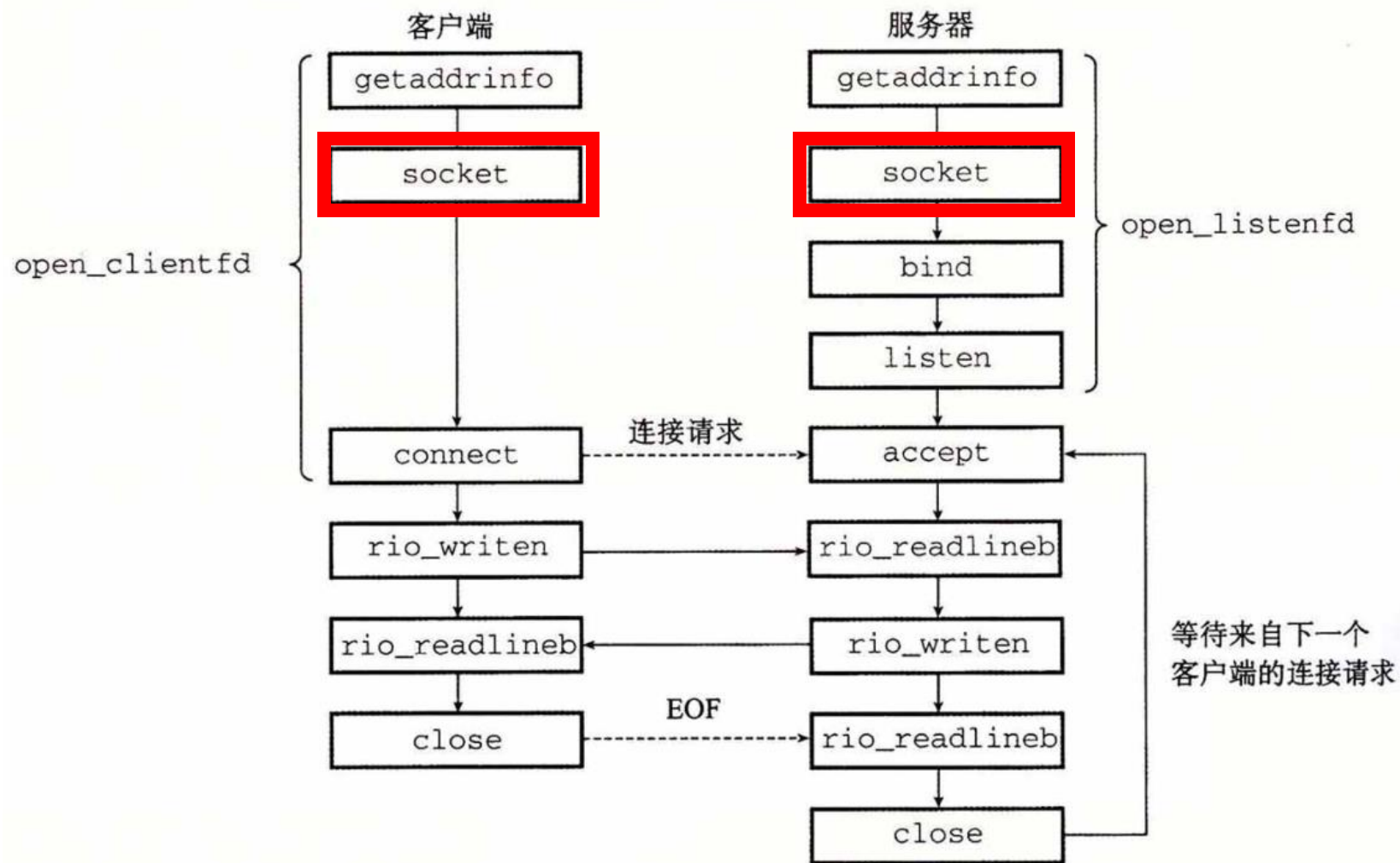


图 11-12 基于套接字接口的网络应用概述

- 主动套接字
- 监听套接字
- 已连接描述符

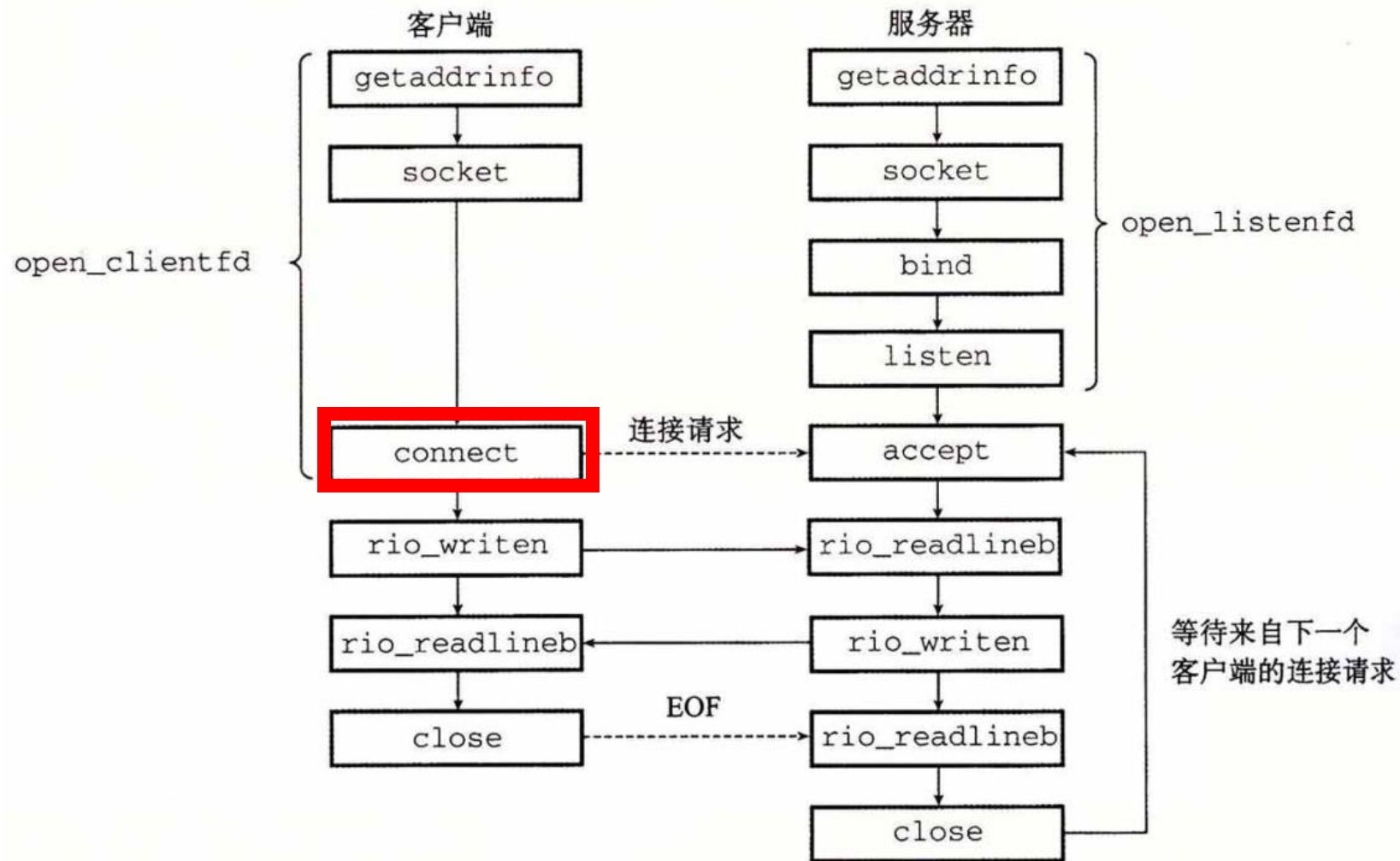


客户端和服务端使用 `socket` 函数来创建一个套接字描述符(socket descriptor)。

```
#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。



