第11章 网络编程
Network Programming: Part II

100076202： 计算机系统导论

任课教师：
计卫星 宿红毅 张艳

原作者：
Randal E. **Bryant and** David R. O’Hallaron

议题

- 上次的问题 **Questions from yesterday**
- 上次没有讨论的内容 **Material we didn’t get to yesterday**
  - 使用套接字发送数据 Transmitting data using sockets
  - 套接字地址 Socket addresses
  - `getaddrinfo`
- 建立连接 **Setting up connections**
- 应用层协议示例：**HTTP Application protocol example: HTTP**

协议栈 Protocol Stacks

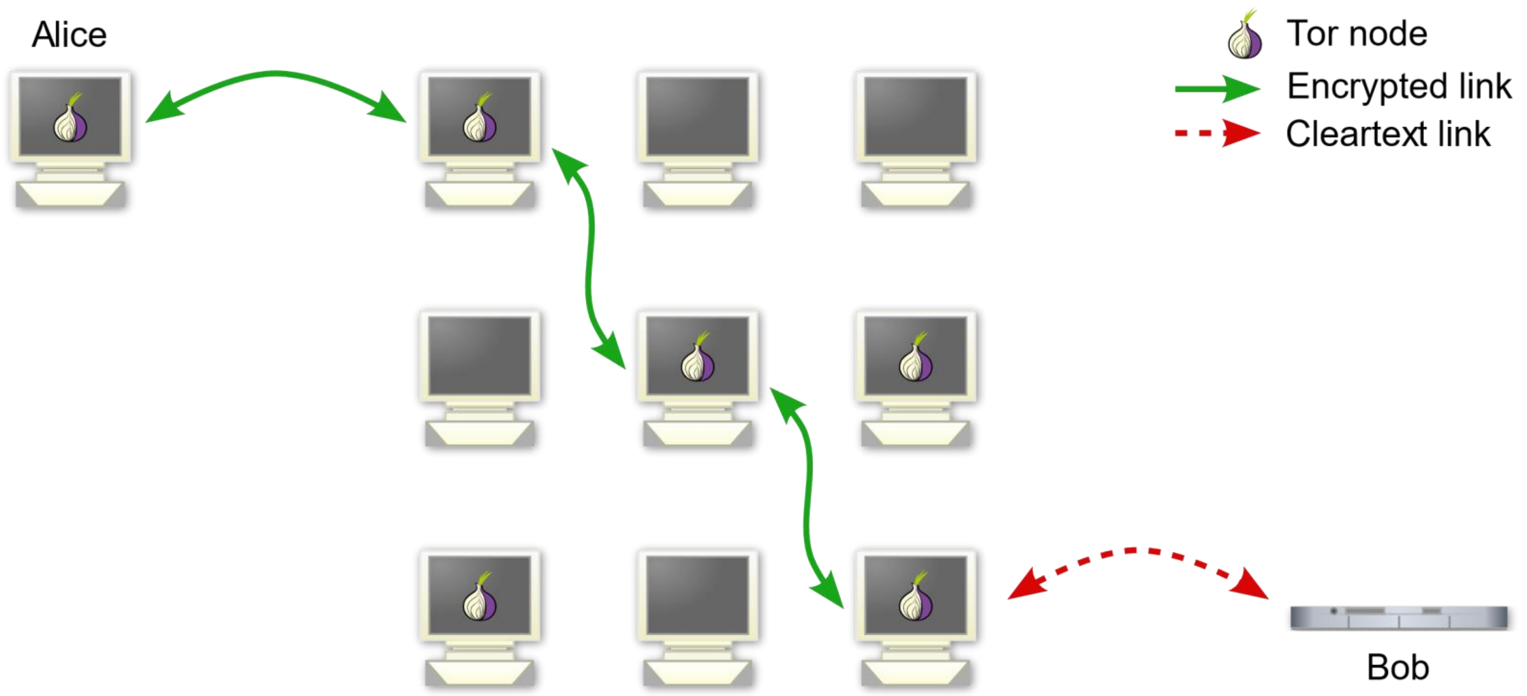
OSI模型 OSI Model

Application
Presentation
Session
Transport
Network
Data Link
Physical

互联网模型 Internet Model

Application	HTTP	SMTP	SSH	DNS
Security	TLS			
Transport	TCP			UDP
Addressing	IP			
Physical Link	Ethernet	WiFi	SDH	

洋葱网站又名“暗网”
Onion sites aka “the dark web”

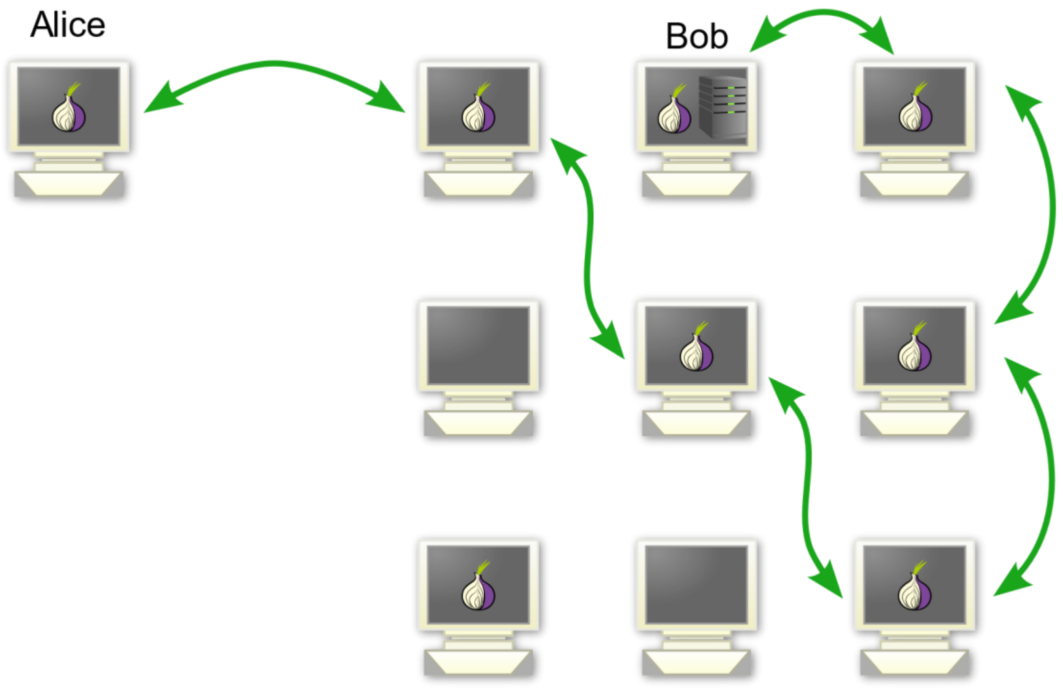


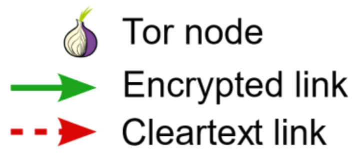
这不是暗网。这只是使用**Tor（The Onion Router）**来保护“网络”的正常使用。

This isn’t the dark web.

This is just using Tor to protect regular use of the ‘net.

洋葱网站又名“暗网”
Onion sites aka “the dark web”





“暗网”由只有在使用**Tor**和类似协议时才能访问的网站和其他服务组成。

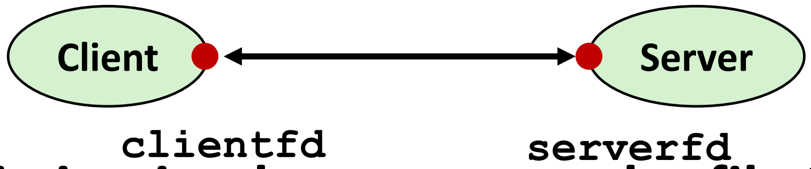
The “dark web” consists of web sites and other services that are accessible *only* when using Tor and similar protocols.

议题

- 上次的问题 Questions from yesterday
- **上次没有讨论的内容 Material we didn’t get to yesterday**
  - 使用套接字发送数据 Transmitting data using sockets
  - 套接字地址 Socket addresses
  - **`getaddrinfo`**
- 建立连接 Setting up connections
- 应用层协议示例：HTTP　Application protocol example: HTTP

Sockets

- **What is a socket?**
  - To the kernel, a socket is an endpoint of communication
  - To an application, a socket is a file descriptor that lets the application read/write from/to the network
  - Using the FD abstraction lets you reuse code & interfaces
- **Clients and servers communicate with each other by reading from and writing to socket descriptors**

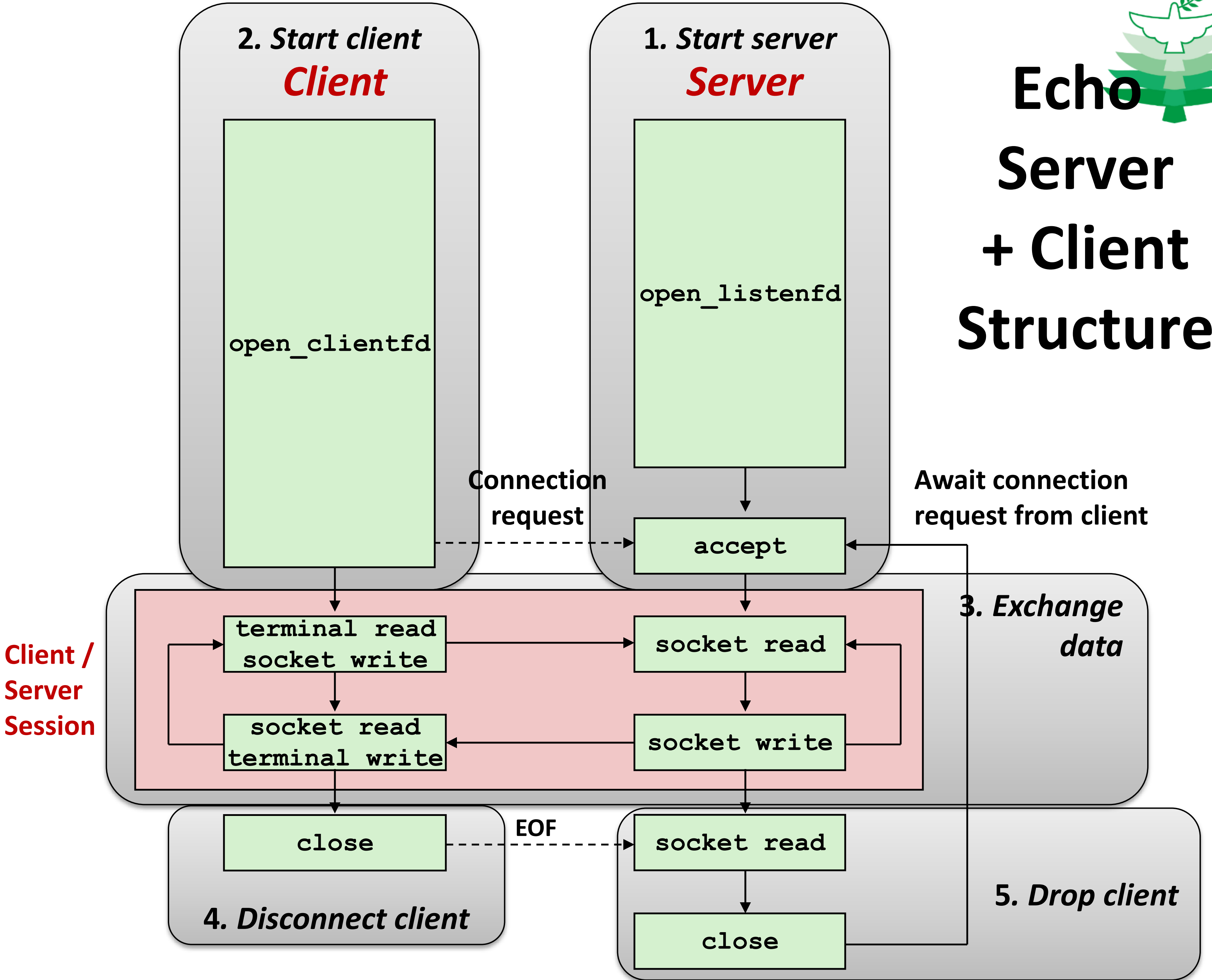


- **The main distinction between regular file I/O and socket I/O is how the application "opens" the socket descriptors**

