套接字

Socket

套接字（socket）接口

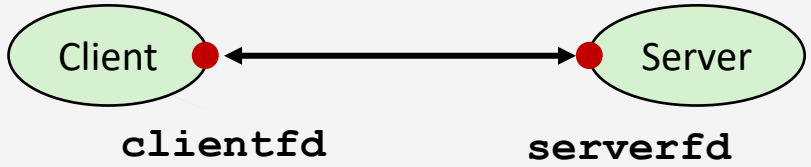
- 与Unix I/O一起使用的系统级函数集，用于构建网络应用程序。
- 这些函数是在80年代初作为Unix的Berkeley发行版的一部分创建的，该发行版包含互联网协议的早期版本。
- 在所有现代系统上可用：
  - Unix家族、Windows、OS X、iOS、Android、ARM。

套接字

Socket

套接字（Socket）

- 什么是套接字？
  - 对于内核来说，套接字是通信的端点。
  - 对于应用程序来说，套接字是一个文件描述符，允许应用程序从网络中读取或向网络中写入数据。
    - 注意：所有Unix I/O设备，包括网络，都被视为文件。
- 客户端和服务器通过读取和写入套接字描述符与彼此通信。

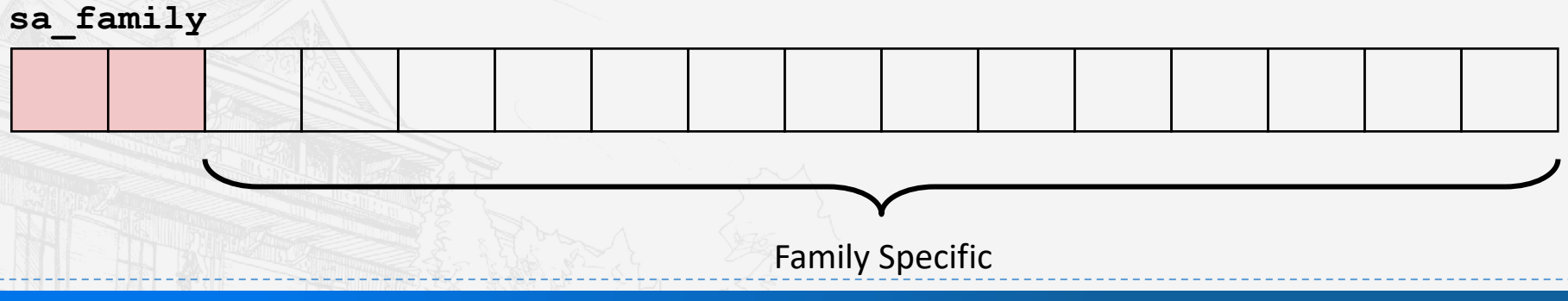


- 常规文件I/O和套接字I/O之间的主要区别在于应用程序如何“打开”套接字描述符。

套接字地址结构体

- 通用套接字地址：
  - 用于 connect、bind 和 accept 等函数的地址参数。
  - 在设计套接字接口时，C语言还没有通用（void *）指针的语法
  - 为了方便转换，我们采用了Stevens约定：typedef struct sockaddr SA;

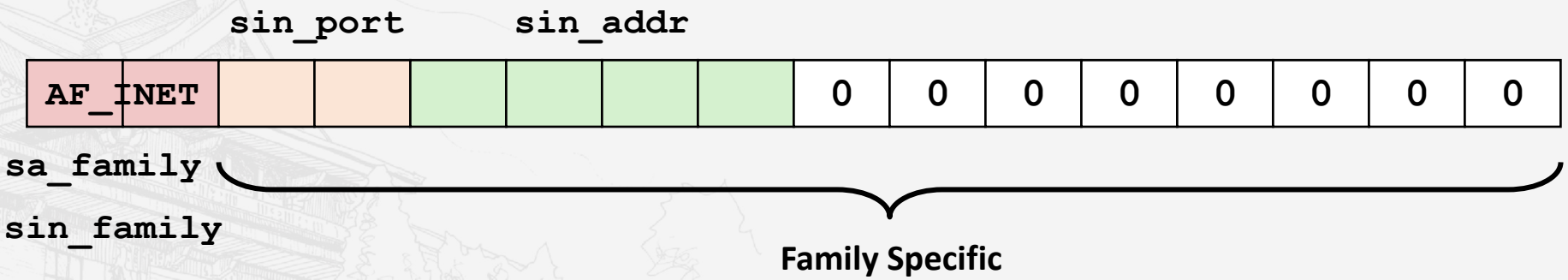
```
struct sockaddr {
  uint16_t  sa_family;    /* Protocol family */
  char      sa_data[14];  /* Address data.  */
};
```



套接字地址结构体

- Internet特定的套接字地址：
  - 对于那些需要套接字地址参数的函数，必须将 (struct sockaddr_in *) 强制转换为 (struct sockaddr *)。

