Lab7 基于Socket接口实现自定义协议通信

Dr. Xiqun Lu
College of Computer Science
Zhejiang University

什么是 socket？[1]

- socket 的原意是“插座”，在计算机通信领域，socket 被翻译为“套接字”，它是计算机之间进行通信的一种约定或一种方式。通过 socket 这种约定，一台计算机可以接收其他计算机的数据，也可以向其他计算机发送数据。
  - 为了与远程计算机进行数据传输，需要连接到Internet，而 socket 就是用来连接到Internet的工具。

- UNIX/Linux 中的 socket 是什么？
  - 在 UNIX/Linux 系统中，为了统一对各种硬件的操作，简化接口，不同的硬件设备也都被看成一个文件。对这些文件的操作，等同于对磁盘上普通文件的操作。
    - 通常用 0 来表示标准输入文件（stdin），它对应的硬件设备就是键盘；
    - 通常用 1 来表示标准输出文件（stdout），它对应的硬件设备就是显示器。
  - 为了表示和区分已经打开的文件，UNIX/Linux 会给每个文件分配一个 ID，这个 ID 就是一个整数，被称为文件描述符（File Descriptor）。
  - 网络连接也是一个文件，它也有文件描述符！
  - 我们可以通过 socket() 函数来创建一个网络连接，或者说打开一个网络文件，socket() 的返回值就是文件描述符。
    - 有了文件描述符，我们就可以使用普通的文件操作函数来传输数据了
    - 用 read() 读取从远程计算机传来的数据；
    - 用 write() 向远程计算机写入数据。

什么是 socket?

- Window 系统中的 socket 是什么？
  - Windows 也有类似“文件描述符”的概念，但通常被称为“文件句柄”。
  - 与 UNIX/Linux 不同的是，Windows 会区分 socket 和文件，Windows 就把 socket 当做一个网络连接来对待，因此需要调用专门针对 socket 而设计的数据传输函数，针对普通文件的输入输出函数就无效了。

- 这个世界上有很多种套接字（socket）！
  - Internet套接字，根据数据的传输方式，可以将 Internet 套接字分成两种类型：**流格式套接字（SOCK_STREAM）**和**数据报格式套接字（SOCK_DGRAM）**

- **流格式套接字（SOCK_STREAM）**
  - 数据在传输过程中不会消失；
  - 数据是按照顺序传输的；
  - 数据的发送和接收不是同步的 (流格式套接字的内部有一个缓冲区（也就是字符数组），通过 socket 传输的数据将保存到这个缓冲区。接收端在收到数据后并不一定立即读取，只要数据不超过缓冲区的容量，接收端有可能在缓冲区被填满以后一次性地读取，也可能分成好几次读取。)
  - Example: HTTP (TCP)

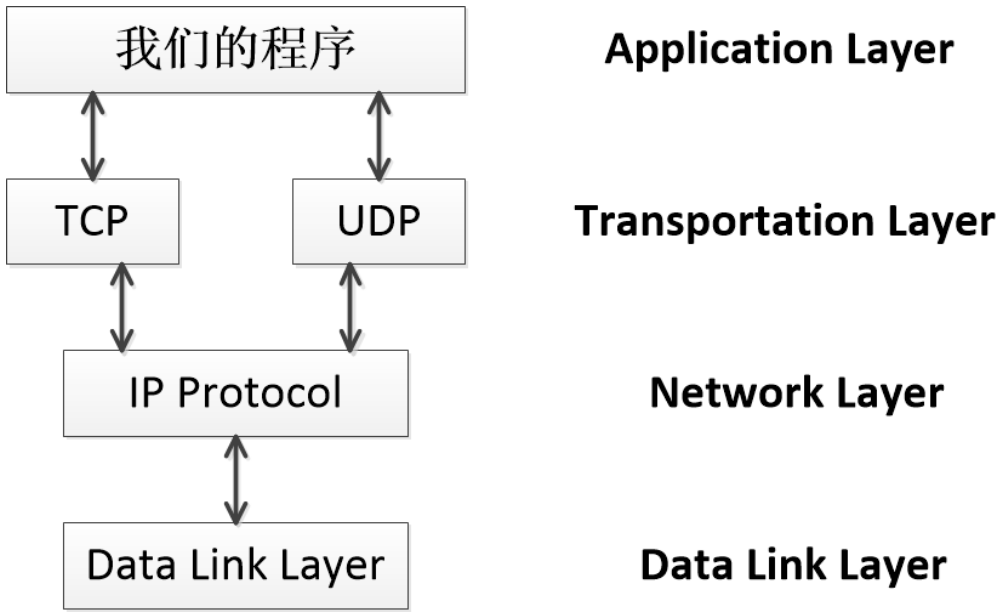
什么是 socket？

- **数据报格式套接字**（SOCK_DGRAM）计算机只管传输数据，不作数据校验，如果数据在传输中损坏，或者没有到达另一台计算机，是没有办法补救的。也就是说，数据错了就错了，无法重传。
  - 强调快速传输而非传输顺序；
  - 传输的数据可能丢失也可能损毁；
  - 限制每次传输的数据大小；
  - 数据的发送和接收是同步的
  - Example: QQ视频和语音聊天 (UDP)