Socket Programming Example

- **Echo server and client**
- **Server**
  - Accepts connection request
  - Repeats back lines as they are typed
- **Client**
  - Requests connection to server
  - Repeatedly:
    - Read line from terminal
    - Send to server
    - Read reply from server
    - Print line to terminal

Echo Server + Client Structure



Recall: Unbuffered RIO Input/Output

- **Same interface as Unix `read` and `write`**
- **Especially useful for transferring data on network sockets**

```
#include "csapp.h"

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

Return: num. bytes transferred if OK, 0 on EOF (`rio_readn` only), -1 on error

- `rio_readn` returns short count only if it encounters EOF
  - Only use it when you know how many bytes to read
- `rio_writen` never returns a short count
- Calls to `rio_readn` and `rio_writen` can be interleaved arbitrarily on the same descriptor

Recall: Buffered RIO Input Functions

- Efficiently read text lines and binary data from a file partially cached in an internal memory buffer

```
#include "csapp.h"

void rio_readinitb(rio_t *rp, int fd);

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

Return: num. bytes read if OK, 0 on EOF, -1 on error

- **`rio_readlineb`** reads a ***text line*** of up to **`maxlen`** bytes from file **`fd`** and stores the line in **`usrbuf`**
  - Especially useful for reading text lines from network sockets
- Stopping conditions
  - **`maxlen`** bytes read
  - EOF encountered
  - Newline (‘**`\n`**’) encountered

Echo Client: Main Routine

```
#include "csapp.h"

int main(int argc, char **argv)
{
    int clientfd;
    char *host, *port, buf[MAXLINE];
    rio_t rio;

    host = argv[1];
    port = argv[2];

    clientfd = Open_clientfd(host, port);
    Rio_readinitb(&rio, clientfd);

    while (Fgets(buf, MAXLINE, stdin) != NULL) {
        Rio_writen(clientfd, buf, strlen(buf));
        Rio_readlineb(&rio, buf, MAXLINE);
        Fputs(buf, stdout);
    }
    Close(clientfd);
    exit(0);
}
```

echoclient.c

Echo Server: `echo` function

- **The server uses RIO to read and echo text lines until EOF (end-of-file) condition is encountered.**
  - EOF condition caused by client calling `close(clientfd)`

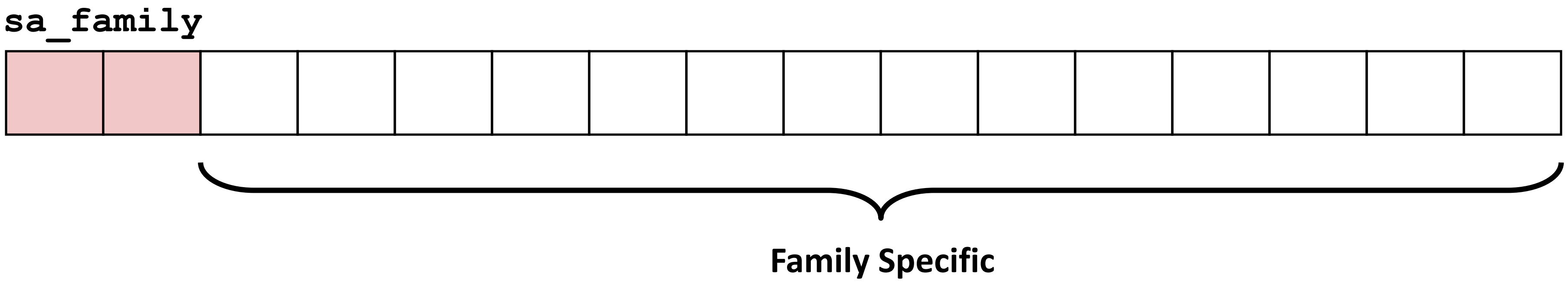
```
void echo(int connfd)
{
    size_t n;
    char buf[MAXLINE];
    rio_t rio;

    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        printf("server received %d bytes\n", (int)n);
        Rio_writen(connfd, buf, n);
    }
}
```
echo.c

Socket Address Structures

- **Generic socket address:**
  - For address arguments to **`connect`**, **`bind`**, and **`accept`** *(next lecture)*
  - In C++ this would be an abstract base class
  - For casting convenience, we adopt the Stevens convention:

    `typedef struct sockaddr SA;`

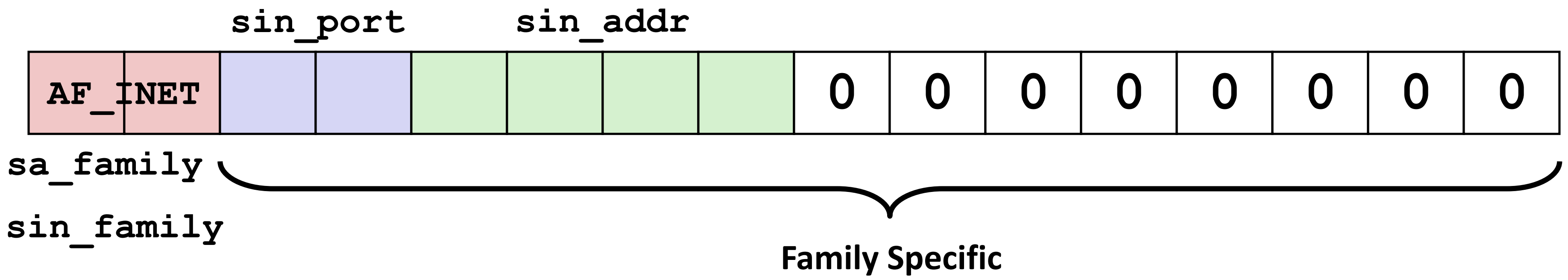
```
struct sockaddr {
  uint16_t  sa_family;    /* Protocol family */
  char      sa_data[14];  /* Address data  */
};
```



Socket Address Structures

- **Internet (IPv4) specific socket address:**
  - Must cast (`struct sockaddr_in *`) to (`struct sockaddr *`) for functions that take socket address arguments.

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



Host and Service Conversion: `getaddrinfo`

- **`getaddrinfo` is the modern way to convert string representations of hostnames, host addresses, ports, and service names to socket address structures.**
  - Replaces obsolete `gethostbyname` and `getservbyname` funcs.

- **Advantages:**
  - Reentrant (can be safely used by threaded programs).
  - Allows us to write portable protocol-independent code
    - Works with both IPv4 and IPv6

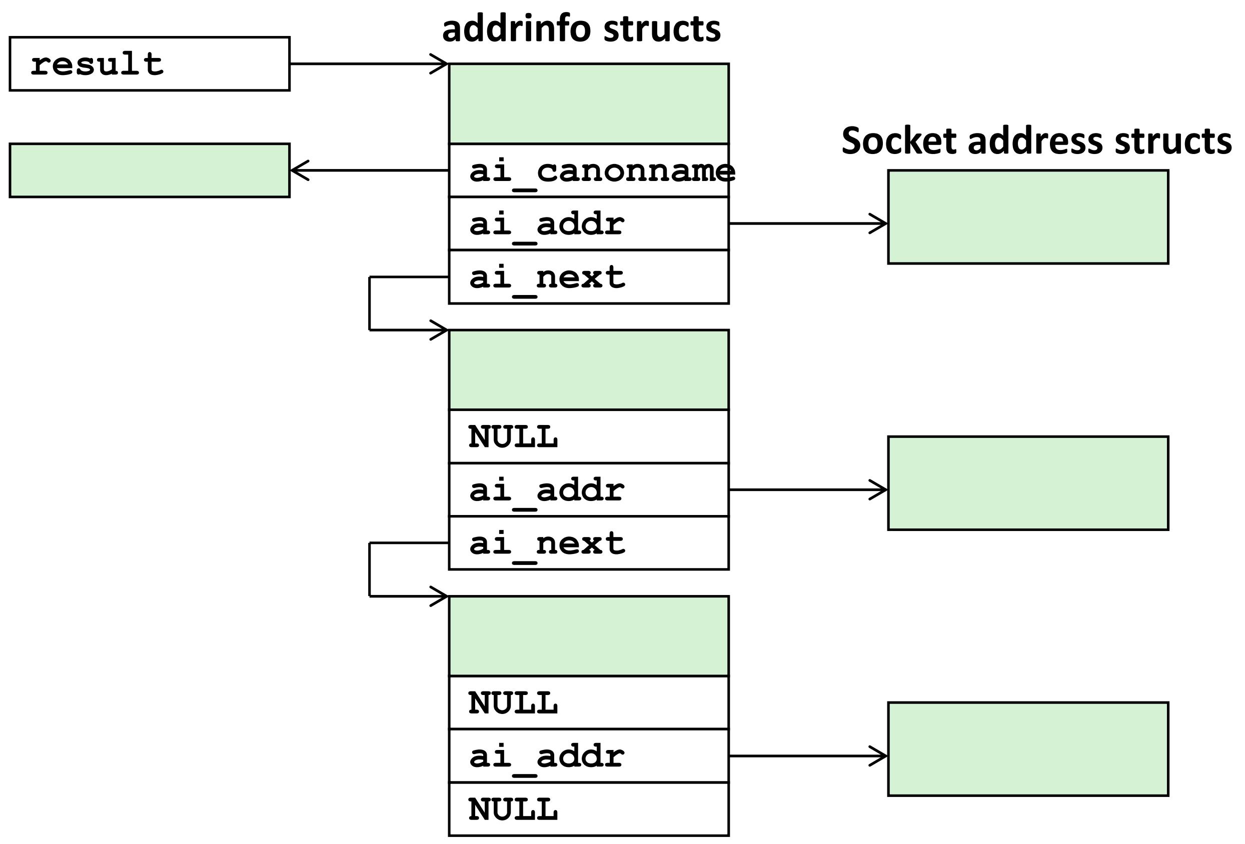
- **Disadvantages**
  - Somewhat complex
  - Fortunately, a small number of usage patterns suffice in most cases.

Host and Service Conversion: `getaddrinfo`

```
int getaddrinfo(const char *host,           /* Hostname or address */
                const char *service,        /* Port or service name */
                const struct addrinfo *hints,/* Input parameters */
                struct addrinfo **result);  /* Output linked list */

void freeaddrinfo(struct addrinfo *result); /* Free linked list */

const char *gai_strerror(int errcode);      /* Return error msg */
```

- **Given `host` and `service`, `getaddrinfo` returns `result` that points to a linked list of `addrinfo` structs, each of which points to a corresponding socket address struct, and which contains arguments for the sockets interface functions.**
- **Helper functions:**
  - `freeadderinfo` frees the entire linked list.
  - `gai_strerror` converts error code to an error message.

Linked List Returned by `getaddrinfo`



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 host name */
    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 */
};
```

- **Each addrinfo struct returned by getaddrinfo contains arguments that can be passed directly to `socket` function.**
- **Also points to a socket address struct that can be passed directly to `connect` and `bind` functions .**

(**`socket`**, **`connect`**, **`bind`** to be discussed next)

Host and Service Conversion: `getnameinfo`

- **`getnameinfo` is the inverse of getaddrinfo, converting a socket address to the corresponding host and service.**
  - Replaces obsolete `gethostbyaddr` and `getservbyport` funcs.
  - Reentrant and protocol independent.