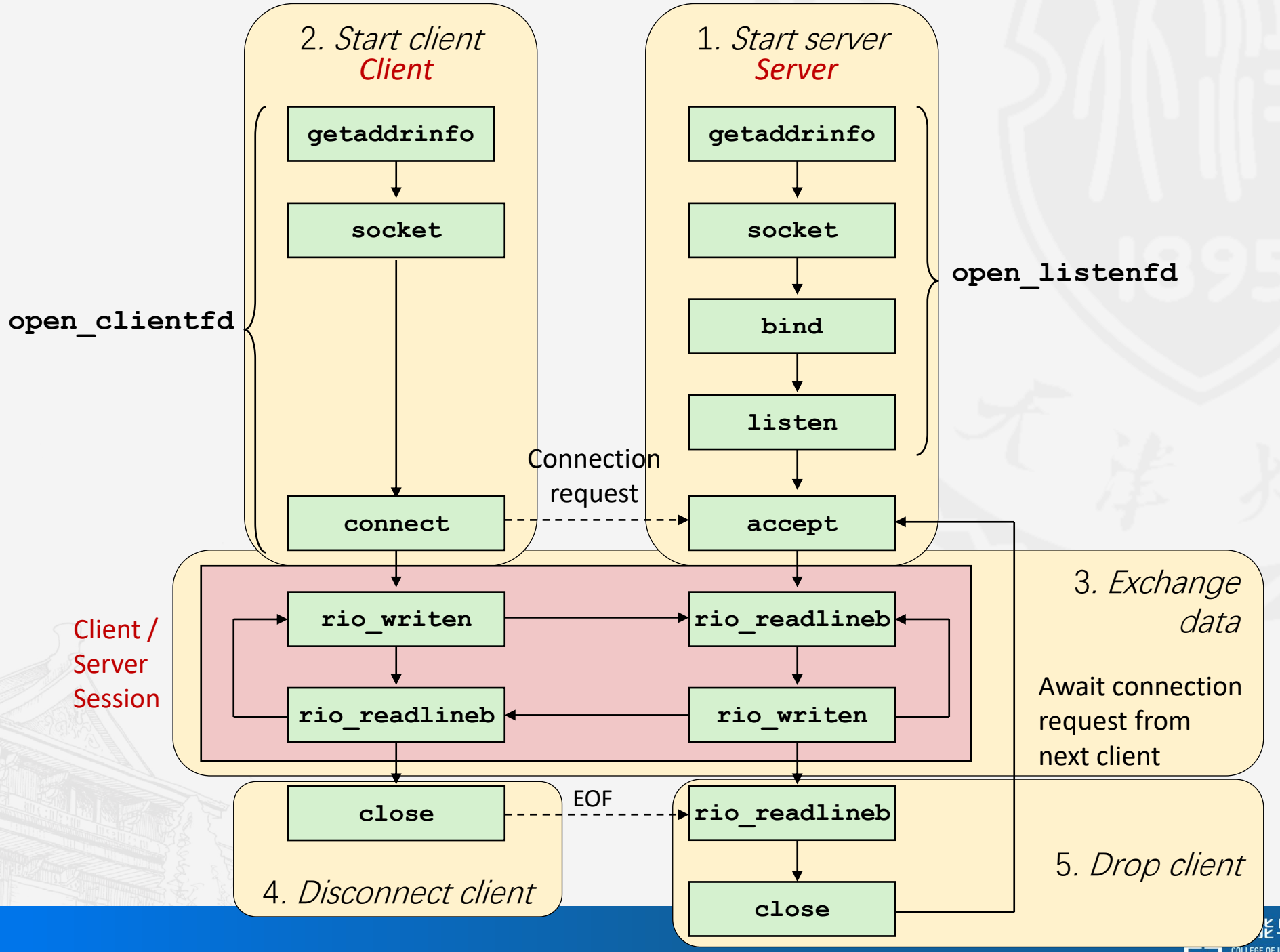
```
struct sockaddr_in  {
  uint16_t        sin_family;  /* Protocol family (always AF_INET) */
  uint16_t        sin_port;    /* Port num in network byte order */
  struct in_addr  sin_addr;    /* IP addr in network byte order */
  unsigned char   sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};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



套接字

Socket

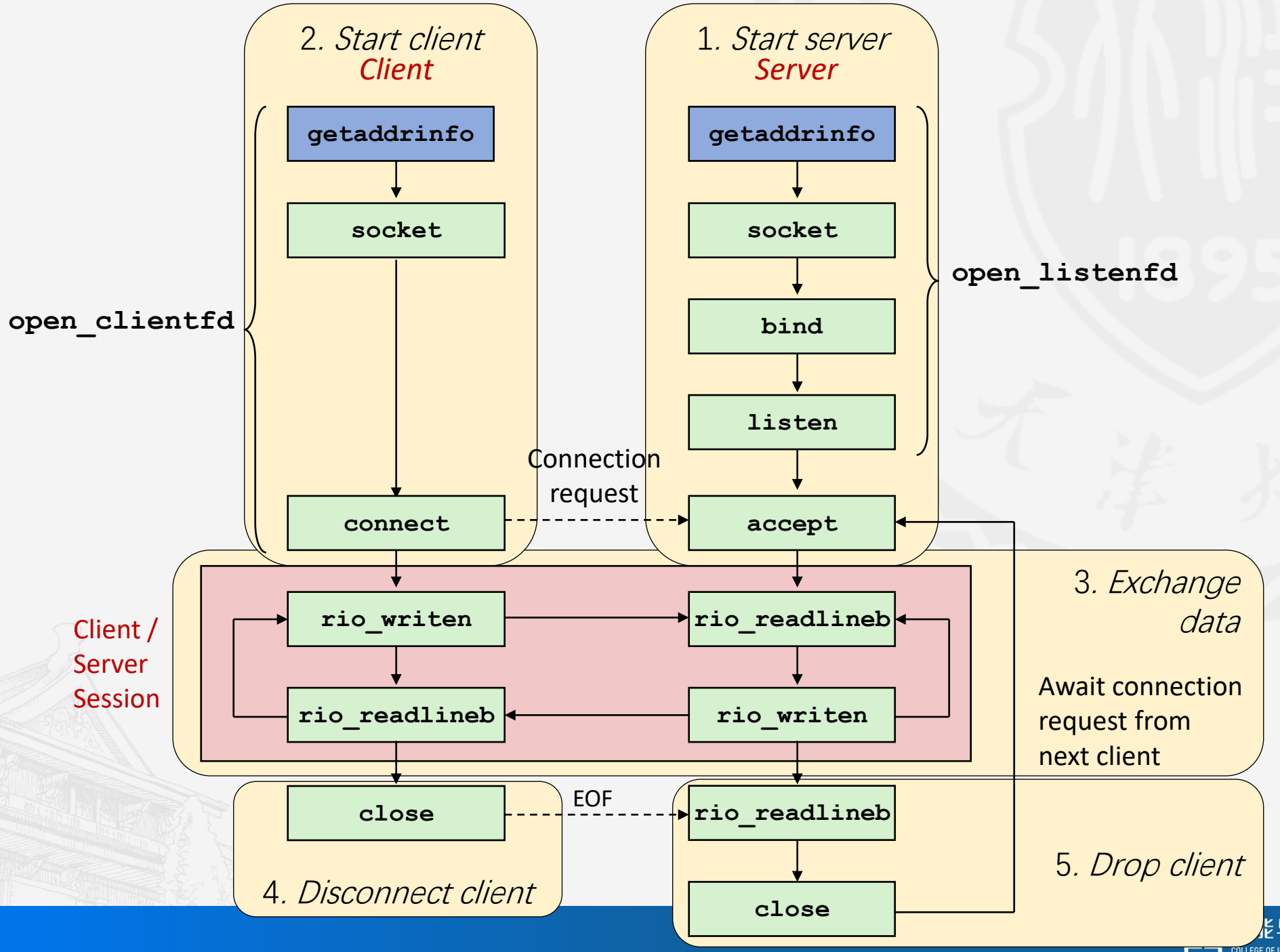
套接字接口



套接字

Socket

套接字接口



主机和服务转换：getaddrinfo

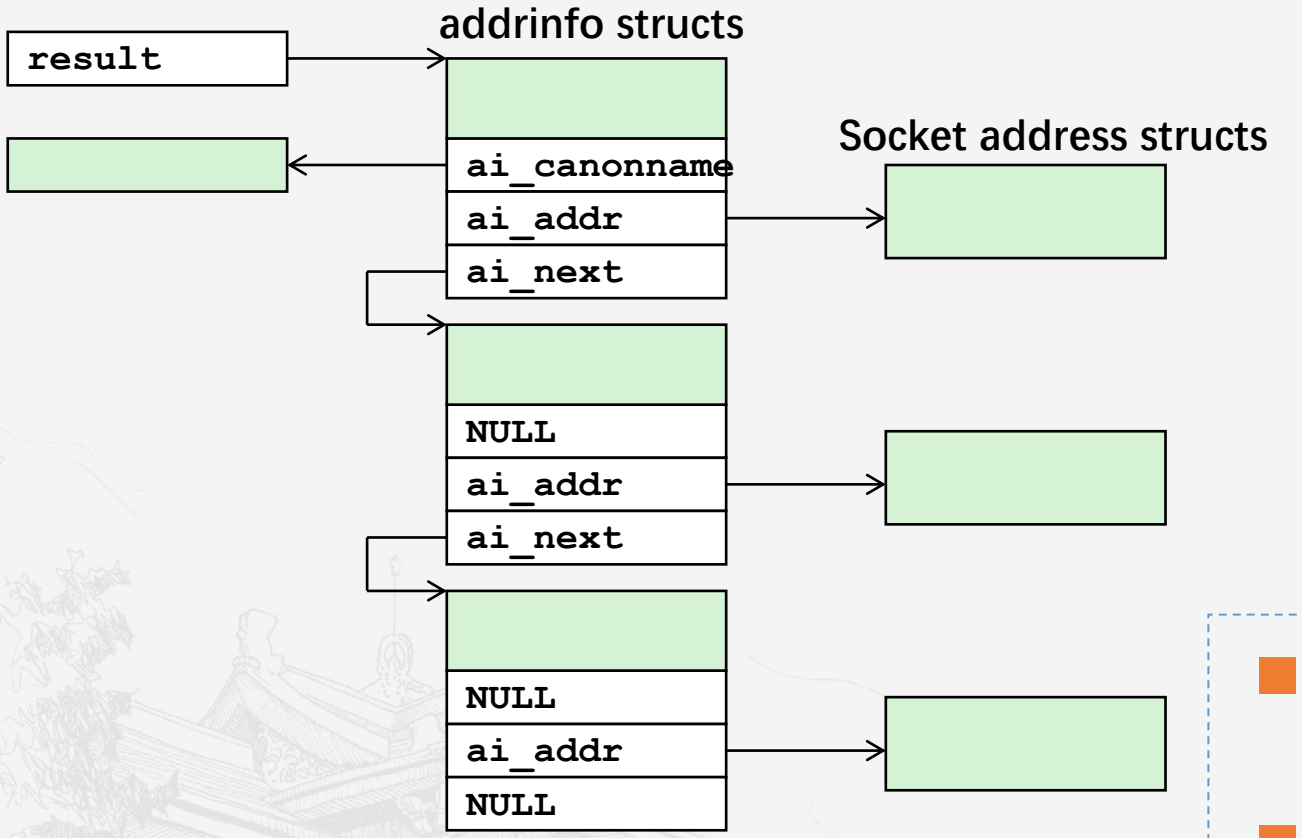
- getaddrinfo 是一种现代化的方法，用于将主机名、主机地址、端口和服务名称的字符串表示转换为套接字地址结构。它取代了过时的 gethostbyname 和 getservbyname 函数。
- 优点：
  - 可重入：可以安全地在多线程程序中使用。
  - 允许编写可移植、协议无关的代码：可以在不同的网络协议（如IPv4和IPv6）中工作。
- 缺点：
  - 复杂
  - 幸运的是，大多数情况下只需掌握少量的使用模式即可满足需求。

主机和服务转换：getaddrinfo（2）

```
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     */
```

- getaddrinfo 接受主机和服务作为输入参数，并返回一个指向 addrinfo 结构体的链表指针。每个 addrinfo 结构体都指向一个相应的套接字地址结构，并包含套接字接口函数的参数。
- 辅助函数：
  - freeaddrinfo：释放整个链表的内存。它用于释放由 getaddrinfo 返回的 addrinfo 链表，以防止内存泄漏
  - gai_strerror：将错误代码转换为错误消息。该函数返回与 getaddrinfo 相关的错误代码的文本描述，便于调试和处理错误

getaddrinfo返回的链表结构




```
struct addrinfo
{
  int ai_flags
  int ai_family
  int ai_socktype
  int ai_protocol
  socklen_t ai_addrlen;
  struct sockaddr *ai_addr
  char *ai_canonname;
  struct addrinfo *ai_next;
};
```

- 客户端：遍历该列表，依次尝试每个套接字地址，直到 socket 和 connect 调用成功为止。
- 服务器：遍历列表，直到 socket 和 bind 调用成功为止。

套接字

Socket

addrinfo结构体

```
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 */
};
```

- getaddrinfo 返回的每个 addrinfo 结构体都包含可以直接传递给 socket 函数的参数。
- 它还指向一个套接字地址结构，可以直接传递给 connect 和 bind 函数。

主机和服务转换：getnameinfo

```
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 */
```

- getnameinfo 是 getaddrinfo 的逆操作，它将套接字地址转换为对应的主机和服务名称。
  - 该函数取代了过时的 gethostbyaddr 和 getservbyport 函数
  - 可重入的、协议无关的

```c
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);
    }
    /* 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);
}
```

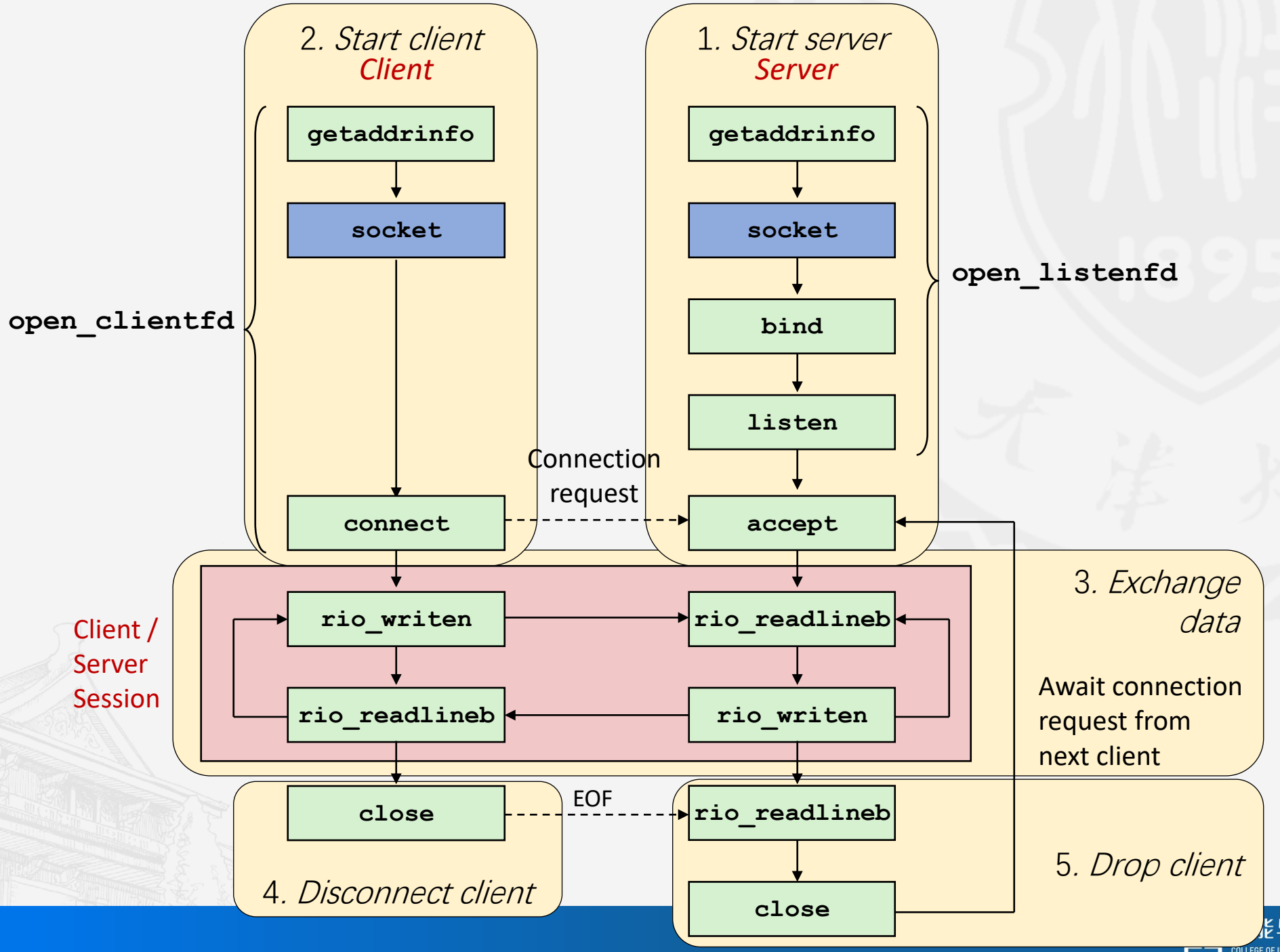
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

#define MAXLINE 1024
```

```
> ./hostinfo localhost
127.0.0.1

> ./hostinfo  www.tju.edu.cn
202.113.2.198

> ./hostinfo www.baidu.com
182.61.200.6
182.61.200.7
```