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。

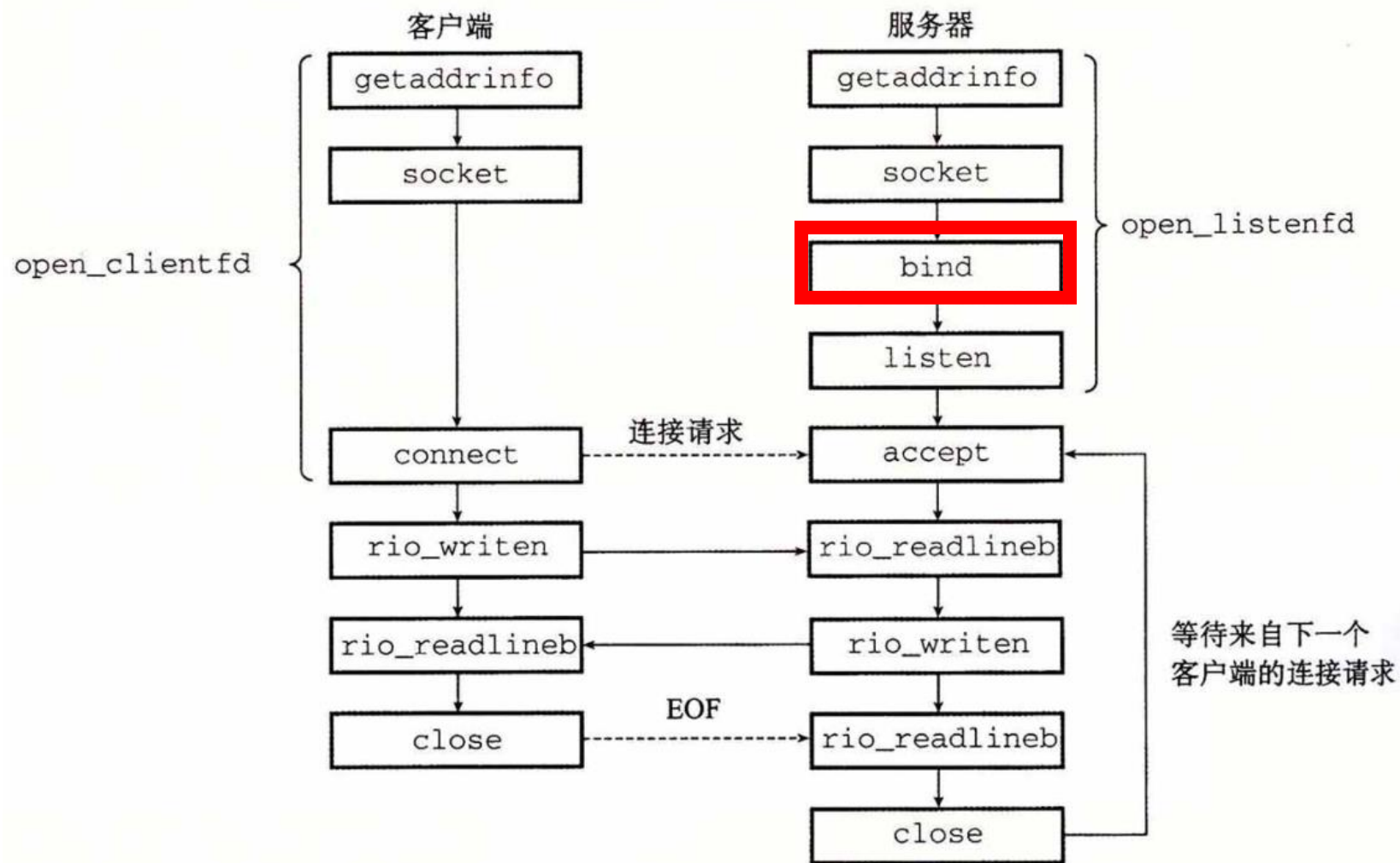


图 11-12 基于套接字接口的网络应用概述

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

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

返回：若成功则为 0，若出错则为 -1。