```
int getnameinfo(const SA *sa, socklen_t salen, /* In: socket addr */
                char *host, size_t hostlen,    /* Out: host */
                char *serv, size_t servlen,    /* Out: service */
                int flags);                    /* optional flags */
```

Conversion Example

```
#include "csapp.h"

int main(int argc, char **argv)
{
    struct addrinfo *p, *listp, hints;
    char buf[MAXLINE];
    int rc, flags;

    /* Get a list of addrinfo records */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;       /* IPv4 only */
    hints.ai_socktype = SOCK_STREAM; /* Connections only */
    if ((rc = getaddrinfo(argv[1], NULL, &hints, &listp)) != 0) {
        fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(rc));
        exit(1);
    }
```

hostinfo.c

Conversion Example (cont)

```
    /* Walk the list and display each IP address */
    flags = NI_NUMERICHOST; /* Display address instead of name */
    for (p = listp; p; p = p->ai_next) {
        Getnameinfo(p->ai_addr, p->ai_addrlen,
                    buf, MAXLINE, NULL, 0, flags);
        printf("%s\n", buf);
    }

    /* Clean up */
    Freeaddrinfo(listp);

    exit(0);
}
```

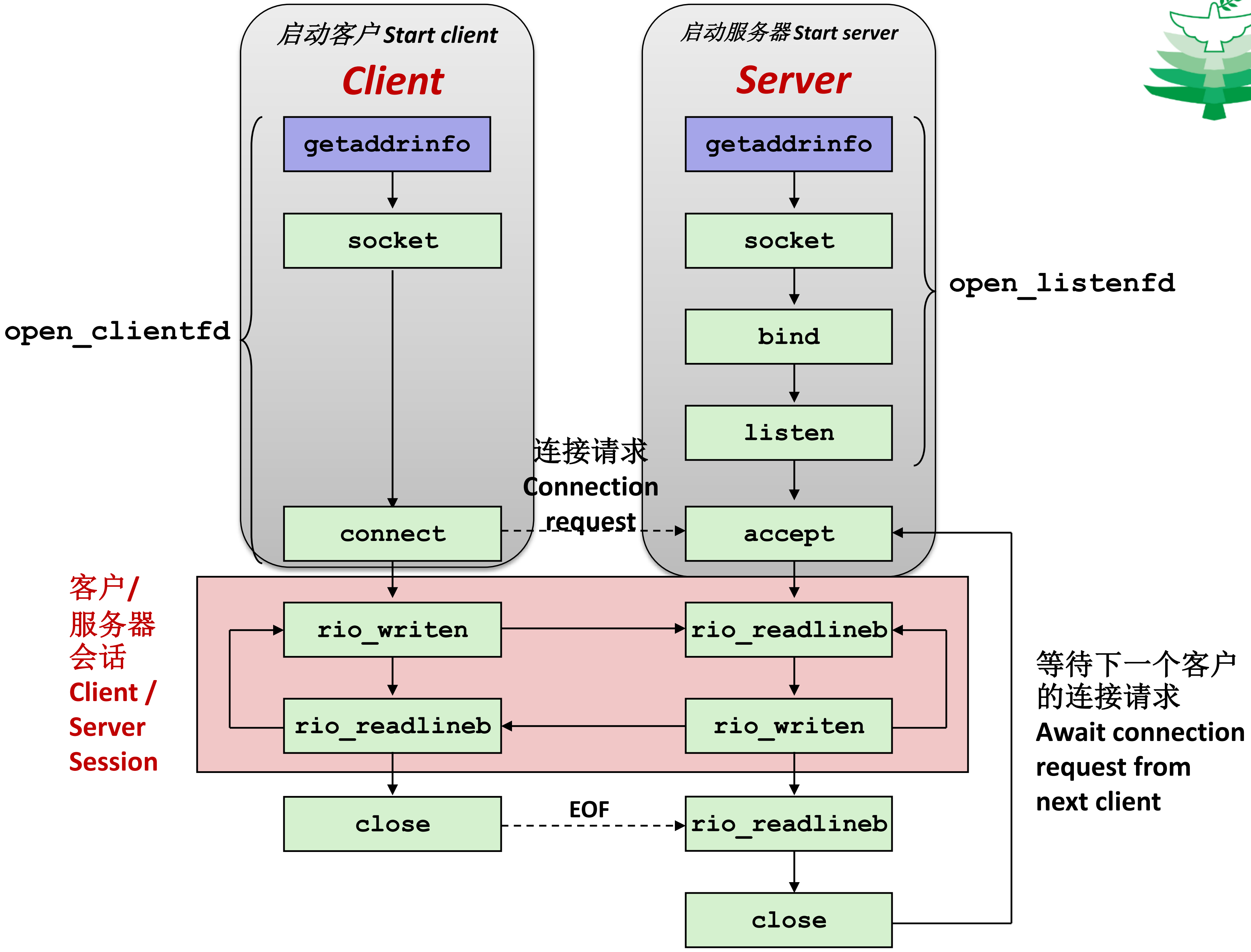
hostinfo.c

Running hostinfo

```
whaleshark> ./hostinfo localhost
127.0.0.1

whaleshark> ./hostinfo whaleshark.ics.cs.cmu.edu
128.2.210.175

whaleshark> ./hostinfo twitter.com
199.16.156.230
199.16.156.38
199.16.156.102
199.16.156.198

whaleshark> ./hostinfo google.com
172.217.15.110
2607:f8b0:4004:802::200e
```

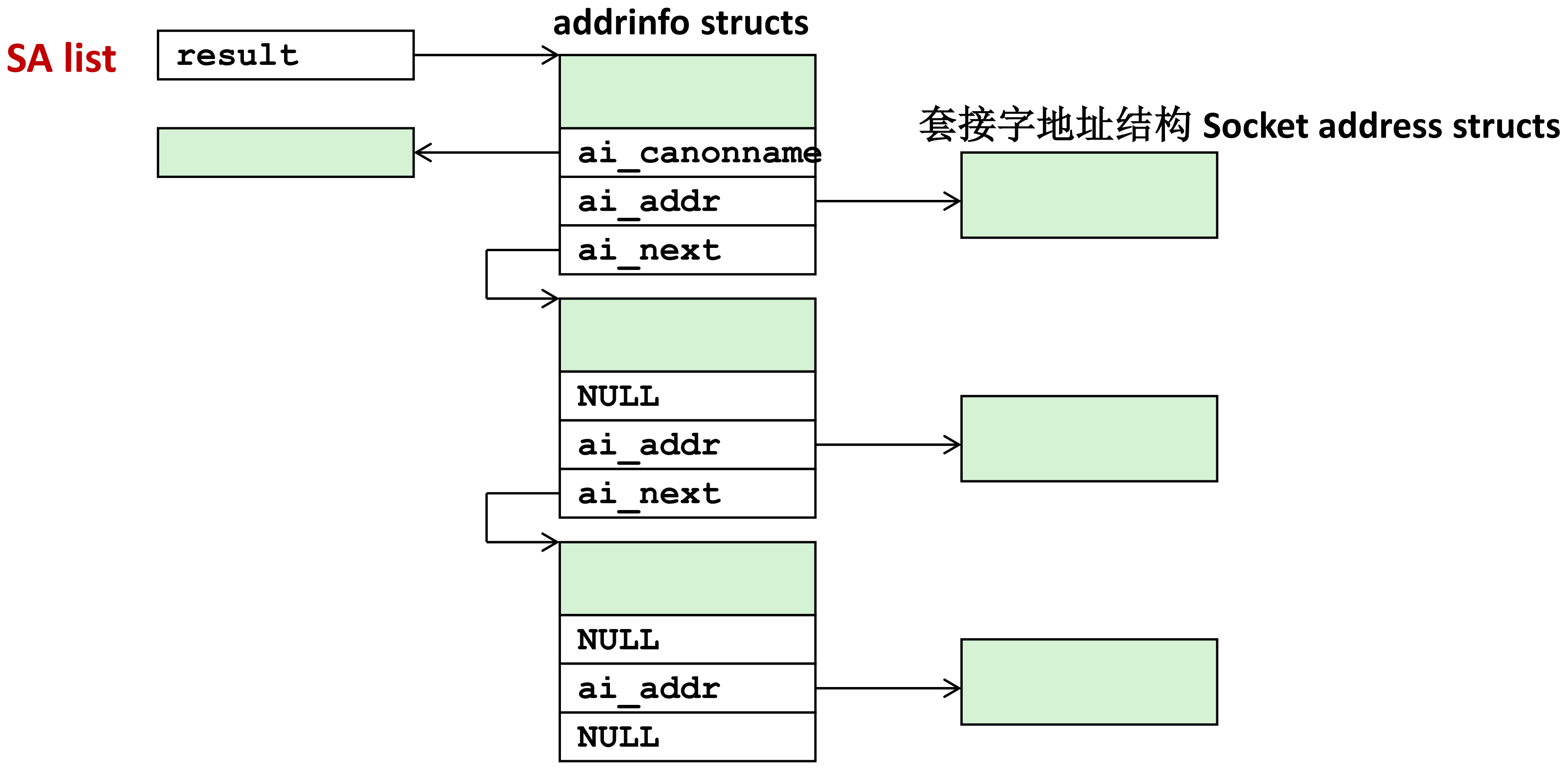
议题

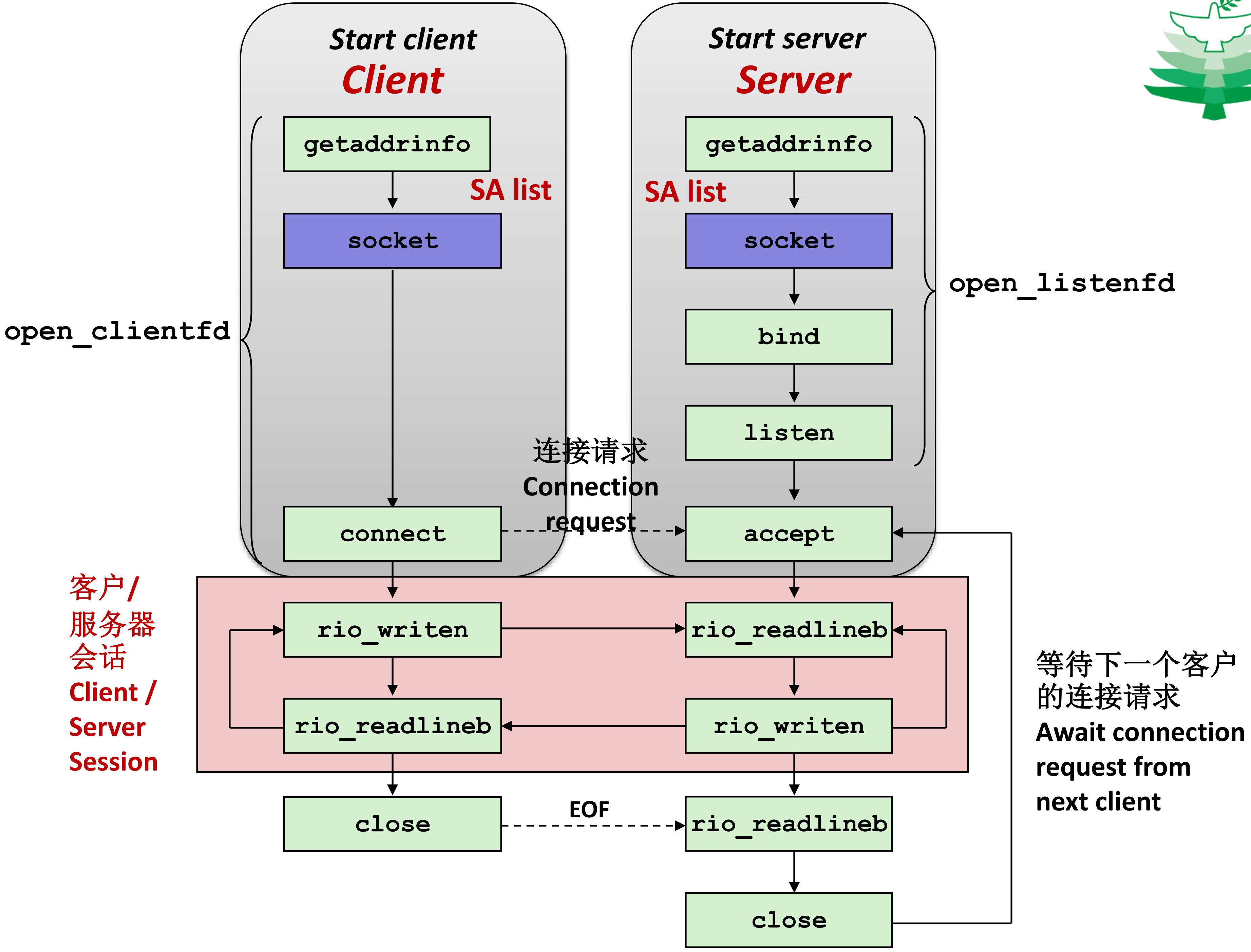
- 上次的问题 **Questions from yesterday**
- 上次没有讨论的内容 **Material we didn't get to yesterday**
  - 使用套接字发送数据 Transmitting data using sockets
  - 套接字地址 Socket addresses
  - `getaddrinfo`
- **建立连接 Setting up connections**
- 应用层协议示例：**HTTP　Application protocol example: HTTP**


启动客户 Start client
Client
启动服务器 Start server
Server
getaddrinfo
socket
connect
open_clientfd
getaddrinfo
socket
bind
listen
accept
open_listenfd
连接请求
Connection request
客户/服务器会话
Client / Server Session
rio_writen
rio_readlineb
rio_readlineb
rio_writen
close
EOF
rio_readlineb
close
等待下一个客户的连接请求
Await connection request from next client

复习：getaddrinfo Review: getaddrinfo

- **Getaddrinfo**将主机名、主机地址、端口和服务名的字符串表示转换成套接字地址结构 **getaddrinfo** converts string representations of hostnames, host addresses, ports, service names to socket address structures




Start client
Client
getaddrinfo
SA list
socket
open_clientfd
connect
连接请求
Connection request
Start server
Server
getaddrinfo
SA list
socket
bind
listen
open_listenfd
accept
客户/服务器会话
Client / Server Session
rio_writen
rio_readlineb
rio_readlineb
rio_writen
close
EOF
rio_readlineb
close
等待下一个客户的连接请求
Await connection request from next client

套接字接口：socket
Sockets Interface: `socket`

- 客户和服务器使用socket函数创建套接字描述符 Clients and servers use the `socket` function to create a *socket descriptor*:

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

- 示例：Example:

```
int clientfd = socket(AF_INET, SOCK_STREAM, 0);
```

特定协议！
Protocol specific!

`AF_INET`: 指示正在使用32位IPv4地址
Indicates that we are using 32-bit IPV4 addresses

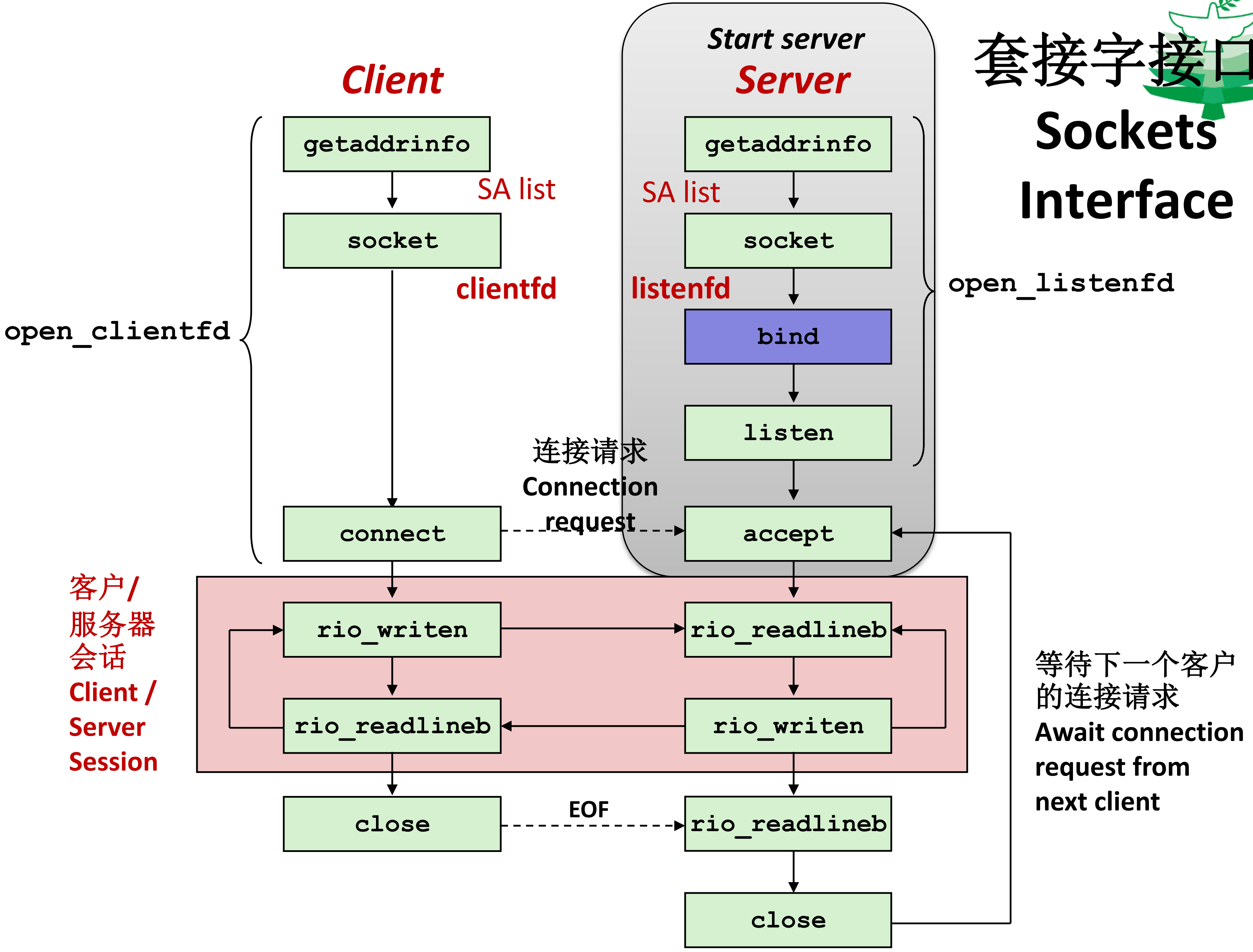
`SOCK_STREAM`: 指示套接字为可靠（TCP）连接端点
Indicates that the socket will be the end point of a reliable (TCP) connection

- 示例：Example:

```
int clientfd = socket(ai->ai_family, ai->ai_socktype,
                      ai->ai_protocol);
```

使用getaddrinfo而且不必知道或关心用哪个协议
Use getaddrinfo and you don't have to know or care which protocol!

套接字接口 Sockets Interface



套接字接口：bind
Sockets Interface: `bind`

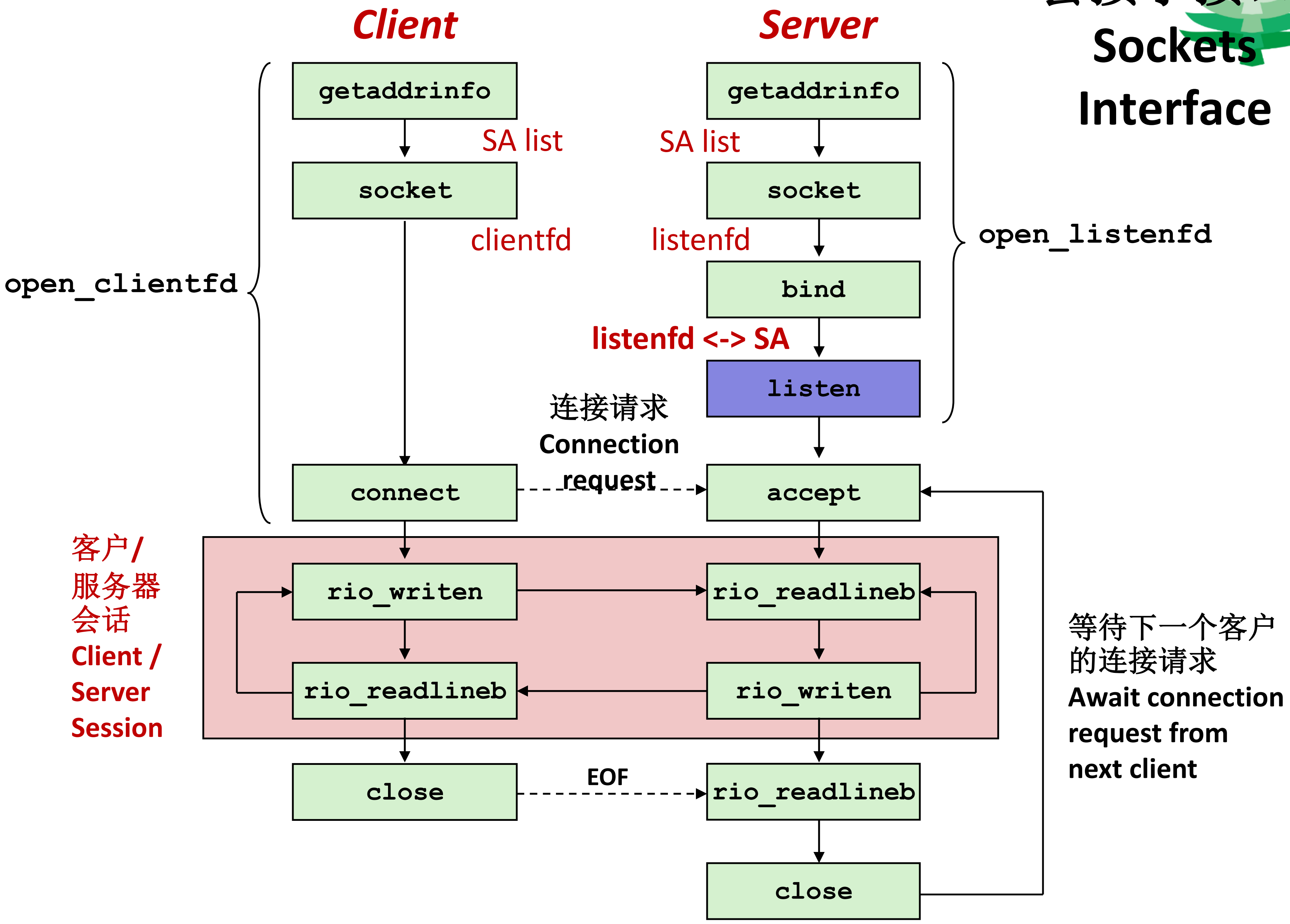
- 服务器使用bind要求内核将服务器套接字地址和套接字描述符相关联 A server uses `bind` to ask the kernel to associate the server's socket address with a socket descriptor:

```
int bind(int sockfd, SA *addr, socklen_t addrlen);
```

Our convention: `typedef struct sockaddr SA;`

- 进程可以通过读取描述符sockfd来读取连接端点为addr到达的字节 Process can read bytes that arrive on the connection whose endpoint is `addr` by reading from descriptor `sockfd`
- 类似地，对sockfd的写入是沿着连接端点addr进行传输 Similarly, writes to `sockfd` are transferred along connection whose endpoint is `addr`
- 最佳实践是使用getaddrinfo提供参数addr和addrlen Best practice is to use `getaddrinfo` to supply the arguments `addr` and `addrlen`.

套接字接口 Sockets Interface



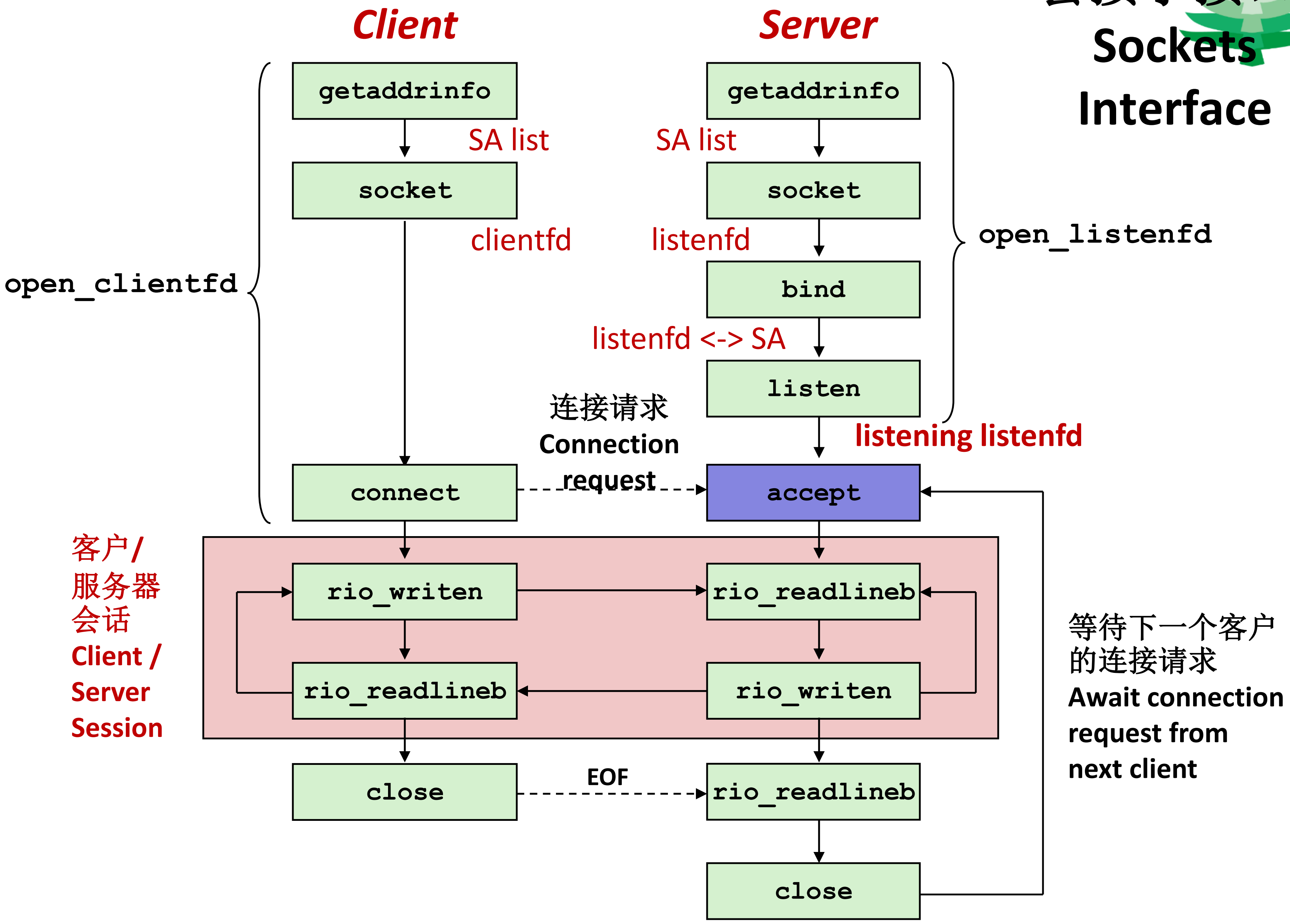
套接字接口：listen
Sockets Interface: listen

- 内核假设来自socket函数的描述符是一个活动套接字，该套接字处于客户端 Kernel assumes that descriptor from socket function is an *active socket* that will be on the client end
- 服务器调用listen函数，告诉内核描述符将由服务器而不是客户端使用： A server calls the listen function to tell the kernel that a descriptor will be used by a server rather than a client:

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

- 将sockfd从活动套接字转换为可接受客户端连接请求的侦听套接字 Converts `sockfd` from an active socket to a *listening socket* that can accept connection requests from clients.
- backlog是关于内核在开始拒绝请求之前应该排队的未完成连接请求数量的提示（默认情况下为128） `backlog` is a hint about the number of outstanding connection requests that the kernel should queue up before starting to refuse requests (128-ish by default)

套接字接口 Sockets Interface



套接字接口：accept
Sockets Interface: `accept`

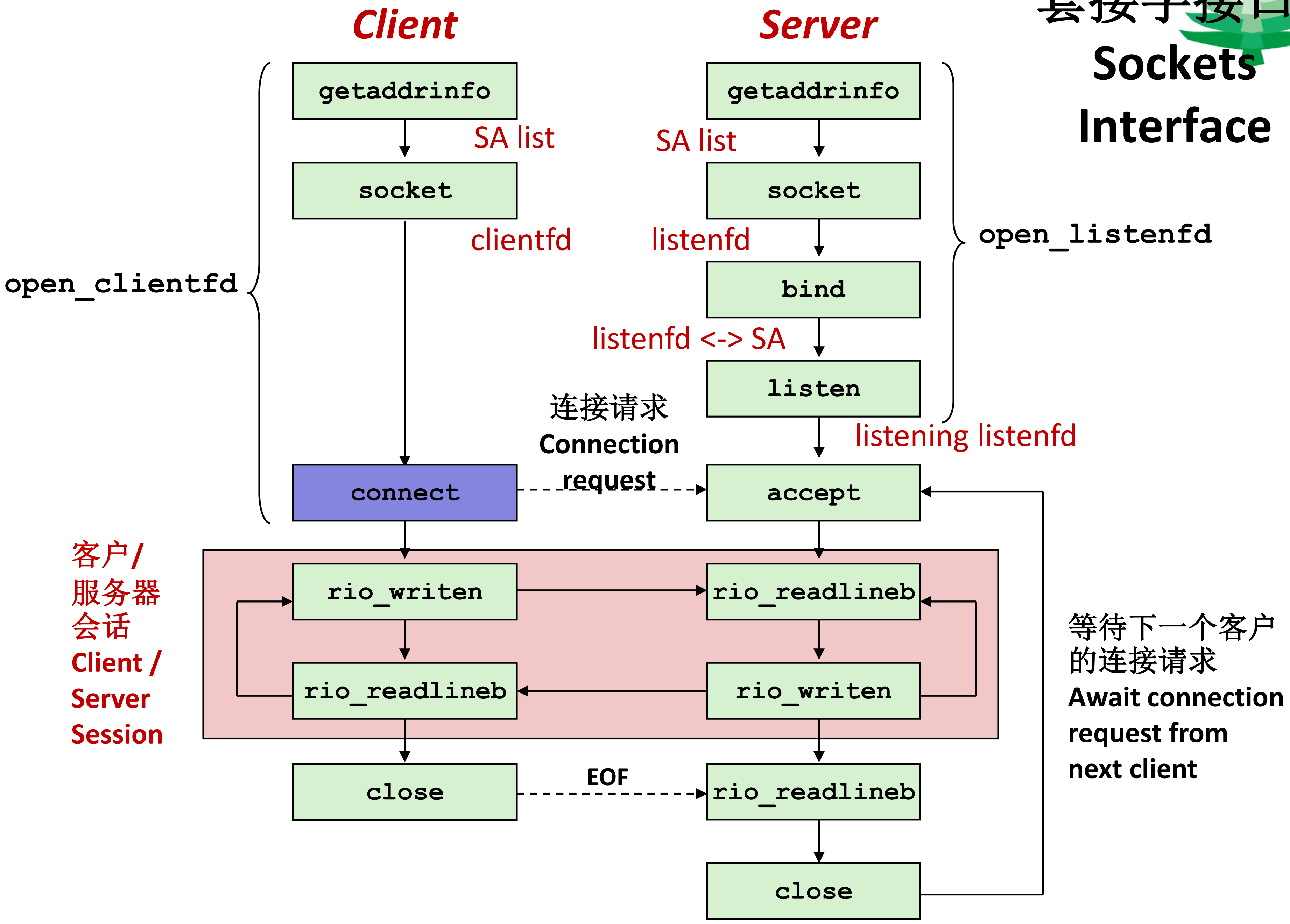
- 服务器通过调用accept等待来自客户端的连接请求：
Servers wait for connection requests from clients by calling `accept`:

```
int accept(int listenfd, SA *addr, int *addrlen);
```

- 等待连接请求到达绑定到listenfd的连接，然后在addr中填写客户端的套接字地址，在addrlen中填写套接字地址的大小 Waits for connection request to arrive on the connection bound to `listenfd`, then fills in client’s socket address in `addr` and size of the socket address in `addrlen`.

- 返回可用于通过Unix I/O例程与客户端通信的连接描述符connfd Returns a *connected descriptor* `connfd` that can be used to communicate with the client via Unix I/O routines.

套接字接口 Sockets Interface



套接字接口：connect
Sockets Interface: connect

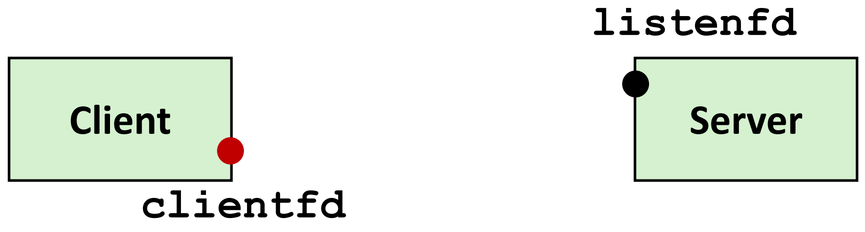
- 客户端通过调用connect建立与服务器的连接： A client establishes a connection with a server by calling connect:

```
int connect(int clientfd, SA *addr, socklen_t addrlen);
```

- 尝试在套接字地址addr处与服务器建立连接 Attempts to establish a connection with server at socket address addr
  - 如果成功，那么clientfd现在就可以读写了。 If successful, then **clientfd** is now ready for reading and writing.
  - 生成的连接以套接字对为特征 Resulting connection is characterized by socket pair

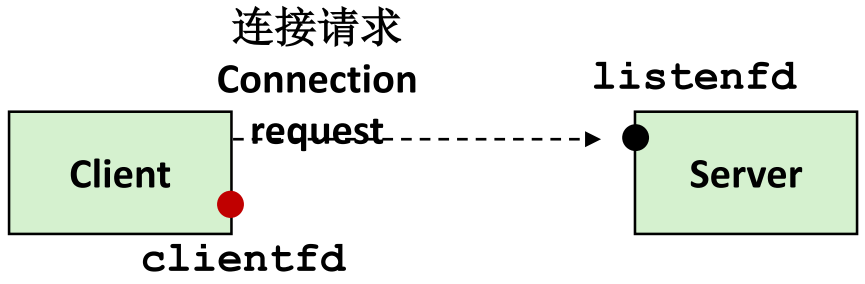
    (**x:y, addr.sin_addr:addr.sin_port**)
    - x是客户端地址 **x** is client address
    - y是唯一标识客户端主机上的客户端进程的临时端口 **y** is ephemeral port that uniquely identifies client process on client host
- 最佳实践是使用getaddrinfo提供参数addr和addrlen Best practice is to use getaddrinfo to supply the arguments

connect/accept Illustrated-说明



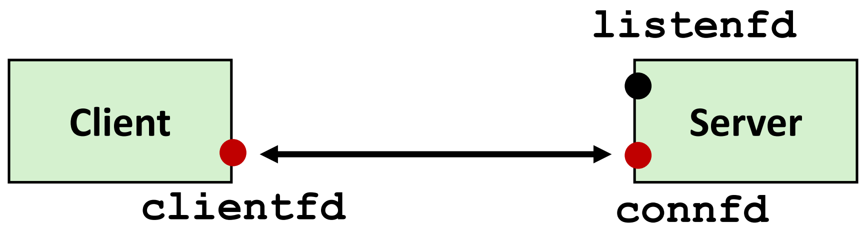
1.服务器阻塞在accept函数，等待侦听描述符listenfd上的连接请求

1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`



2.客户端通过阻塞式调用connect来发出连接请求

2. Client makes connection request by calling and blocking in `connect`

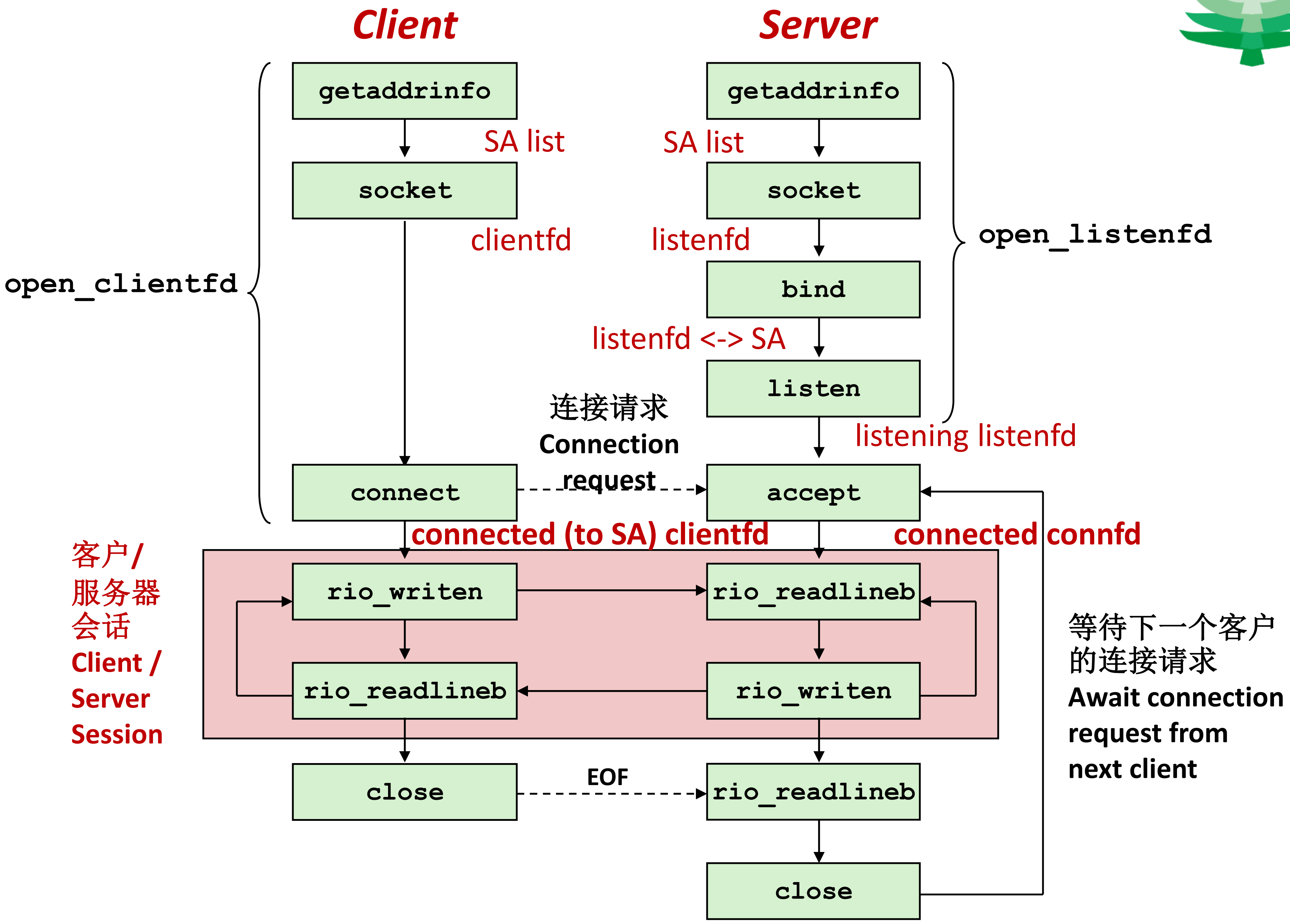


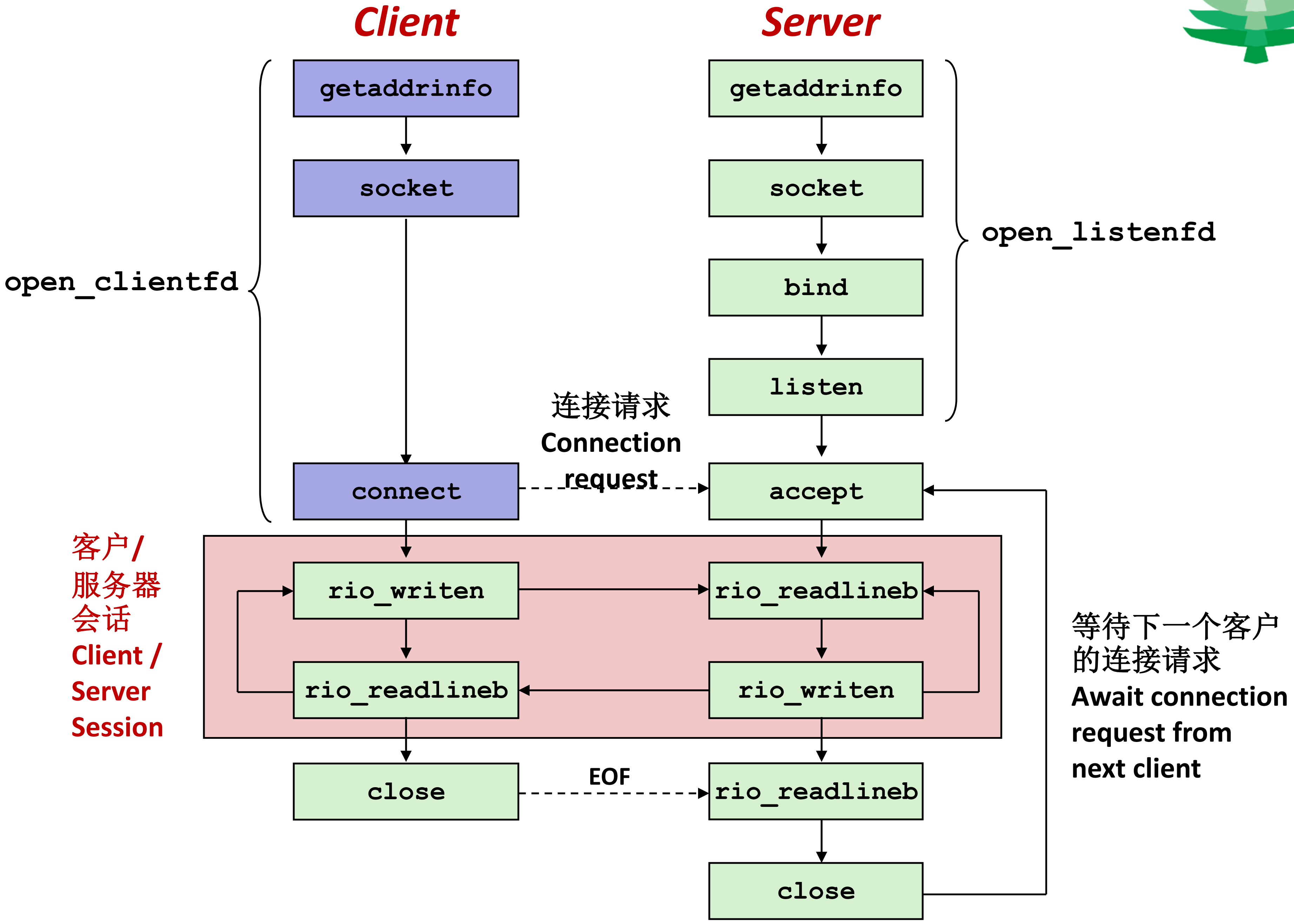
3.服务器从accept返回connfd，客户端从connect返回。现在已在clientfd和connfd之间建立连接

3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`

连接与侦听描述符对比
Connected vs. Listening Descriptors

- 侦听描述符 **Listening descriptor**
  - 客户端连接<u>请求</u>的端点 End point for client connection <u>requests</u>
  - 创建一次并在服务器的生命周期内存在 Created once and exists for lifetime of the server
- 连接描述符 **Connected descriptor**
  - 客户端和服务器之间<u>连接</u>的端点 End point of the <u>connection</u> between client and server
  - 每当服务器接受来自客户端的连接请求时，都会创建一个新的描述符 A new descriptor is created each time the server accepts a connection request from a client
  - 仅在服务客户端所需的时间内存在 Exists only as long as it takes to service client
- 为什么有区别？ **Why the distinction?**
  - 允许同时通过多个客户端连接进行通信的并发服务器 Allows for concurrent servers that can communicate over many client connections simultaneously
    - 例如每次我们收到一个新的请求时，我们都会创建一个子进程来处理这个请求 E.g., Each time we receive a new request, we fork a


Client
Server
getaddrinfo
SA list
socket
clientfd
open_clientfd
getaddrinfo
SA list
socket
listenfd
bind
listenfd <-> SA
listen
listening listenfd
open_listenfd
连接请求
Connection request
connect
accept
connected (to SA) clientfd
connected connfd
客户/服务器会话
Client / Server Session
rio_writen
rio_readlineb
rio_readlineb
rio_writen
close
EOF
rio_readlineb
close
等待下一个客户的连接请求
Await connection request from next client


Client
Server
getaddrinfo
socket
connect
getaddrinfo
socket
bind
listen
accept
open_clientfd
open_listenfd
连接请求
Connection request
客户/服务器会话
Client / Server Session
rio_writen
rio_readlineb
rio_readlineb
rio_writen
close
EOF
rio_readlineb
close
等待下一个客户的连接请求
Await connection request from next client

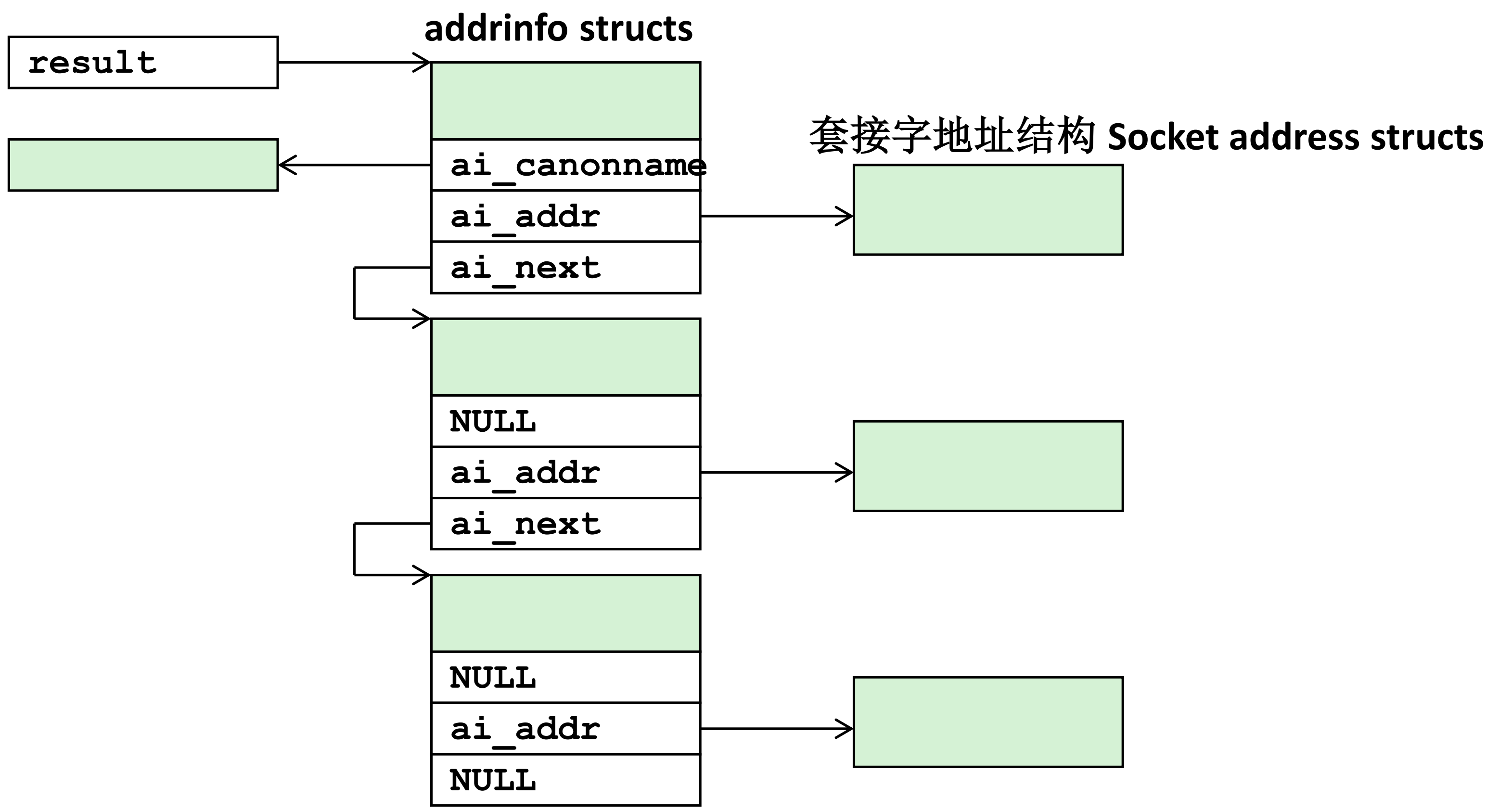
套接字助手：open_clientfd
Sockets Helper: open_clientfd

- 与服务器建立一个连接 Establish a connection with a server