套接字

Socket

套接字接口



创建套接字：socket

- 客户端和服务器使用socket函数创建套接字描述符：

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

- 示例：

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

使用32位IPV4地址

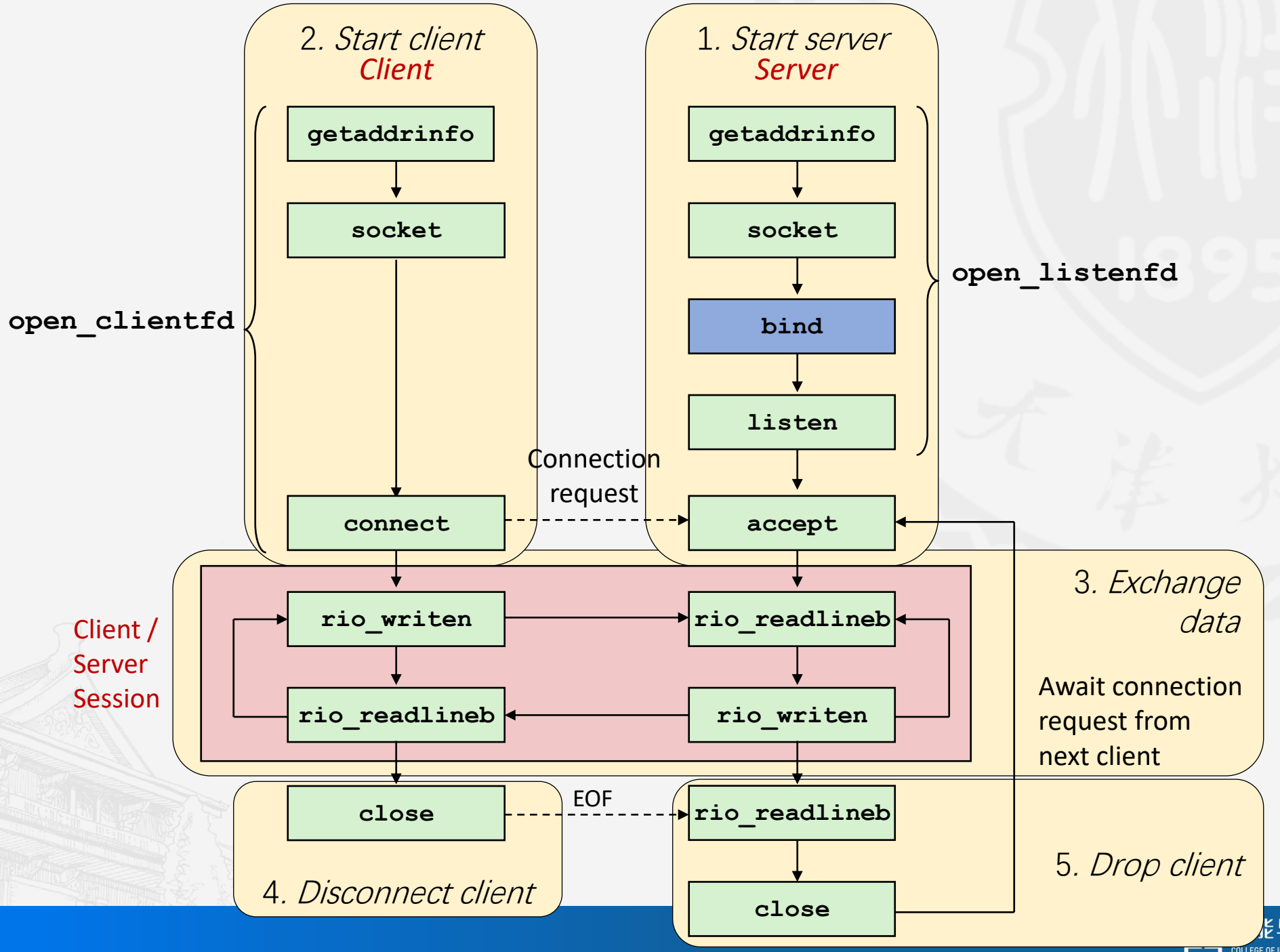
这个套接字是连接的一个端点
使用TCP传输协议

- 最佳做法是使用getaddrinfo自动生成参数，以使代码协议无关。

套接字

Socket

套接字接口



bind

- 服务器使用bind来请求内核将服务器的套接字地址与套接字描述符关联:

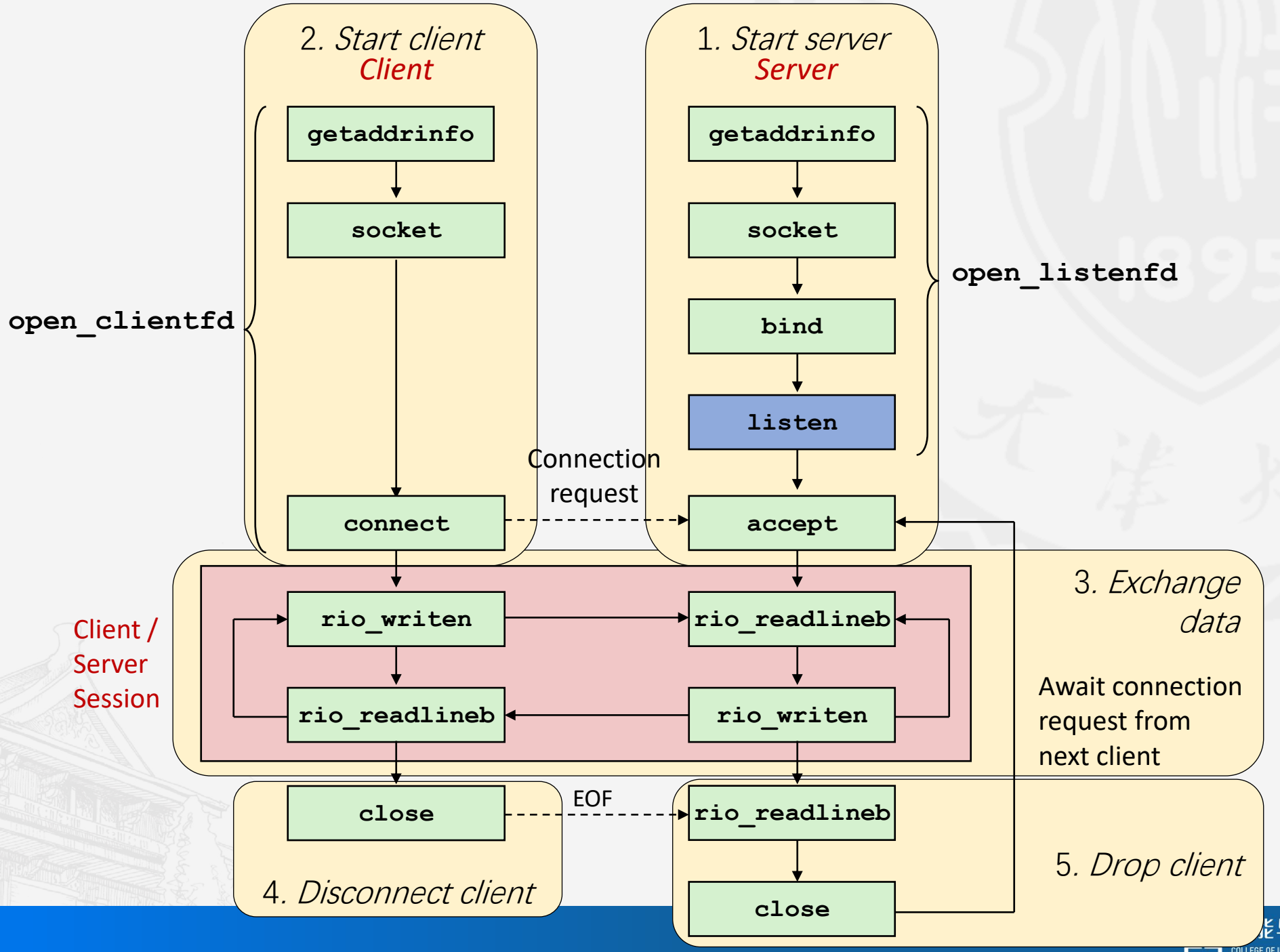
```
int bind(int sockfd, SA *addr, socklen_t addrlen);
```

- 该过程可以通过从描述符sockfd读取来读取到达地址addr的字节。
- 类似地，对sockfd的写入会沿着端点为addr的连接进行传输。
- 最佳做法是使用getaddrinfo来提供参数addr和addrlen。

套接字

Socket

套接字接口



listen

- 默认情况下，内核假定socket函数返回的描述符是一个主动套接字，将位于连接的客户端端点上。
- 服务器调用listen函数告诉内核，描述符将被服务器而不是客户端使用：