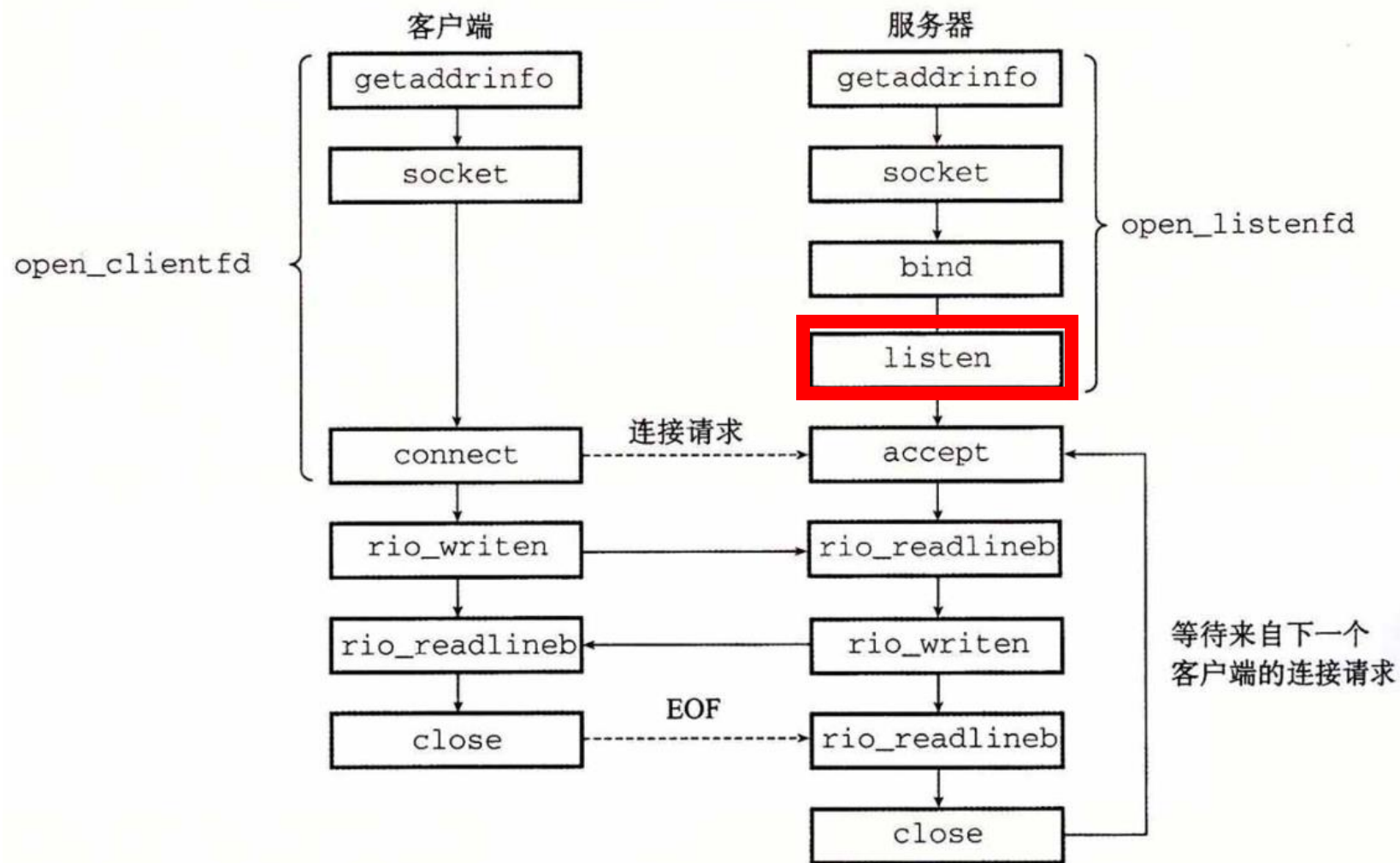


图 11-12 基于套接字接口的网络应用概述

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

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

返回：若成功则为 0，若出错则为 -1。

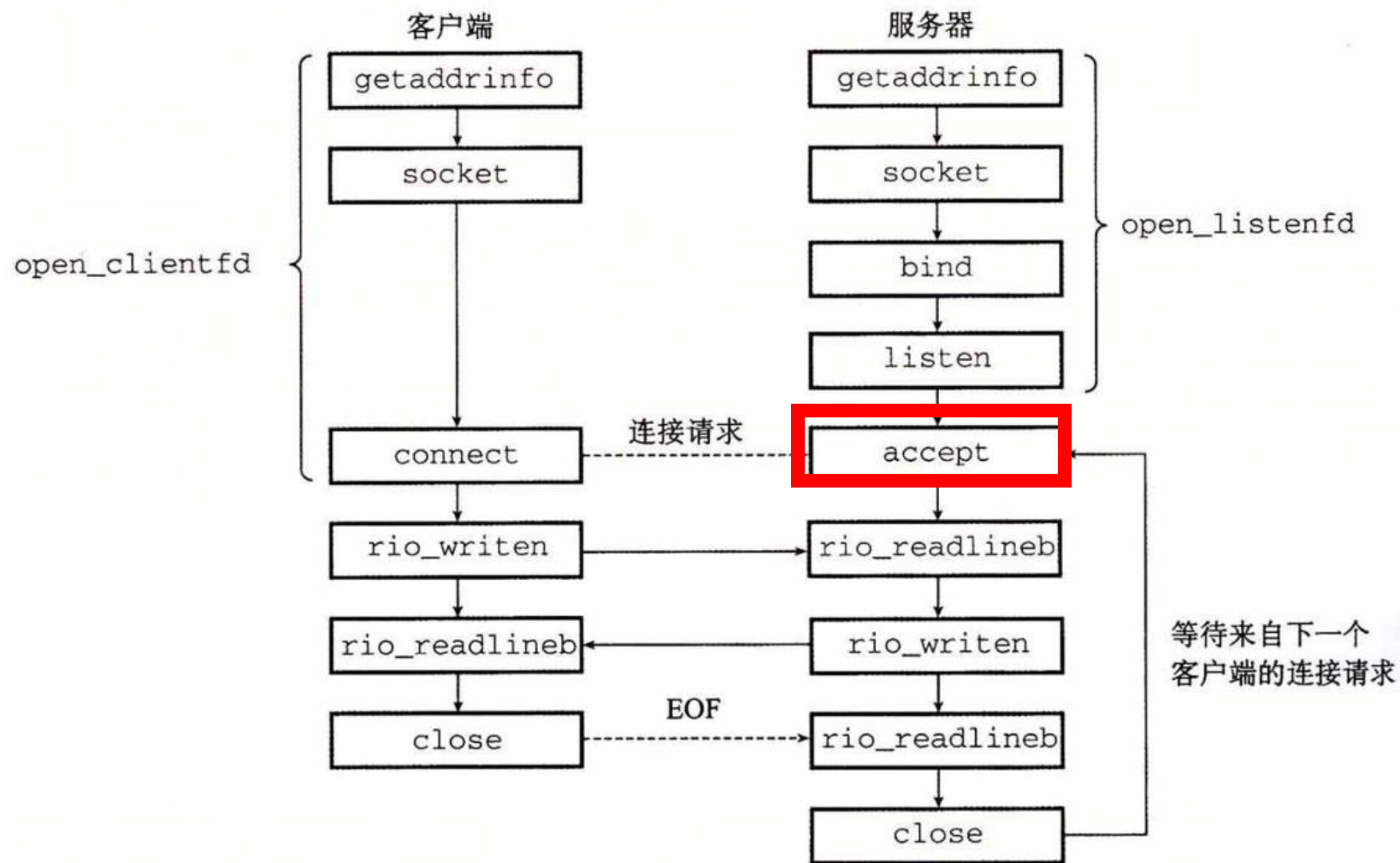


图 11-12 基于套接字接口的网络应用概述