```
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 numeric port arg. */
  hints.ai_flags |= AI_ADDRCONFIG;  /* Recommended for connections */
  Getaddrinfo(hostname, port, &hints, &listp);
```
csapp.c

AI_ADDRCONFIG表示“在此计算机上使用IPv4和IPv6中的任何一种”。这是客户机的良好实践，而不是服务器。
AI_ADDRCONFIG means “use whichever of IPv4 and IPv6 works on this computer”. Good practice for clients, not for servers.

getaddrinfo

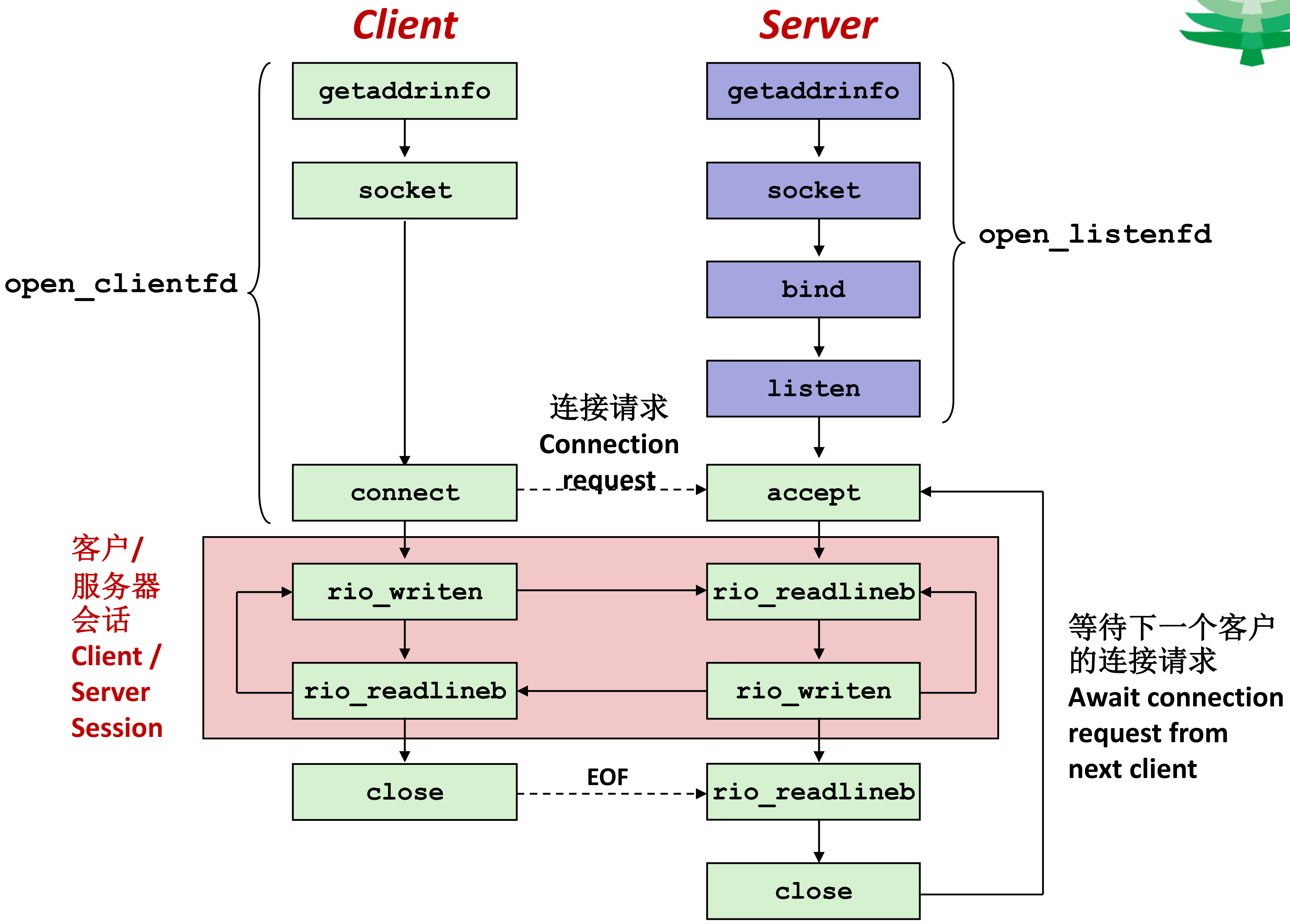


- 客户端：遍历此列表，依次尝试每个套接字地址，直到socket和connect的调用成功 Clients: walk this list, trying each socket address in turn, until the calls to `socket` and `connect` succeed.
- 服务器：遍历列表对所有地址调用socket、listen和bind，然后使用select接受其中任何一个地址上的连接（超出我们的范围） Servers: walk the list calling `socket`, `listen`, `bind` for *all* addresses, then use `select` to accept connections on any of them (beyond our scope)

套接字助手：open_clientfd（续）
Sockets Helper: open_clientfd (cont)

```
    /* 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;
}
```

csapp.c


Client
Server
getaddrinfo
socket
connect
open_clientfd
getaddrinfo
socket
bind
listen
open_listenfd
连接请求
Connection request
accept
客户/服务器会话
Client / Server Session
rio_writen
rio_readlineb
rio_readlineb
rio_writen
close
EOF
rio_readlineb
close
等待下一个客户的连接请求
Await connection request from next client

套接字助手：open_listenfd
Sockets Helper: open_listenfd

- 创建侦听描述符，用于接受客户端的连接请求 Create a listening descriptor that can be used to accept connection requests from clients.

```
 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 connect. */
    hints.ai_flags = AI_PASSIVE | AI_ADDRCONFIG; /* …on any IP addr */
    hints.ai_flags |= AI_NUMERICSERV;            /* …using port no. */
    Getaddrinfo(NULL, port, &hints, &listp);
```

csapp.c

AI_PASSIVE意味着“我计划侦听这个套接字” AI_PASSIVE means “I plan to listen on this socket.”

AI_ADDRCONFIG正常情况下不用于服务器，但是出于方便使用它

AI_ADDRCONFIG normally not used for servers, but we use it for convenience

套接字助手：open_listenfd（续）

Sockets Helper: open_listenfd (cont)

```
    /* 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 */
    }
```
csapp.c

生产服务器不会在第一次成功时跳出循环，我们这样做只是为了简单 A production server would not break out of the loop on the first success. We do that for simplicity only.

套接字助手：open_listenfd（续）
Sockets Helper: open_listenfd (cont)

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

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

- 关键点：open_clientfd和open_listenfd两个都是与任何特定IP版本无关的 **Key point:** `open_clientfd` and `open_listenfd` are both independent of any particular version of IP.

使用telnet测试服务器
Testing Servers Using `telnet`

- **telnet程序对于测试通过互联网连接传输ASCII字符串的服务器非常有用 The `telnet` program is invaluable for testing servers that transmit ASCII strings over Internet connections**
  - 我们的简单回声服务器 Our simple echo server
  - Web服务器 Web servers
  - 邮件服务器 Mail servers

- 用法： **Usage:**
  - linux>telnet <主机> <端口号> **`linux>`** ***`telnet <host> <portnumber>`***
  - 创建与服务器的连接，该服务器运行在<host>上并侦听端口<portnumber> Creates a connection with a server running on ***`<host>`*** and listening on port ***`<portnumber>`***

用telnet测试回声服务器
Testing the Echo Server With `telnet`

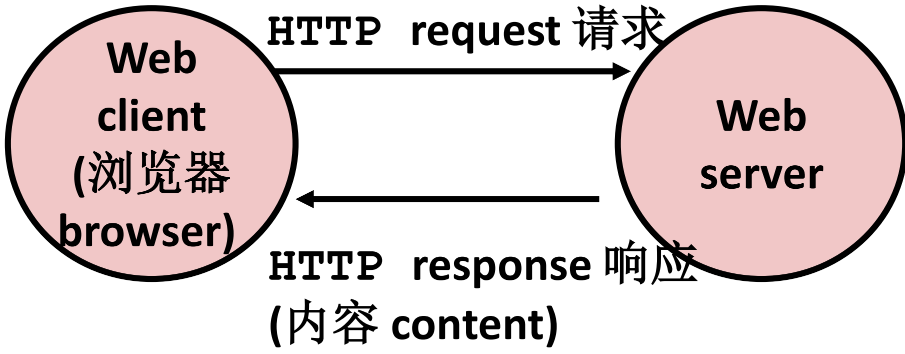
```
whaleshark> ./echoserveri 15213
Connected to (MAKOSHARK.ICS.CS.CMU.EDU, 50280)
server received 11 bytes
server received 8 bytes