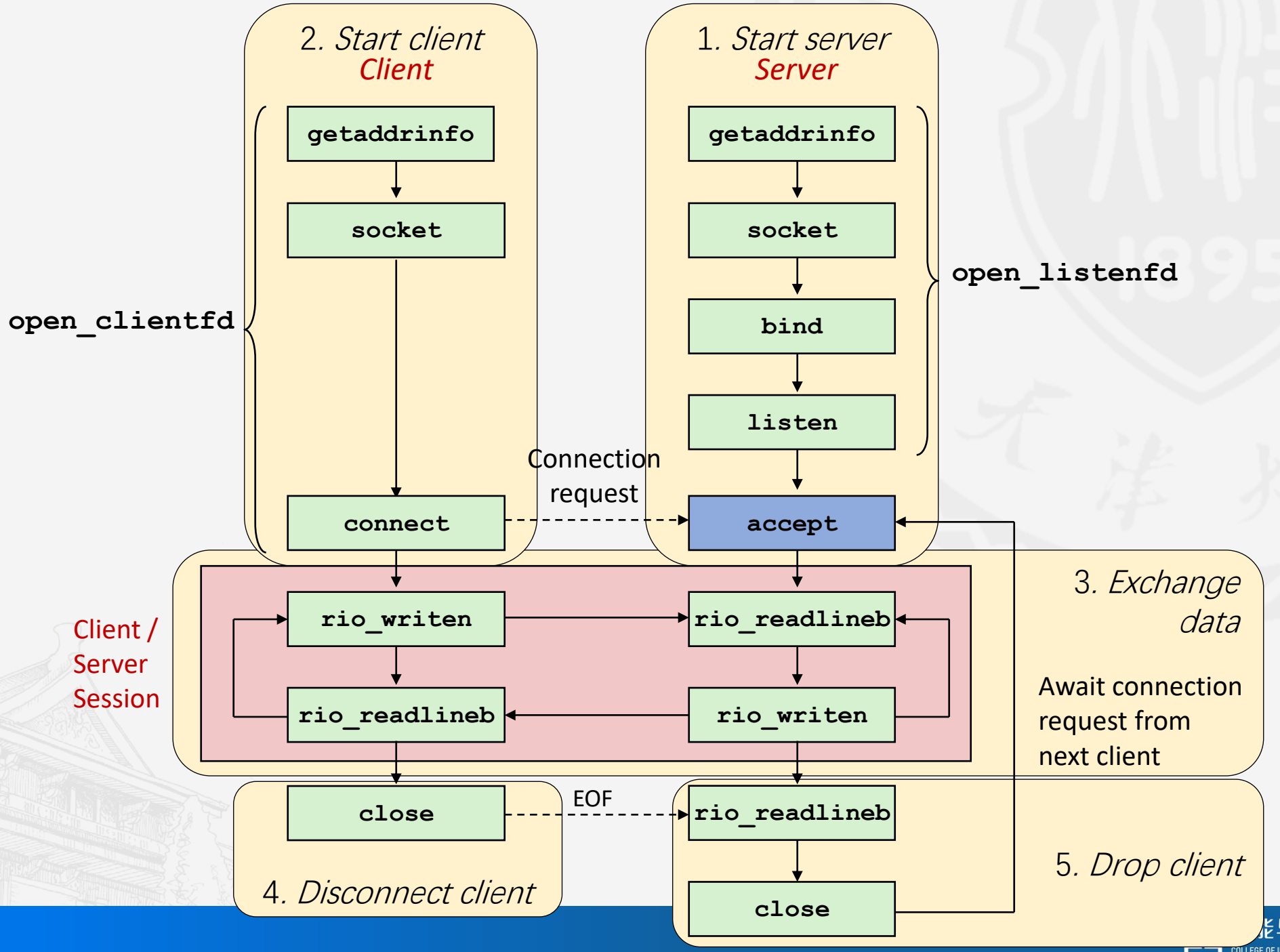
```
int listen(int sockfd, int backlog);
```

- 将sockfd从一个主动套接字转换为一个监听套接字，可以接受来自客户端的连接请求。
- backlog是关于内核在开始拒绝请求之前应该排队等待的未决连接请求数量的提示。

套接字

Socket

套接字接口



accept

- 服务器通过调用accept等待来自客户端的连接请求：

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

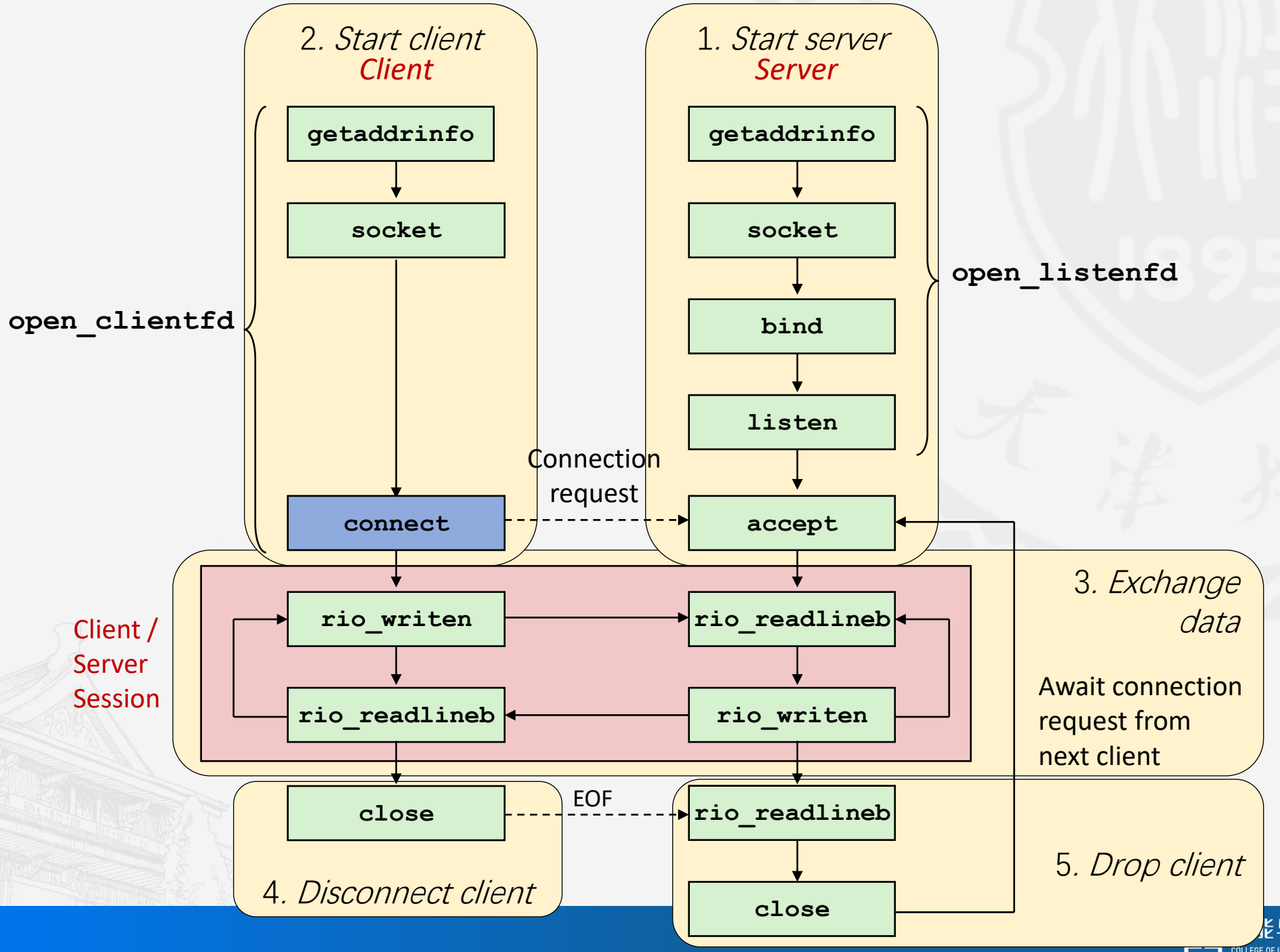
- 等待连接请求到达绑定到listenfd的连接上，然后在addr中填充客户端的套接字地址，并在addrlen中填充套接字地址的大小。

- 返回一个连接的描述符，可以通过Unix I/O与客户端通信。

套接字

Socket

套接字接口



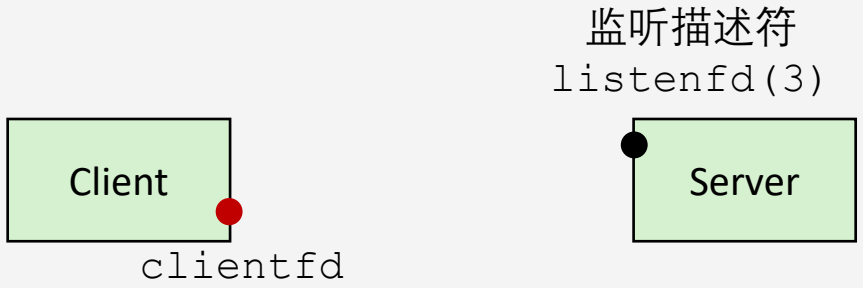
connect

- 客户端通过调用connect与服务器建立连接：

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

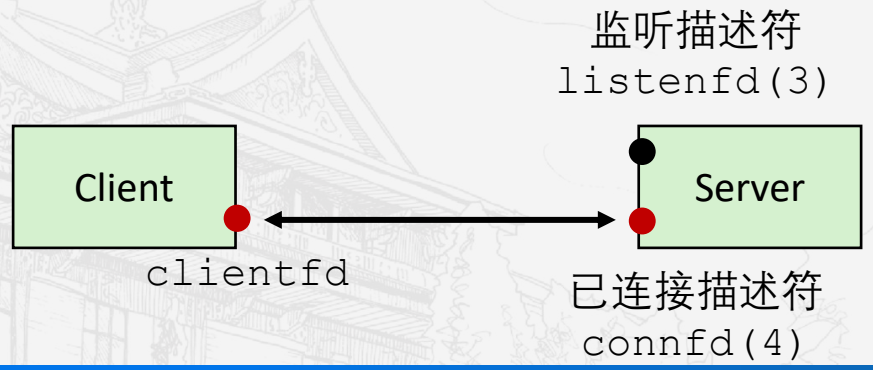
- 尝试与地址为addr的服务器建立连接。
  - 如果成功，则clientfd现在可以进行读写操作。
  - 生成的连接由套接字对来描述（addr.sin_addr：addr.sin_port）
    - x是客户端地址
    - y是在客户端主机上唯一标识客户端进程的临时端口
- 最佳做法是使用getaddrinfo来提供参数addr和addrlen。