```
#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
 - `open_clientfd` and `open_serverfd`
 - `getaddrinfo` and `getnameinfo`
- Client / Server session: Reading & Writing
 - The echo server

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

——CS:APP3e, Page 980

- **HTTP**

- Requests & Responses
- Dynamic content: CGI

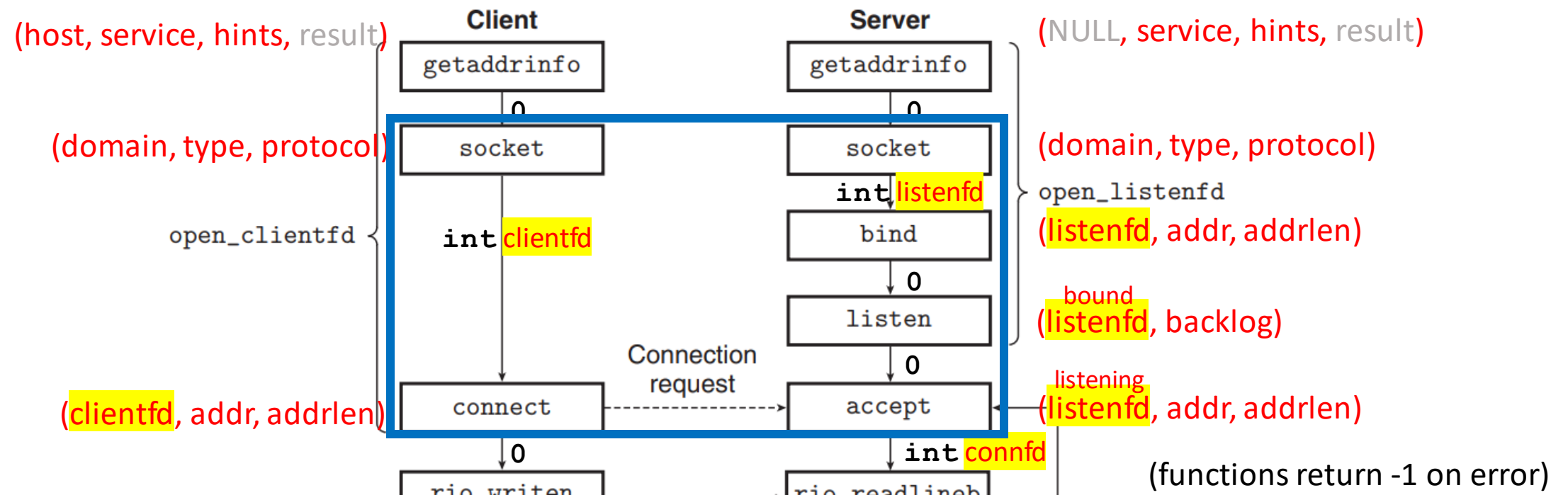
- **The TINY web server**

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
 - **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

But before we dive in...