- 我们所说的 socket 编程，是站在**传输层**的基础上，所以可以使用 TCP/UDP 协议。

TCP/IP

- 目前实际使用的网络模型是 TCP/IP 模型，它对 OSI 模型进行了简化，只包含了四层，从上到下分别是**应用层**、**传输层**、**网络层**和**链路层**（网络接口层），每一层都包含了若干协议。
- TCP/IP 模型包含了 TCP、IP、UDP、Telnet、FTP、SMTP 等上百个互为关联的协议，其中 TCP 和 IP 是最常用的两种底层协议，所以把它们统称为“TCP/IP 协议族”。
-



IP 地址、MAC 地址和端口号

- 在茫茫的互联网海洋中，要找到一台计算机非常不容易，有三个要素必须具备，它们分别是 IP 地址、MAC 地址和端口号。
- **IP 地址**：IPv4 or IPv6地址，一台计算机可以拥有一个独立的 IP 地址，一个局域网也可以拥有一个独立的 IP 地址（对外就好像只有一台计算机）。
- **MAC 地址**：每个网卡的 MAC 地址在全世界都是独一无二的
- **端口号**：一台计算机可以同时提供多种网络服务，例如 Web 服务（网站）、FTP 服务（文件传输服务）、SMTP 服务（邮箱服务）等，仅有 IP 地址和 MAC 地址，计算机虽然可以正确接收到数据包，但是却不知道要将数据包交给哪个网络程序来处理。为了区分不同的网络程序，计算机会为每个网络程序分配一个独一无二的端口号（Port Number）。端口（Port）是一个虚拟的、逻辑上的概念。可以将端口理解为一道门，数据通过这道门流入流出，每道门有不同的编号，就是端口号。
  - Web 服务的端口号是 80
  - FTP 服务的端口号是 21
  - SMTP 服务的端口号是 25

Linux 下的 socket() 函数

- 在 Linux 下使用 <sys/socket.h> 头文件中 socket() 函数来创建套接字，原型为：
- int socket(int af, int type, int protocol);
  - af 为地址族（Address Family），也就是 IP 地址类型，常用的有 AF_INET 和 AF_INET6。AF 是“Address Family”的简写，INET是“Inetnet”的简写。
    - AF_INET 表示 IPv4 地址
    - AF_INET6表示IPv6地址
  - type 为数据传输方式/套接字类型，常用的有 SOCK_STREAM（流格式套接字/面向连接的套接字） 和 SOCK_DGRAM（数据报套接字/无连接的套接字）
  - protocol 表示传输协议，常用的有 IPPROTO_TCP 和 IPPTOTO_UDP，分别表示 TCP 传输协议和 UDP 传输协议。

Linux下的socket演示程序

- include部分

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

Linux下的socket演示程序服务器端 server.cpp

```
int main(){
//创建套接字
int serv_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);  // 创建一个套接字，参数AF_INET—IPv4地址，SOCK_STREAM—面向连接的套接字，IPPROTO_TCP 表示使用 TCP 协议。
 //将套接字和IP、端口绑定
struct sockaddr_in serv_addr;
memset(&serv_addr, 0, sizeof(serv_addr));  //每个字节都用0填充
serv_addr.sin_family = AF_INET; //使用IPv4地址
serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1"); //具体的IP地址
serv_addr.sin_port = htons(1234); //端口
bind(serv_sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)); //通过 bind() 函数将套接字 serv_sock 与特定的 IP 地址和端口绑定，IP 地址和端口都保存在 sockaddr_in 结构体中。
 //进入监听状态，等待用户发起请求
listen(serv_sock, 20); //处于被动监听状态，是指套接字一直处于“睡眠”中，直到客户端发起请求才会被“唤醒”。Listen()函数使用主动连接套接口变为被连接套接口，使得一个进程可以接受其它进程的请求，从而成为一个服务器进程。
 //接收客户端请求
struct sockaddr_in clnt_addr;
socklen_t clnt_addr_size = sizeof(clnt_addr);
int clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_size); // accept() 函数用来接收客户端的请求。程序一旦执行到 accept() 就会被阻塞（暂停运行），直到客户端发起请求。

//向客户端发送数据
char str[] = “Hello";
write(clnt_sock, str, sizeof(str)); //write() 函数用来向套接字文件中写入数据，也就是向客户端发送数据。
//关闭套接字
close(clnt_sock);
close(serv_sock);
 return 0;
}
```