对accept的进一步说明



1. 服务器在accept函数中阻塞，等待在监听描述符listenfd上的连接请求。

2. 客户端通过调用connect函数发起连接请求，并在此过程中阻塞。



3. 服务器从accept函数中返回connfd。客户端从connect函数中返回。现在clientfd和connfd之间建立了连接。

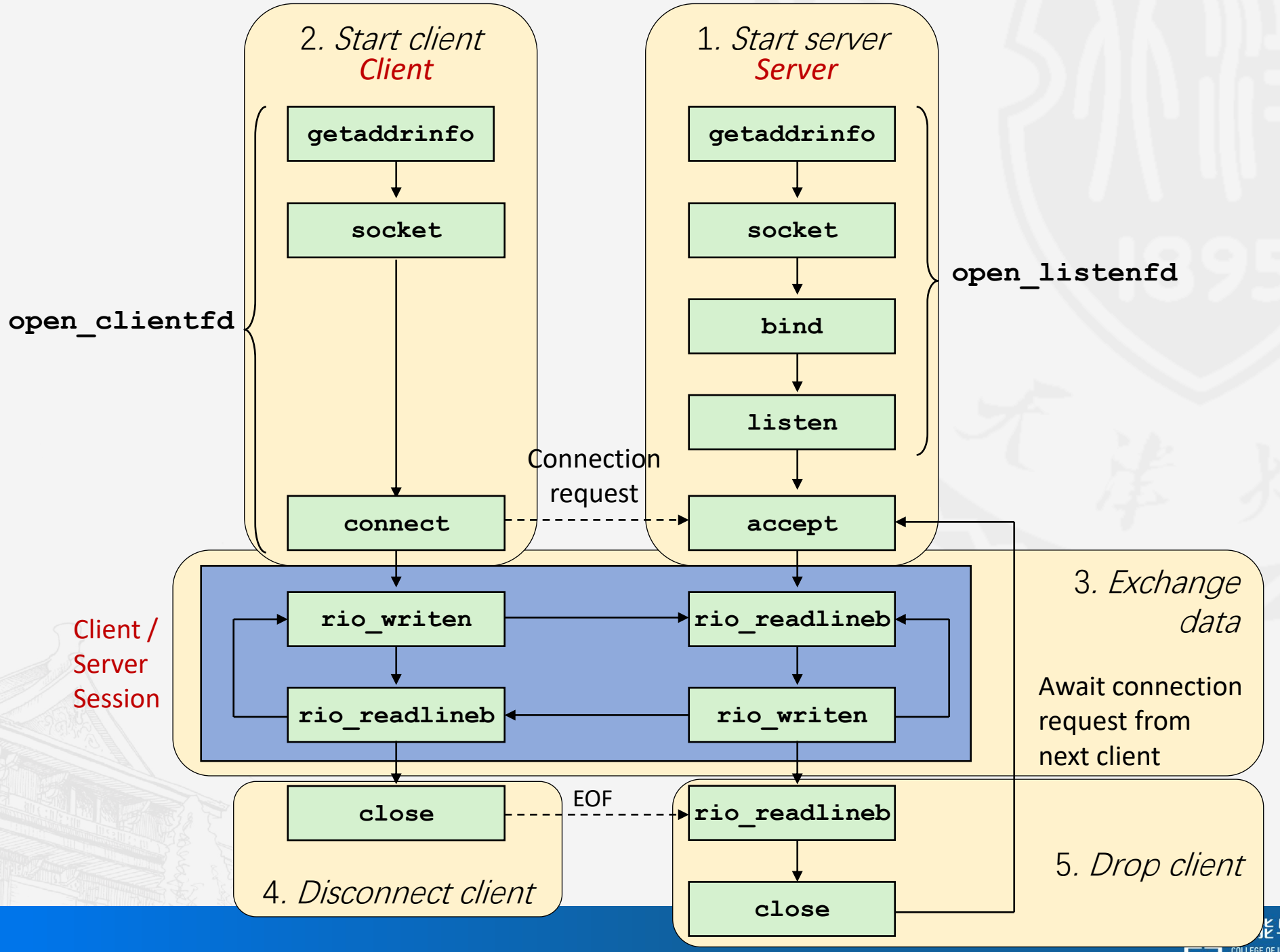
监听描述符和已连接描述符

- 监听描述符
  - 用于响应客户端连接请求的端点
  - 一次创建并在服务器的生命周期内存在
- 连接描述符
  - 客户端和服务器之间连接的端点
  - 每次服务器接受客户端连接请求时都会创建一个新的描述符
  - 仅在为客户端提供服务的时间段内存在
- 为什么要区分这两者?
  - 可以实现并发服务器，可以同时处理多个客户端连接。
  - 例如：每当接收到新的请求时，我们会创建一个子进程来处理该请求。

套接字

Socket

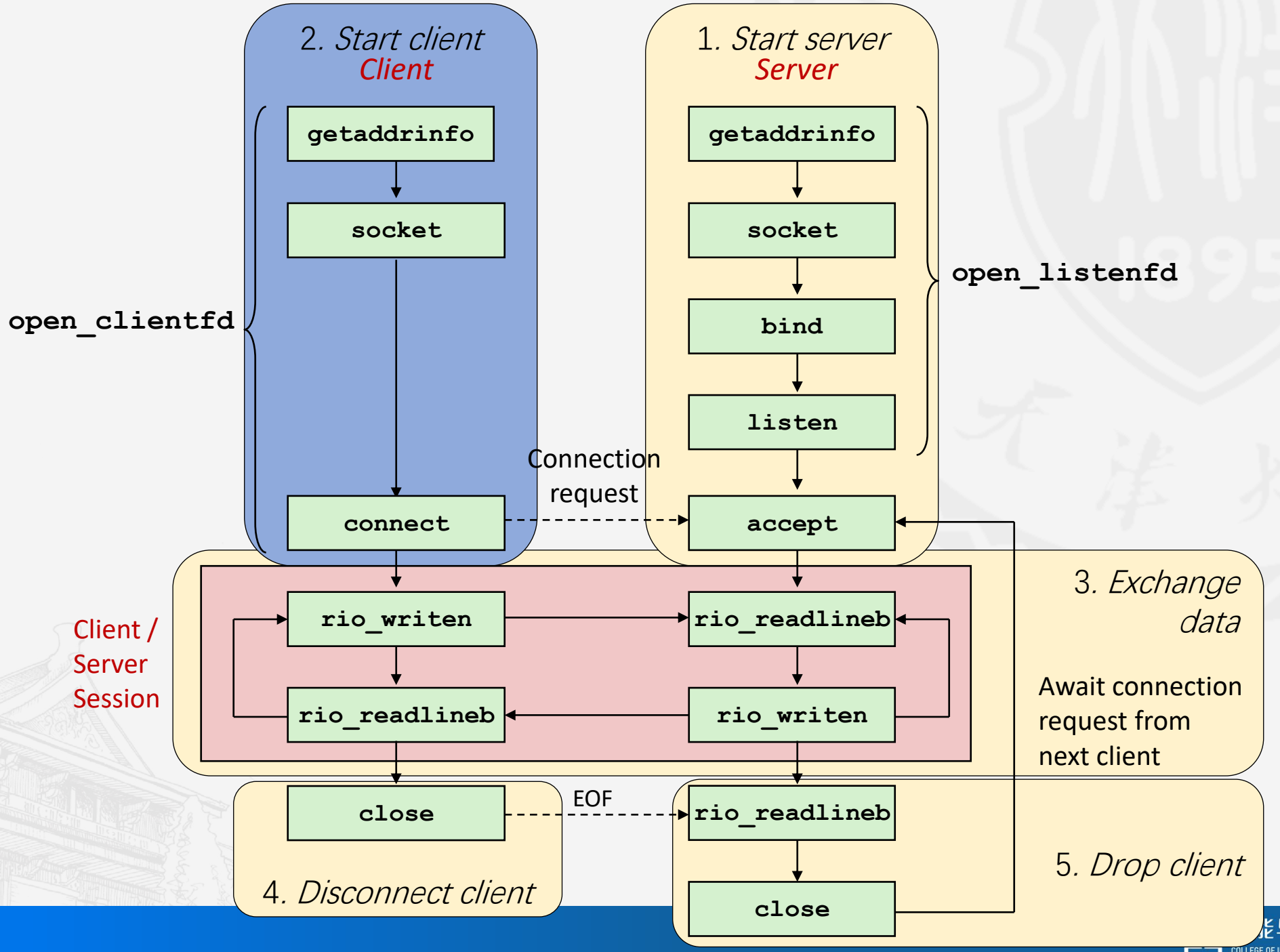
套接字接口



套接字

Socket

套接字接口



套接字接口辅助函数open_clientfd

建立与服务器的连接

```
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);
```

套接字接口辅助函数open_clientfd (2)