- **getaddrinfo**
 - A “better” `inet_pton`

```
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

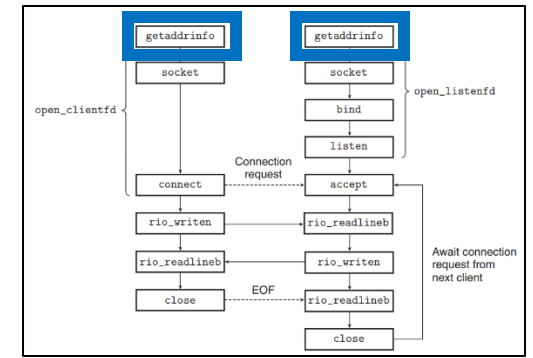
Recall:

p = presentation

(dotted decimal, e.g., 127.0.0.1)

n = network

(e.g., 0x7f000001)



But before we dive in...

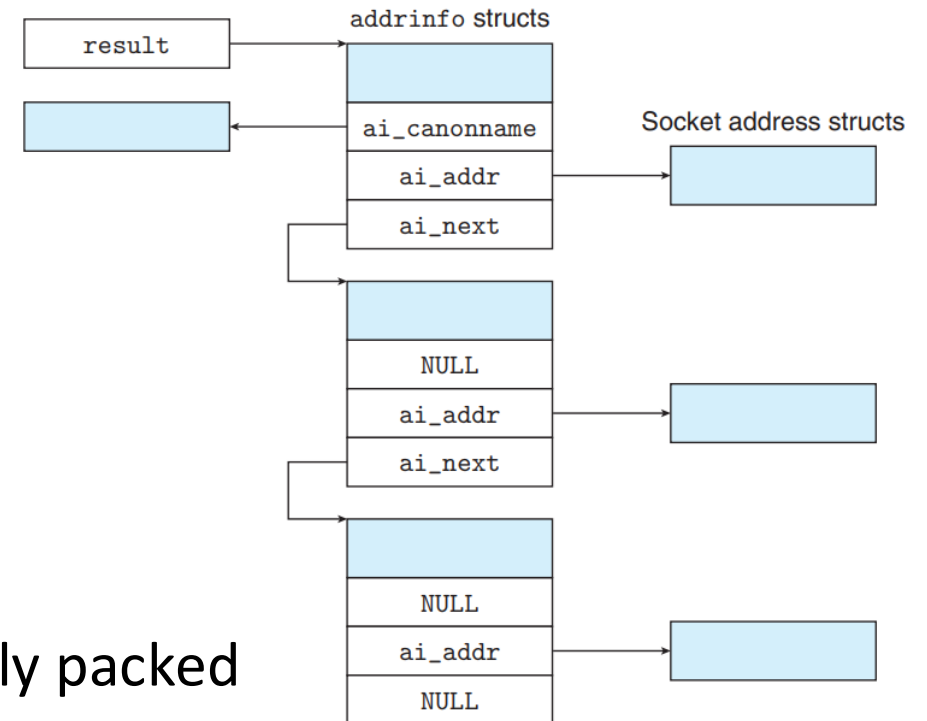
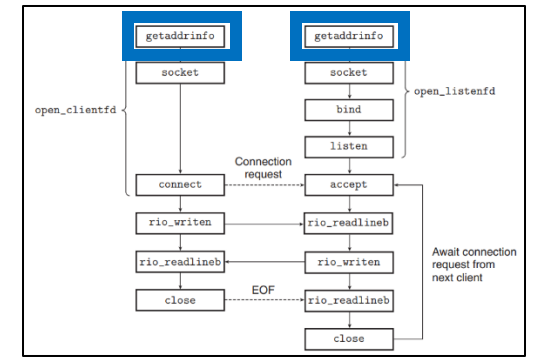
- The **addrinfo** struct

```
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

- Traverse for connection

- Args for **socket**, **connect**, **listen** are neatly packed



But before we dive in...

- **getnameinfo**

- A “better” `inet_ntop`

```
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

Recall:

p = presentation

(dotted decimal, e.g., 127.0.0.1)

n = network

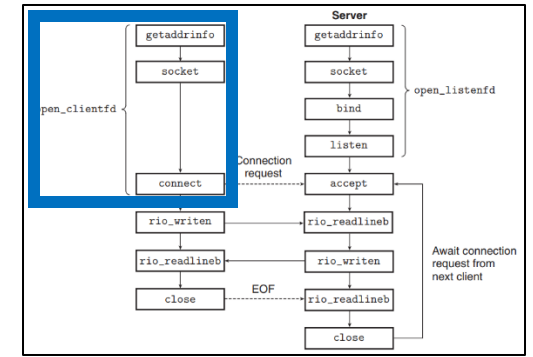
(e.g., 0x7f000001)

- Domain name in `host`
 - (can configure `FLAGS` to return numeric address string)
- Service name in `service`
 - (can configure `FLAGS` to return port number)
- (Used in `TINY` for printing statements)

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

open_clientfd: Motivation



- 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

In our case,
port number



(host, service, hints, result)

(domain, type, protocol)

open_clientfd

(clientfd, addr, addrlen)

Client

getaddrinfo

0

socket

int clientfd

connect

0

• So...