makoshark> telnet whaleshark.ics.cs.cmu.edu 15213
Trying 128.2.210.175...
Connected to whaleshark.ics.cs.cmu.edu (128.2.210.175).
Escape character is '^]'.
Hi there!
Hi there!
Howdy!
Howdy!
^]
telnet> quit
Connection closed.
makoshark>
```

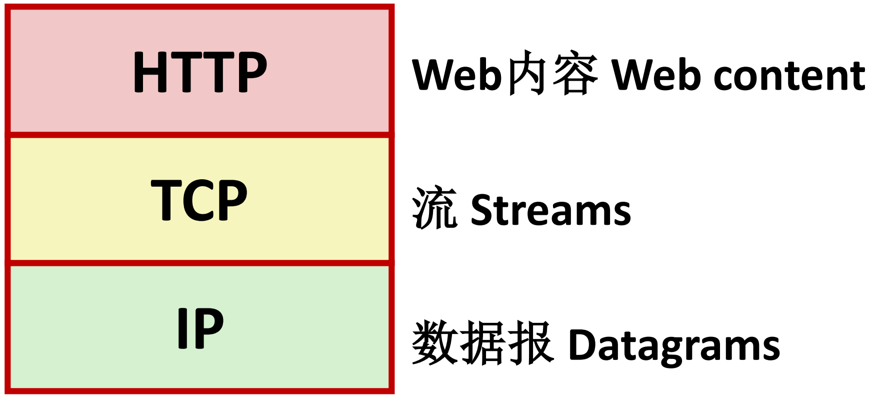
议题

- 上次的问题 **Questions from yesterday**
- 上次没有讨论的内容 **Material we didn’t get to yesterday**
  - 使用套接字发送数据 Transmitting data using sockets
  - 套接字地址 Socket addresses
  - `getaddrinfo`
- 建立连接 **Setting up connections**
- **应用层协议示例：HTTP　Application protocol example: HTTP**

Web服务器基础Web Server Basics

- 客户和服务器使用超文本传送协议（HTTP）通信 **Clients and servers communicate using the HyperText Transfer Protocol (HTTP)**
  - 客户和服务器建立TCP连接 Client and server establish TCP connection
  - 客户请求内容 Client requests content
  - 服务器对请求内容进行响应 Server responds with requested content
  - 客户和服务器关闭连接（最终）Client and server close connection (eventually)
- 当前版本是**HTTP/2.0**，但是**HTTP/1.1**仍在广泛使用 **Current version is HTTP/2.0 but HTTP/1.1 widely used still**
  - RFC 2616, June, 1999.


Web client (浏览器 browser)
HTTP request 请求
Web server
HTTP response 响应 (内容 content)



`http://www.w3.org/Protocols/rfc2616/rfc2616.html`

Web内容 Web Content

- **Web服务器返回内容给客户 Web servers return *content* to clients**
  - *内容:* 字节序列，具有相关联的MIME（多用途互联网邮件扩展）类型 *content:* a sequence of bytes with an associated MIME (Multipurpose Internet Mail Extensions) type

- **MIME类型示例 Example MIME types**

`text/html`	HTML文档 HTML document
`text/plain`	无格式文本 Unformatted text
`image/gif`	GIF格式编码的二进制图像 Binary image encoded in GIF format
`image/png`	PNG格式编码的二进制图像 Binary image encoded in PNG format
`image/jpeg`	JPG格式编码的二进制图像 Binary image encoded in JPEG format

完整的MIME类型列表网址：You can find the complete list of MIME types at:
`http://www.iana.org/assignments/media-types/media-types.xhtml`

静态和动态内容 Static and Dynamic Content

- **HTTP响应中返回的内容可以是*静态*的，也可以是*动态*的 The content returned in HTTP responses can be either *static* or *dynamic***
  - 静态内容：存储在文件中并响应HTTP请求检索的内容 *Static content*: content stored in files and retrieved in response to an HTTP request
    - 示例：HTML文件、图像、音频剪辑、Javascript程序 Examples: HTML files, images, audio clips, Javascript programs
    - 请求标识哪个内容文件 Request identifies which content file
  - 动态内容：响应HTTP请求而动态生成的内容 *Dynamic content*: content produced on-the-fly in response to an HTTP request
    - 示例：由服务器代表客户端执行的程序生成的内容 Example: content produced by a program executed by the server on behalf of the client
    - 请求标识包含可执行代码的文件 Request identifies file containing executable code
- **Web内容关联一个文件，该文件由服务器管理 *Web content associated with a file that is managed by the server***

统一资源定位符和客户及服务器如何使用
URLs and how clients and servers use them

- 文件的惟一名字：**URL**（统一资源定位符） **Unique name for a file: URL (Universal Resource Locator)**
- 示例**URL**： **Example URL:**
  `http://www.cmu.edu:80/index.html`
- 客户使用前缀(**`http://www.cmu.edu:80`**) 进行推理：
  Clients use *prefix* (`http://www.cmu.edu:80`) to infer:
  - 要联系的服务器（协议）类型 What kind (protocol) of server to contact (HTTP)
  - 服务器所在位置 Where the server is (**`www.cmu.edu`**)
  - 它正在侦听哪个端口 What port it is listening on (80)

统一资源定位符和客户及服务器如何使用
URLs and how clients and servers use them

- 服务器使用后缀（/index.html）： **Servers use *suffix* (`/index.html`) to:**
  - 确定请求是针对静态内容还是动态内容 Determine if request is for static or dynamic content.
    - 对此没有硬性规定 No hard and fast rules for this
    - 一种约定：可执行文件位于cgi-bin目录中 One convention: executables reside in **`cgi-bin`** directory
  - 在文件系统上查找文件 Find file on file system
    - 后缀中的首字母“/”表示所请求内容的主目录 Initial “/” in suffix denotes home directory for requested content.
    - 最小后缀为“/”，服务器扩展为配置的默认文件名（通常为index.html） Minimal suffix is “/”, which server expands to configured default filename (usually, **`index.html`**)

HTTP请求示例 HTTP Request Example

```
GET / HTTP/1.1                    Client: request line
Host: www.cmu.edu                 Client: required HTTP/1.1 header
                                  Client: blank line terminates headers
```

- HTTP标准要求每个文本行以“\r\n”（回车换行）结尾 HTTP standard requires that each text line end with "\r\n"
- 空行（“\r\n”）终止请求和响应首部 Blank line ("\r\n") terminates request and response headers

HTTP请求 HTTP Requests

- **HTTP请求是一个*请求行*，后跟零个或多个*请求首部* HTTP request is a *request line*, followed by zero or more *request headers***
- 请求行：方法 **uri** 版本 **Request line: `<method> <uri> <version>`**
  - 方法是GET、POST、OPTIONS、HEAD、PUT、DELETE或TRACE之一 **`<method>`** is one of **`GET`**, **`POST`**, **`OPTIONS`**, **`HEAD`**, **`PUT`**, **`DELETE`**, or **`TRACE`**
  - uri通常是代理的URL，服务器的URL后缀 **`<uri>`** is typically URL for proxies, URL suffix for servers
    - URL是URI（统一资源标识符）的一种类型 A URL is a type of URI (Uniform Resource Identifier)
    - 参见 See http://www.ietf.org/rfc/rfc2396.txt
  - 版本是请求的HTTP版本（HTTP/1.0或HTTP/1.1） **`<version>`** is HTTP version of request (**`HTTP/1.0`** or **`HTTP/1.1`**)

HTTP请求 HTTP Requests

- **HTTP请求是一个*请求行*，后跟零个或多个*请求首部* HTTP request is a *request line*, followed by zero or more *request headers***
- 请求首部：首部名：首部数据 **Request headers: `<header name>: <header data>`**
  - 向服务器提供其他信息 Provide additional information to the server

HTTP响应 HTTP Responses

- **HTTP响应是一个*响应行*，后跟零个或多个*响应首部*，可能后跟*内容*，并用空行（“\r\n”）分隔首部和内容 HTTP response is a *response line* followed by zero or more *response headers*, possibly followed by *content*, with blank line (“`\r\n`”) separating headers from content.**
- 响应行：**Response line:**

  版本 状态代码 状态消息 **`<version> <status code> <status msg>`**

  - 版本是响应的HTTP版本 <version> is HTTP version of the response
  - 状态码是数字状态 <status code> is numeric status
  - 状态消息是对应的英文文本 <status msg> is corresponding English text

`200`	**`OK`**	请求已正确处理 Request was handled without error
`301`	**`Moved`**	提供备用URL Provide alternate URL
`404`	**`Not found`**	服务器找不到文件 Server couldn’t find the file

HTTP响应 HTTP Responses

- **HTTP响应是一个*响应行*，后跟零个或多个*响应首部*，可能后跟*内容*，并用空行（“\r\n”）分隔首部和内容 HTTP response is a *response line* followed by zero or more *response headers*, possibly followed by *content*, with blank line (“`\r\n`”) separating headers from content.**
- 响应首部：首部名：首部数据 **Response headers: `<header name>: <header data>`**
  - 提供有关响应的其他信息 Provide additional information about response
  - **`Content-Type`**：响应主体中内容的MIME类型 MIME type of content in response body
  - **`Content-Length`**:响应主体中内容的长度 Length of content in response body

示例HTTP事务 Example HTTP Transaction

```
whaleshark> telnet www.cmu.edu 80           Client: open connection to server
Trying 128.2.42.52...                       Telnet prints 3 lines to terminal
Connected to WWW-CMU-PROD-VIP.ANDREW.cmu.edu.
Escape character is '^]'.
GET / HTTP/1.1                              Client: request line
Host: www.cmu.edu                           Client: required HTTP/1.1 header
                                            Client: blank line terminates headers
HTTP/1.1 301 Moved Permanently              Server: response line
Date: Wed, 05 Nov 2014 17:05:11 GMT         Server: followed by 5 response headers
Server: Apache/1.3.42 (Unix)                Server: this is an Apache server
Location: http://www.cmu.edu/index.shtml    Server: page has moved here
Transfer-Encoding: chunked                  Server: response body will be chunked
Content-Type: text/html; charset=...        Server: expect HTML in response body
                                            Server: empty line terminates headers
15c                                         Server: first line in response body
<HTML><HEAD>                                Server: start of HTML content
…
</BODY></HTML>                              Server: end of HTML content
0                                           Server: last line in response body
Connection closed by foreign host.          Server: closes connection
```

- HTTP标准要求每个文本行以 "\r\n"（回车换行）结尾 HTTP standard requires that each text line end with "`\r\n`"
- 空行（"\r\n"）终止请求和响应首部 Blank line ("`\r\n`") terminates request and response headers

示例HTTP事务，例2
Example HTTP Transaction, Take 2

```
whaleshark> telnet www.cmu.edu 80          Client: open connection to server
Trying 128.2.42.52...                      Telnet prints 3 lines to terminal
Connected to WWW-CMU-PROD-VIP.ANDREW.cmu.edu.
Escape character is '^]'.
GET /index.shtml HTTP/1.1                  Client: request line
Host: www.cmu.edu                          Client: required HTTP/1.1 header
                                           Client: blank line terminates headers
HTTP/1.1 200 OK                            Server: response line
Date: Wed, 05 Nov 2014 17:37:26 GMT        Server: followed by 4 response headers
Server: Apache/1.3.42 (Unix)
Transfer-Encoding: chunked
Content-Type: text/html; charset=...
                                           Server: empty line terminates headers
1000                                       Server: begin response body
<html ..>                                  Server: first line of HTML content
…
</html>
0                                          Server: end response body
Connection closed by foreign host.         Server: close connection
```

示例HTTP(S)事务，例3 Example HTTP(S) Transaction, Take 3

```
whaleshark> openssl s_client www.cs.cmu.edu:443
CONNECTED(00000005)
…
Certificate chain
…
-
Server certificate
-----BEGIN CERTIFICATE-----
MIIGDjCCBPagAwIBAgIRAMiF7LBPDoySilnNoU+mp+gwDQYJKoZIhvcNAQELBQAw
djELMAkGA1UEBhMCVVMxCzAJBgNVBAgTAk1JMRIwEAYDVQQHEwlBbm4gQXJib3Ix
EjAQBgNVBAoTCUludGVybmV0MjERMA8GA1UECxMISW5Db21tb24xHzAdBgNVBAMT
wkWkvDVBBCwKXrShVxQNsj6J
…
-----END CERTIFICATE-----
subject=/C=US/postalCode=15213/ST=PA/L=Pittsburgh/street=5000 Forbes
Ave/O=Carnegie Mellon University/OU=School of Computer
Science/CN=www.cs.cmu.edu         issuer=/C=US/ST=MI/L=Ann
Arbor/O=Internet2/OU=InCommon/CN=InCommon RSA Server CA
SSL handshake has read 6274 bytes and written 483 bytes
…
>GET / HTTP/1.0

HTTP/1.1 200 OK
Date: Tue, 12 Nov 2019 04:22:15 GMT
Server: Apache/2.4.10 (Ubuntu)
Set-Cookie: SHIBLOCATION=scsweb; path=/; domain=.cs.cmu.edu
... HTML Content Continues Below ...
```