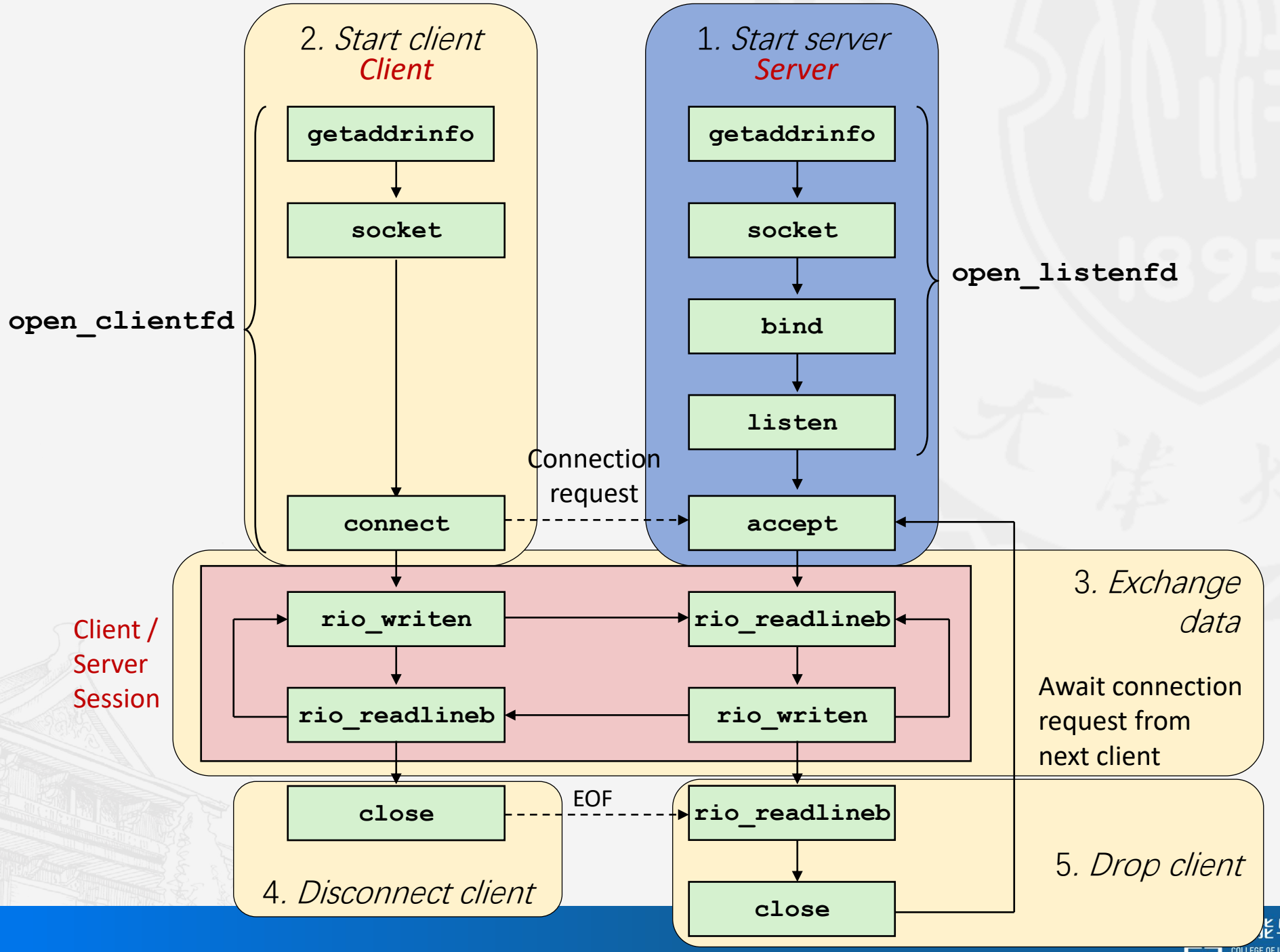
```
    /* Walk the list for one that we can successfully connect to */
    for (p = listp; p; p = p->ai_next) {
        /* Create a socket descriptor */
        if ((clientfd = socket(p->ai_family, p->ai_socktype,
                               p->ai_protocol)) < 0)
            continue; /* Socket failed,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

套接字

Socket

套接字接口



套接字接口辅助函数open_listenfd

创建一个监听描述符，用于接受来自客户端的连接请求。

```
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);
```

套接字接口辅助函数open_listenfd (2)

```
/* 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 */
}
```

套接字接口辅助函数open_listenfd (3)

```
    /* 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

请注意：open_clientfd和open_listenfd这两个都与IP协议无关。

都是通过getaddrinfo生成的socket参数

套接字

Socket

示例：echo客户端

```c
#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

套接字

Socket

示例：echo服务器

main函数

```c
#include "csapp.h"
void echo(int connfd);

int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr; /* Enough room for any addr */
    char client_hostname[MAXLINE], client_port[MAXLINE];

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage); /* Important! */
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        Getnameinfo((SA *) &clientaddr, clientlen,
                    client_hostname, MAXLINE, client_port, MAXLINE, 0);
        printf("Connected to (%s, %s)\n", client_hostname, client_port);
        echo(connfd);
        Close(connfd);
    }
    exit(0);
}
```

echoserveri.c

Echo服务器：echo函数

- 服务器使用RIO读取并回显文本行，直到遇到EOF（文件结束）条件
  - EOF条件由客户端调用close(clientfd)触发

```
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

使用telnet工具测试服务器

- telnet程序对于测试通过Internet连接传输ASCII字符串的服务器非常有用
  - 课程内实现的echo服务器
  - Web服务器
  - 邮件服务器。

- 用法：`linux> telnet <host> <portnumber>`
  - 在本地创建一个在<host>上并监听端口<portnumber>的服务器的连接。

使用telnet工具测试服务器（2）

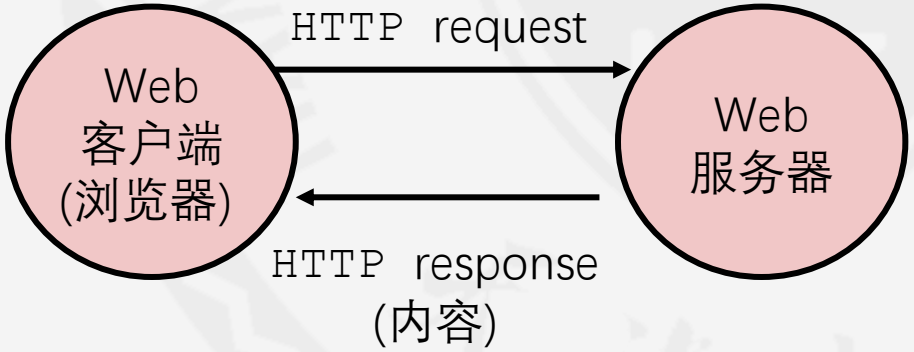
```
server> ./echoserveri 15213
Connected to (MAKOSHARK.ICS.CS.CMU.EDU, 50280)
server received 11 bytes
server received 8 bytes



clinet> 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.
```

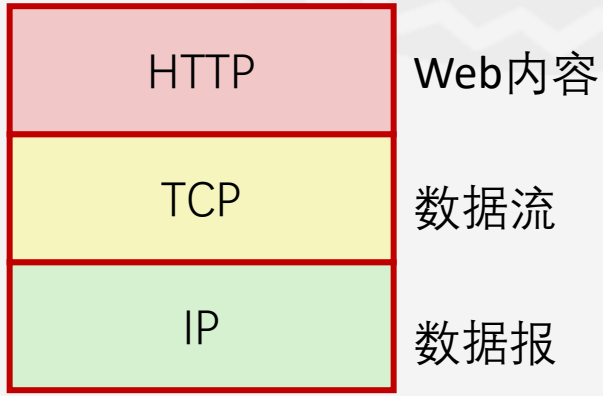
套接字

Socket

- 客户端和服务器使用超文本传输协议（HTTP）进行通信。
  - 客户端和服务器建立TCP连接。
  - 客户端请求内容。
  - 服务器以请求的内容作出响应。
  - 客户端和服务器关闭连接（最终）。
- 当前流行的版本：
  - HTTP/1.1　　RFC 2616，1999年6月
  - HTTP/2.0　　RFC 7540，2015年5月
  - RFC：Request for Comments

Web服务





Web内容

- Web服务器向客户端返回内容
  - 内容：MIME（Multipurpose Internet Mail Extensions）类型相关的数据
- 常见MIME类型
  - text/html HTML文档
  - text/plain 未格式化的文本
  - image/gif 以GIF格式编码的二进制图像
  - image/png 以PNG格式编码的二进制图像
  - image/jpeg 以JPEG格式编码的二进制图像
- 更多信息见　http://www.iana.org/assignments/media-types/media-types.xhtml

静态和动态内容

- HTTP响应中返回的内容可以是静态的或动态的。
  - 静态内容：存储在文件中，并在HTTP请求的响应中检索到的内容。
    - 例如：HTML文件、图像、音频片段。
    - 请求标识出哪个内容文件。
  - 动态内容：根据HTTP请求实时生成的内容。
    - 例如：由服务器代表客户端执行的程序产生的内容。
    - 请求标识包含可执行代码的文件。
- 总之：Web内容与由服务器管理的文件相关联。

通用资源定位符：URL

- 每条由Web服务器返回的内容都是和它管理的某个文件相关联的，文件的唯一名称是URL（Universal Resource Locator）
- 示例：：http://www.cmu.edu:80/index.html
- 客户端使用前缀（http://www.cmu.edu:80）来推断：
  - 要连接的服务类型（协议）：HTTP
  - 服务器位置：www.cmu.edu
  - 正在侦听的端口：80
- 服务器使用后缀（/index.html）：
  - 确定请求是针对静态内容还是动态内容
    - 没有硬性规则；一种约定：可执行文件位于cgi-bin目录中
  - 在文件系统中找到文件
    - 后缀中的初始“/”表示请求内容的主目录。
    - 最小后缀是“/”，服务器将其扩展为配置的默认文件名（通常为index.html）

HTTP请求

- HTTP请求由请求行和零个或多个请求头组成。
- 请求行格式：<method> <uri> <version>
  - <method> 是以下之一：GET、POST、OPTIONS、HEAD、PUT、DELETE或TRACE。
  - <uri> 对于代理服务器通常是URL，对于服务器通常是URL的后缀
    - URL是URI（统一资源标识符）的一种类型
    - 参见http://www.ietf.org/rfc/rfc2396.txt。
  - <version> 是请求的HTTP版本（HTTP/1.0或HTTP/1.1）。
- 请求头格式：<header name>: <header data>
  - 提供额外的信息给服务器。

HTTP响应

- HTTP响应由响应行和零个或多个响应头组成，可能后跟内容，在头部和内容之间由空行（"\r\n"）分隔。
- 响应行格式：<version> <status code> <status msg>
  - <version> 是响应的HTTP版本。
  - <status code> 是数字状态代码。
  - <status msg> 是相应的英文文本。
    - 200 OK：请求已成功处理。
    - 301 Moved：提供替代URL。
    - 404 Not Found：服务器找不到文件。

HTTP响应（2）

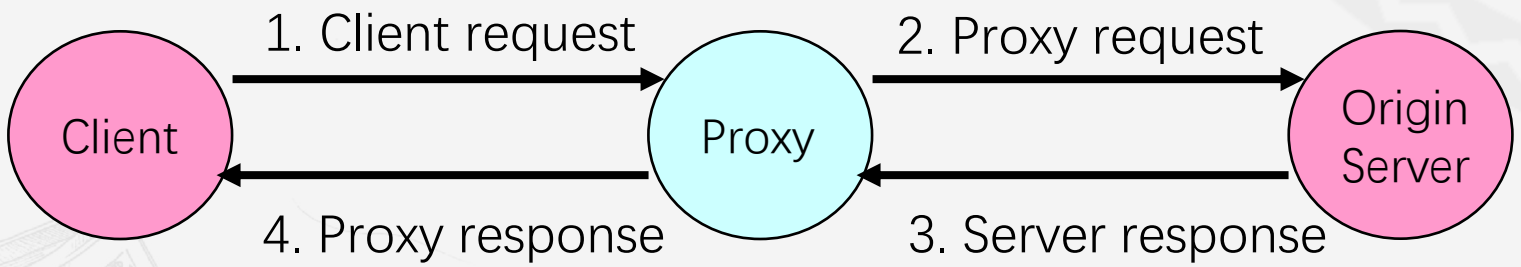
- 响应头格式：<header name>: <header data>
  - 提供有关响应的额外信息。
  - Content-Type：响应主体中内容的MIME类型。
  - Content-Length：响应主体中内容的长度。