```
#include "csapp.h"
```

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

Returns: descriptor if OK, -1 on error

open_clientfd

code/src/csapp.c

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

```
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;
}
```

args are neatly packed
in **addrinfo** struct

code/src/csapp.c

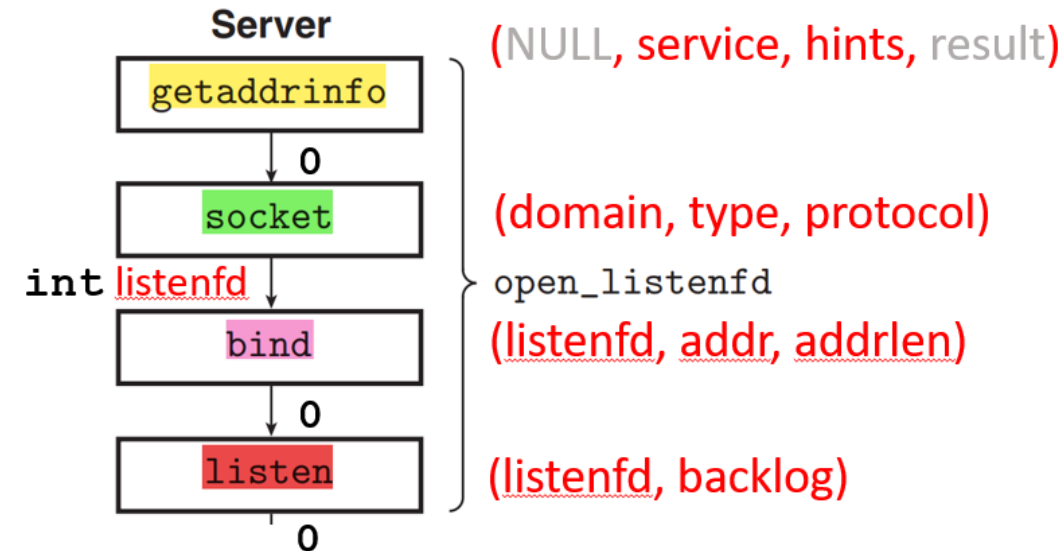
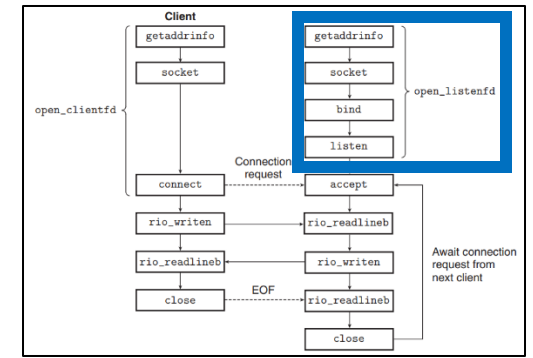
open_listenfd: Motivation

- 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
 - `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

Reading & Writing

- Recall 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

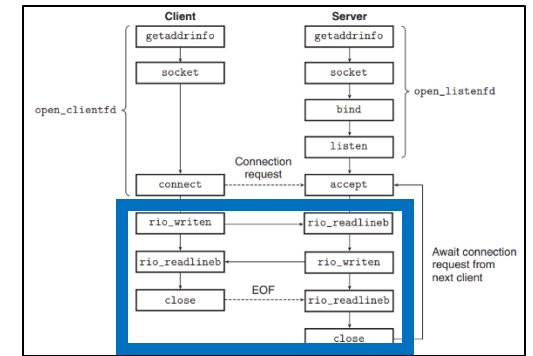
```
ssize_t rio_writen(int fd, void *usrbuf, size_t n);
```

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



Recall:

rio_readlineb

= read next text line,
buffered

writen = n bytes
(handles short counts)

- These are used by both clients and servers

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

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 }
```

Example



好想早点开写 malloc lab啊，哭了，然而我还在准备回课

好想早点开写 malloc lab啊，哭了，然而我还在准备回课

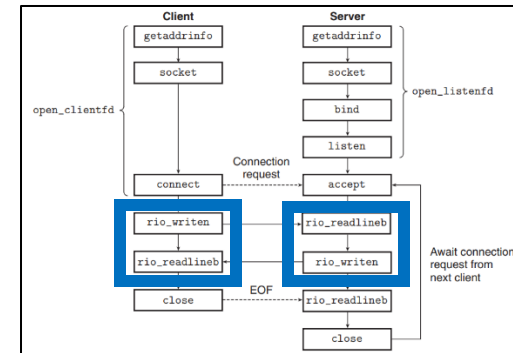


Server echo function

- 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);
13    }
```



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

Web Basics

- HTTP (hypertext transfer protocol)
 - Client: Browser
 - Server: Web server
- HTML (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
 - `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

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



200
OK



404
Not Found

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 / gets expanded into default, (usually /index.html)
 - This is why
- 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

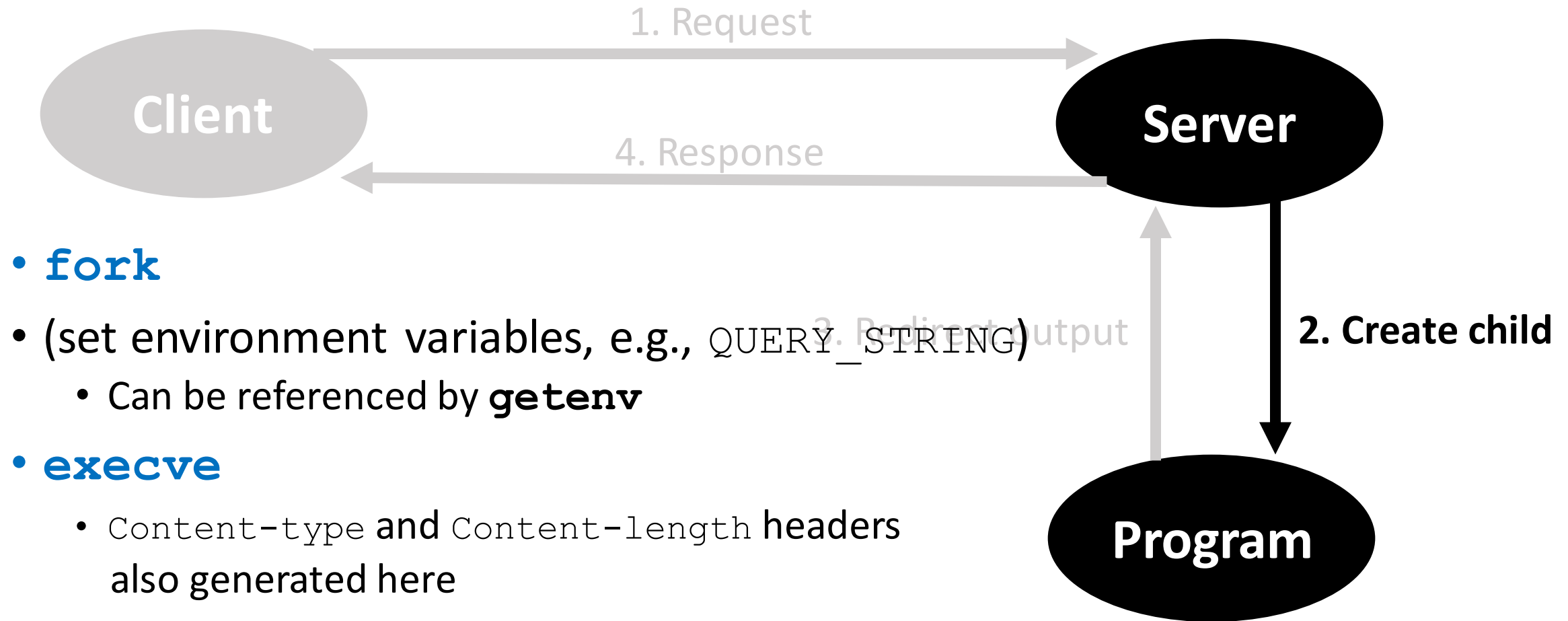
Today...

- Sockets interface revisited
 - Wrappers
 - `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

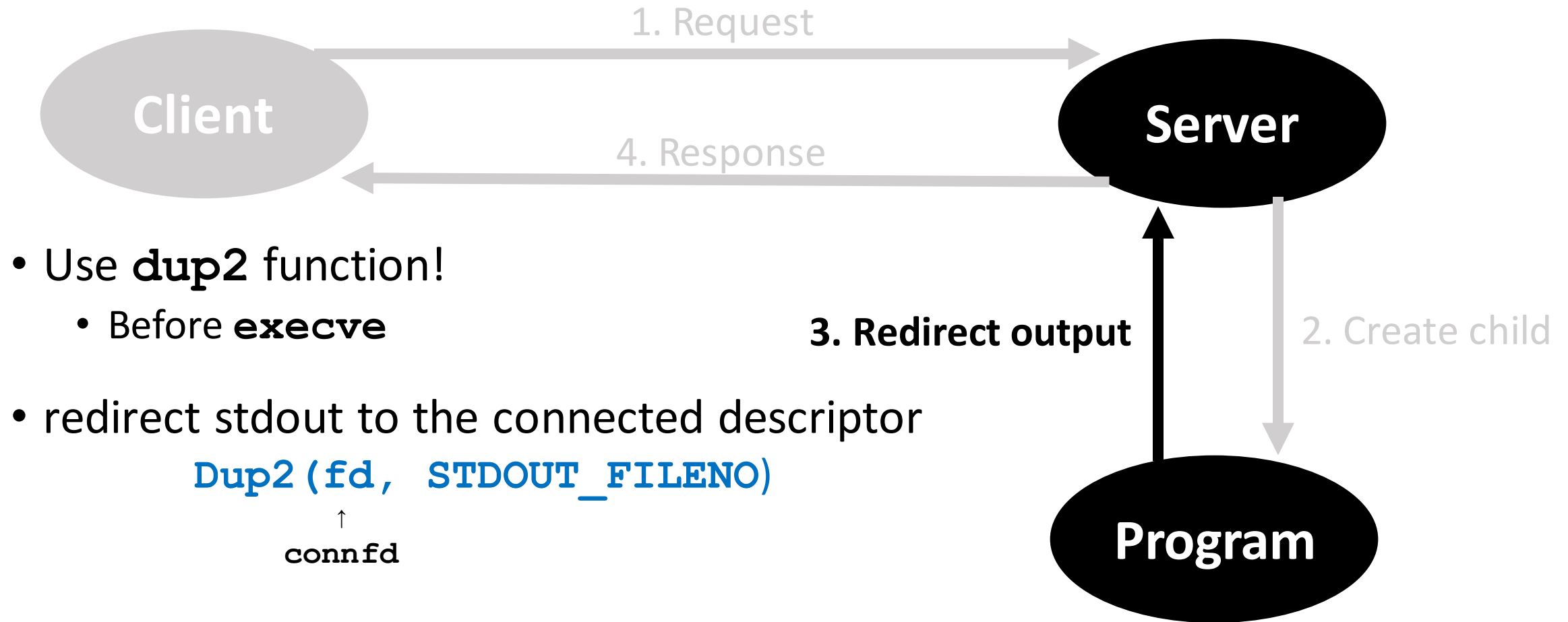