示例：HTTP事务

```
> telnet www.tju.edu.cn 80                                客户端：打开一个服务器的连接
Trying 202.113.2.198...                                   Telnet在终端显示3行信息
Connected to www.tju.edu.cn.
Escape character is '^]'.
GET / HTTP/1.1                                            客户端：请求行
Host: www.tju.edu.cn                                      客户端：HTTP/1.1的请求头
                                                          客户端：使用一个空行结束请求头
HTTP/1.1 200 OK                                           服务器：响应行
Date: Mon, 22 Apr 2024 08:56:51 GMT                       服务器：响应头
Server: *********
Content-Length: 42811
Content-Type: text/html
Content-Language: zh-CN
Set-Cookie: UqZBpD3n3iXPAw1X=v1olI9JQSDjSO; Expires=Mon, 22-Apr-2024 20:58:30 GMT; Path=/
                                                          服务器：空行表示响应头的结束
<!DOCTYPE html>                                           服务器：响应内容
<html>
...
</html>                                                   服务器：响应内容结束
Connection closed by foreign host.                        服务器：关闭连接
```

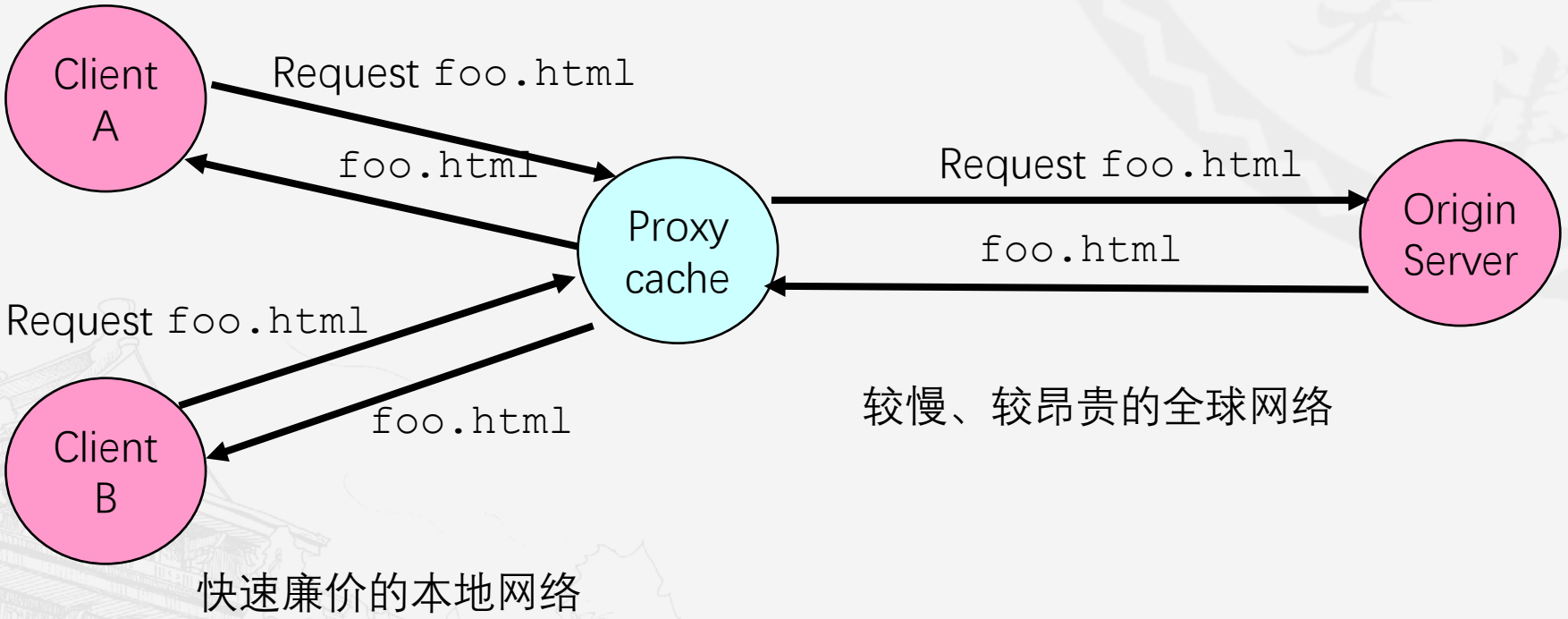
代理服务器

- 代理是客户端和源服务器之间的中间人。
  - 对于客户端，代理行为类似于服务器。
  - 对于服务器，代理行为类似于客户端。



为什么要使用代理服务器

- 可以在请求和响应经过时执行有用的功能。
  - 例如：缓存、日志记录、匿名化、过滤、转码



Tiny Web服务器

- 教材中提供了一个Tiny Web服务器
  - 一个顺序执行的Web服务
  - 可以向真实浏览器提供静态和动态内容
    - 包括：文本文件、HTML文件、GIF、PNG和JPEG图像
  - 239行C代码，带注释
  - 没有真正的Web服务器完整和健壮
    - 您可以使用格式不正确的HTTP请求（例如，使用 “\n”作为行的结束符，而不是“\r\n”）来破坏它

Tiny Web的功能

- 接受来自客户端的连接
- 从客户端（通过连接的套接字）读取请求
- 拆分为<method> <uri> <version>
  - 如果方法不是GET，则返回错误
- 如果URI包含“cgi-bin”，则提供动态内容
  - 如果有文件名为“abcgi-bingo.html”则会执行错误的操作
  - 通过Fork系统调用创建进程来执行程序
- 否则提供静态内容
  - 将文件复制到输出

Tiny Web的静态内容服务

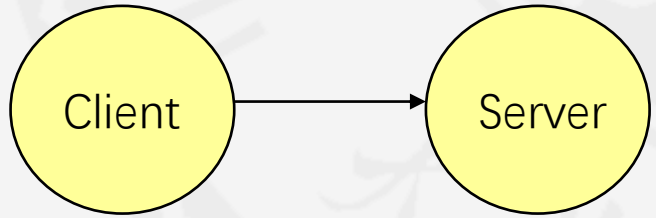
```c
void serve_static(int fd, char *filename, int filesize)
{
    int srcfd;
    char *srcp, filetype[MAXLINE], buf[MAXBUF];

    /* Send response headers to client */
    get_filetype(filename, filetype);
    sprintf(buf, "HTTP/1.0 200 OK\r\n");
    sprintf(buf, "%sServer: Tiny Web Server\r\n", buf);
    sprintf(buf, "%sConnection: close\r\n", buf);
    sprintf(buf, "%sContent-length: %d\r\n", buf, filesize);
    sprintf(buf, "%sContent-type: %s\r\n\r\n", buf, filetype);
    Rio_writen(fd, buf, strlen(buf));

    /* Send response body to client */
    srcfd = Open(filename, O_RDONLY, 0);
    srcp = Mmap(0, filesize, PROT_READ, MAP_PRIVATE, srcfd, 0);
    Close(srcfd);
    Rio_writen(fd, srcp, filesize);
    Munmap(srcp, filesize);
}
```

tiny.c

Tiny Web的动态内容服务

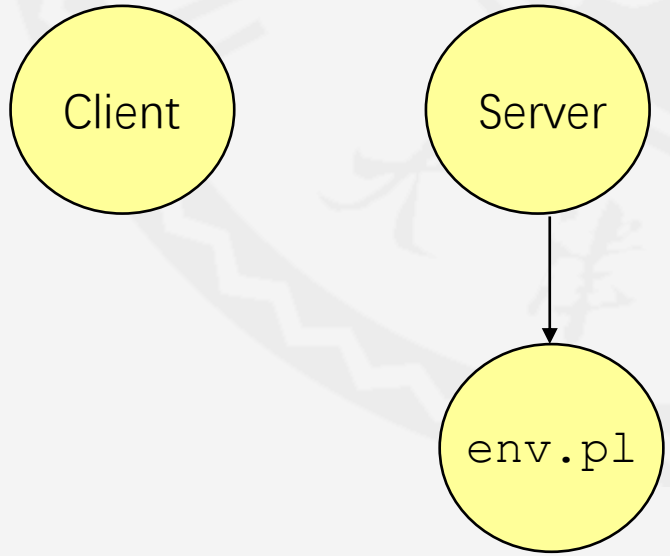
- 客户端向服务器发送请求
- 如果请求的URI包含字符串“/cgi-bin”，则Tiny Web服务假定该请求是针对动态内容的

```
GET /cgi-bin/env.pl HTTP/1.1
```