Dynamic Content: CGI



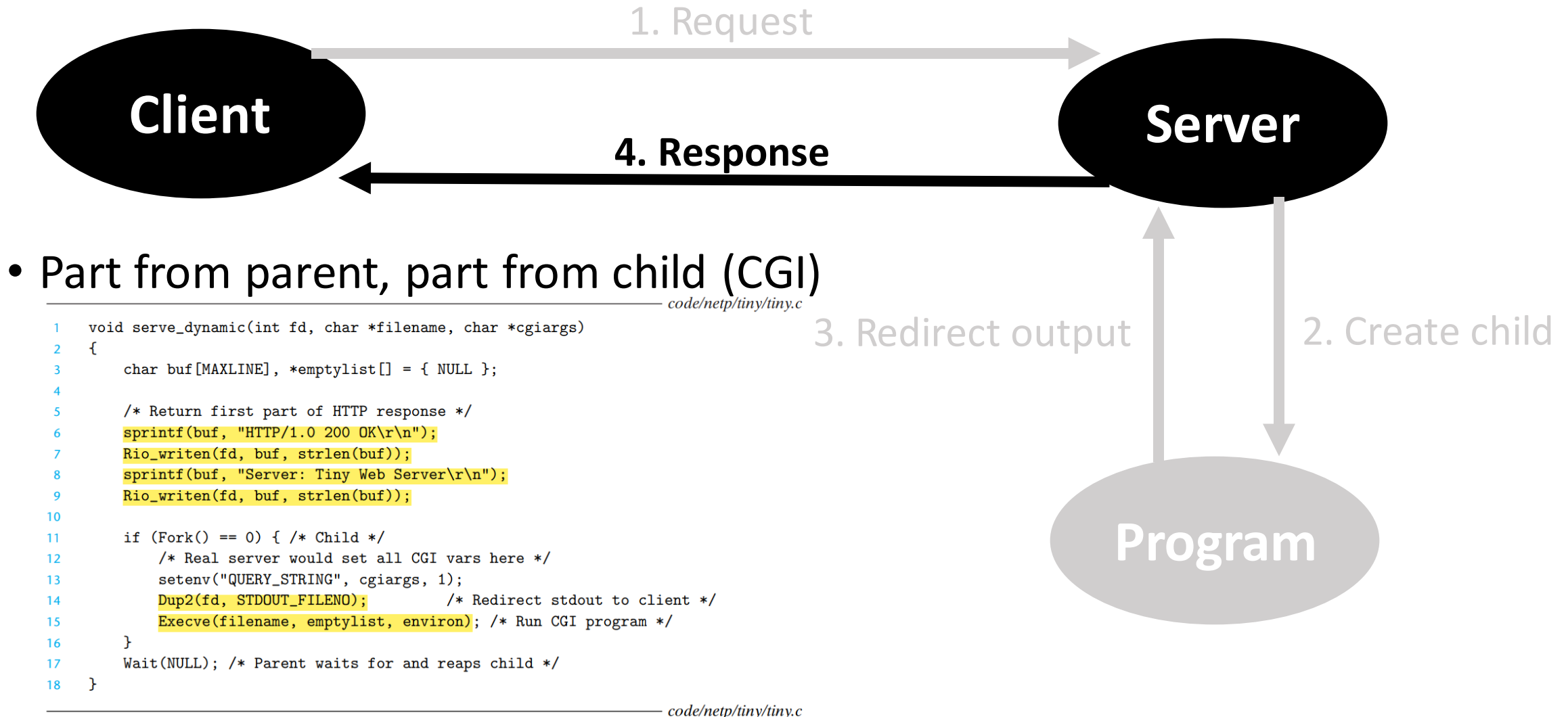
Dynamic Content: CGI



Dynamic Content: CGI



Dynamic Content: CGI



Today...

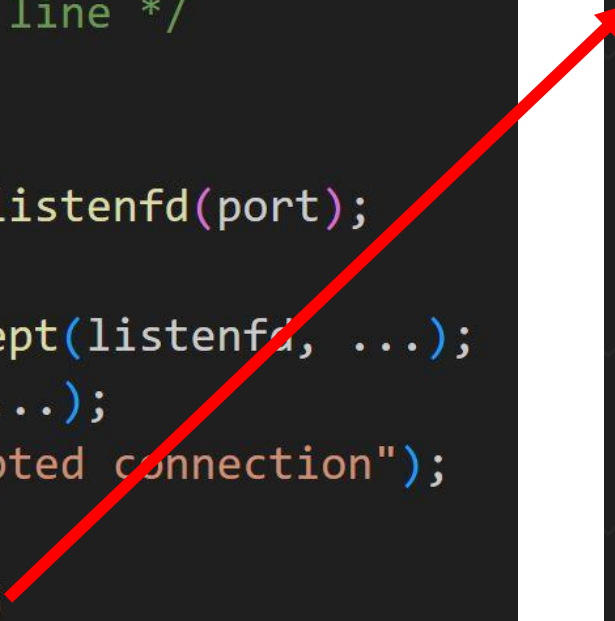
- Sockets interface revisited
 - Wrappers
 - `open_clientfd` and `open_serverfd`
 - `getaddrinfo` and `getnameinfo`
 - Client / Server session: Reading & Writing
 - The echo server
- HTTP
 - Requests & Responses
 - Dynamic content: CGI
- **The TINY web server**

The TINY web server

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

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

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



```
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

```
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, ...);
    }
}
```

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

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

```
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

The End