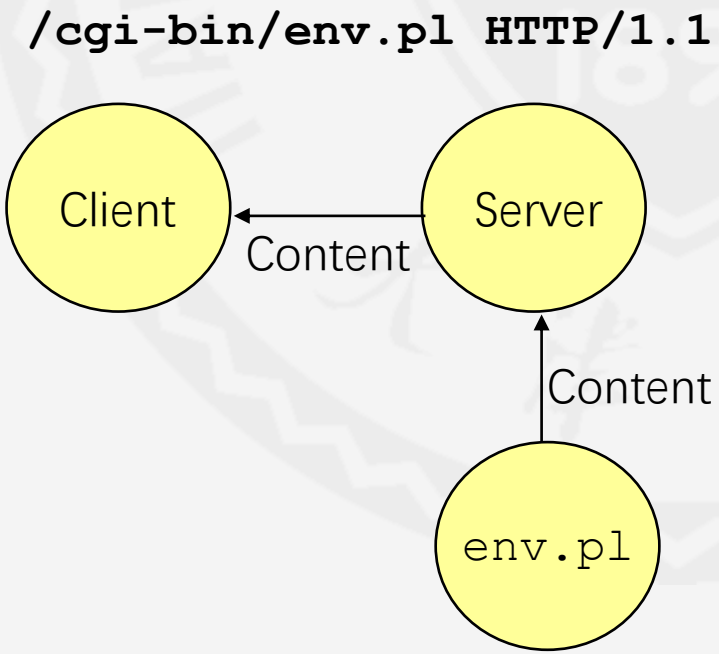
Tiny Web的动态内容服务（2）

- 服务器创建一个子进程，并在该进程中运行由URI标识的程序



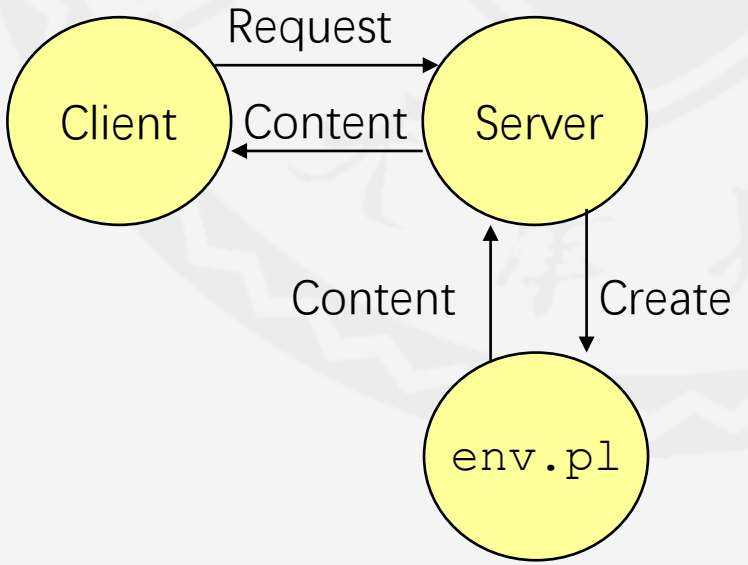
Tiny Web的动态内容服务（3）

- 子进程运行并生成动态内容
- 服务器捕获子进程的内容，并将其原样转发给客户端



动态内容服务的问题

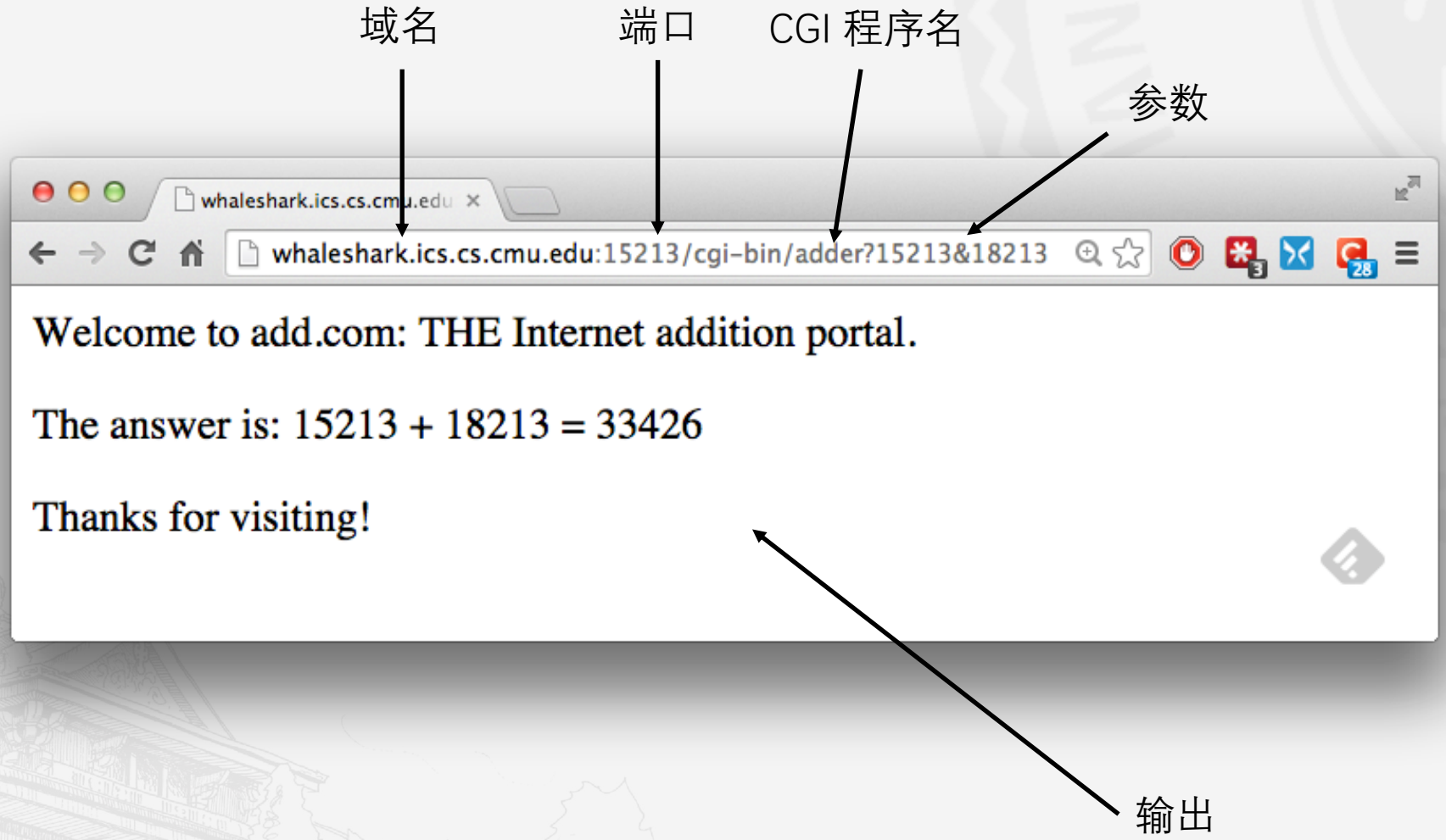
- 客户端如何将程序参数传递给服务器?
- 服务器如何将这些参数传递给子进程?
- 服务器如何将与请求相关的其他信息传递给子进程?
- 服务器如何捕获子进程生成的内容?
- 这些问题由**通用网关接口（Common Gateway Interface, CGI）**规范解决。



通用网关接口：CGI

- 子进程是根据CGI规范编写的，因此它们通常被称为CGI程序。
- CGI定义了一个相当简单的标准，用于在客户端（浏览器）、服务器和子进程之间传输信息。
- CGI是动态生成内容的原始标准。它已经在很大程度上被其他更快的技术取代：
  - 例如：fastCGI、Apache模块、Java servlets、Rails控制器。
  - 避免了动态创建进程（昂贵且缓慢）

一个提供加法服务的cgi程序



为GET方法提供一个动态内容服务

- **问题：**客户端如何将参数传递给服务器?
- **解决方案：**参数附加在URI后面

- 可以直接在浏览器中键入的URL中 或 HTML链接中的URL中进行编码
  - http://add.com/cgi-bin/adder?15213&18213
  - adder是服务器上将执行加法的CGI程序。
  - 参数列表以“？”开头
  - 参数由“&”分隔
  - 空格由“+”或“%20”表示

为GET方法提供一个动态内容服务（2）

- URL后缀： **cgi-bin/adder?15213&18213**
- 显示在浏览器上的结果

```
Welcome to add.com: THE Internet addition portal.

The answer is: 15213 + 18213 = 33426

Thanks for visiting!
```

为GET方法提供一个动态内容服务（3）

- **问题：**服务器如何将这些参数传递给子进程?
- **解决方案：**通过环境变量QUERY_STRING
  - 一个包含“?”后面所有内容的字符串
  - 对于add程序：QUERY_STRING = “15213&18213”

```
    /* Extract the two arguments */
    if ((buf = getenv("QUERY_STRING")) != NULL) {
        p = strchr(buf, '&');
        *p = '\0';
        strcpy(arg1, buf);
        strcpy(arg2, p+1);
        n1 = atoi(arg1);
        n2 = atoi(arg2);
    }
```

为GET方法提供一个动态内容服务（4）

- **问题：**服务器如何捕获子进程生成的内容?
- **答案：**子进程在标准输出(stdout)上生成其输出。服务器使用dup2将stdout重定向到其连接的套接字。

```
void serve_dynamic(int fd, char *filename, char *cgiargs)
{
    char buf[MAXLINE], *emptylist[] = { NULL };

    /* Return first part of HTTP response */
    sprintf(buf, "HTTP/1.0 200 OK\r\n");
    Rio_writen(fd, buf, strlen(buf));
    sprintf(buf, "Server: Tiny Web Server\r\n");
    Rio_writen(fd, buf, strlen(buf));

    if (Fork() == 0) { /* Child */
        /* Real server would set all CGI vars here */
        setenv("QUERY_STRING", cgiargs, 1);
        Dup2(fd, STDOUT_FILENO);          /* Redirect stdout to client */
        Execve(filename, emptylist, environ); /* Run CGI program */
    }
    Wait(NULL); /* Parent waits for and reaps child */
}
```

为GET方法提供一个动态内容服务（5）

- 注意，只有CGI子进程知道内容的类型和长度，因此它必须生成这些头部。

```
    /* Make the response body */
    sprintf(content, "Welcome to add.com: ");
    sprintf(content, "%sTHE Internet addition portal.\r\n<p>", content);
    sprintf(content, "%sThe answer is: %d + %d = %d\r\n<p>",
            content, n1, n2, n1 + n2);
    sprintf(content, "%sThanks for visiting!\r\n", content);

    /* Generate the HTTP response */
    printf("Content-length: %d\r\n", (int)strlen(content));
    printf("Content-type: text/html\r\n\r\n");
    printf("%s", content);
    fflush(stdout);

    exit(0);
```

为GET方法提供一个动态内容服务（6）

```
linux> 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 '^]'.
GET /cgi-bin/adder?15213&18213 HTTP/1.0
```

客户端发出HTTP请求

```
HTTP/1.0 200 OK
Server: Tiny Web Server
Connection: close
```

服务器生成的HTTP响应

```
Content-length: 117
Content-type: text/html

Welcome to add.com: THE Internet addition portal.
<p>The answer is: 15213 + 18213 = 33426
<p>Thanks for visiting!
```

CGI程序生成的HTTP响应

```
Connection closed by foreign host.
linux>
```

参考资料

- 网络编程
  - W. Richard Stevens et. al. “Unix Network Programming: The Sockets Networking API”, Volume 1, Third Edition, Prentice Hall, 2003
- Linux编程
  - Michael Kerrisk, “The Linux Programming Interface”, No Starch Press, 2010