Linux下的socket演示程序服务器端 client.cpp

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

int main(){
int sock = socket(AF_INET, SOCK_STREAM, 0); //创建套接字

//向服务器（特定的IP和端口）发起请求
struct sockaddr_in serv_addr;
memset(&serv_addr, 0, sizeof(serv_addr));  //每个字节都用0填充
serv_addr.sin_family = AF_INET; //使用IPv4地址
serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1 ");  //具体Server的IP地址
serv_addr.sin_port = htons(1234); //端口
connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr));  //读取服务器传回的数据
char buffer[40];
read(sock, buffer, sizeof(buffer)-1);  //通过 read() 从套接字文件中读取数据。
printf("Message form server: %s\n", buffer);

//关闭套接字
close(sock);
return 0;
}
```

Window 下的 socket() 函数

- SOCKET socket(int af, int type, int protocol);
- Windows 不把套接字作为普通文件对待，而是返回 SOCKET 类型的句柄。

Window下的socket演示程序

- include部分
  - Windows 下的 socket 程序依赖 Winsock.dll 或 ws2_32.dll，必须提前加载。

```
#include <stdio.h>
#include <winsock2.h>
#pragma comment (lib, "ws2_32.lib") //加载 ws2_32.dll
```

Window下的socket演示程序服务器端 server.cpp

```
 int main(){
//初始化 DLL
WSADATA wsaData;
WSAStartup( MAKEWORD(2, 2), &wsaData);

SOCKET servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP); //创建套接字

//绑定套接字
sockaddr_in sockAddr;
memset(&sockAddr, 0, sizeof(sockAddr)); //每个字节都用0填充
sockAddr.sin_family = PF_INET; //使用IPv4地址
sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1"); //具体的IP地址
sockAddr.sin_port = htons(1234); //端口
bind(servSock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));

listen(servSock, 20); //进入监听状态

//接收客户端请求
SOCKADDR clntAddr;
int nSize = sizeof(SOCKADDR);
SOCKET clntSock = accept(servSock, (SOCKADDR*)&clntAddr, &nSize);

//向客户端发送数据
char *str = "Hello World!";
send(clntSock, str, strlen(str)+sizeof(char), NULL); // Linux 下使用 read() / write() 函数读写，而 Windows 下使用 recv() / send() 函数发送和接收。

//关闭套接字
closesocket(clntSock); //关闭 socket 时，Linux 使用 close() 函数，而 Windows 使用 closesocket() 函数。
closesocket(servSock);

//终止 DLL 的使用
WSACleanup();

return 0;
}
```

Window下的socket演示程序服务器端 client.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <WinSock2.h>
#pragma comment(lib, "ws2_32.lib") //加载 ws2_32.dll

int main(){
//初始化DLL
WSADATA wsaData;
WSAStartup(MAKEWORD(2, 2), &wsaData);

SOCKET sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP); //创建套接字

//向服务器发起请求
sockaddr_in sockAddr;
memset(&sockAddr, 0, sizeof(sockAddr)); //每个字节都用0填充
sockAddr.sin_family = PF_INET;
sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
sockAddr.sin_port = htons(1234);
connect(sock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));

//接收服务器传回的数据
char szBuffer[MAXBYTE] = {0};
recv(sock, szBuffer, MAXBYTE, NULL);

//输出接收到的数据
printf("Message form server: %s\n", szBuffer);

//关闭套接字
closesocket(sock);

//终止使用 DLL
WSACleanup();

system("pause");
return 0;
}
```

Windows 下的 socket 程序和 Linux 思路相同，但细节有所差别：
1) Windows 下的 socket 程序依赖 Winsock.dll 或 ws2_32.dll，必须提前加载。

2) Linux 使用“文件描述符”的概念，而 Windows 使用“文件句柄”的概念；Linux 不区分 socket 文件和普通文件，而 Windows 区分；Linux 下 socket() 函数的返回值为 int 类型，而 Windows 下为 SOCKET 类型，也就是句柄。

3)大家需要记住127.0.0.1，它是一个特殊IP地址，表示本机地址。

int socket(int af, int type, int protocol);

4)有了地址类型和数据传输方式，还不足以决定采用哪种协议吗？为什么还需要第三个参数呢？
一般情况下有了 af 和 type 两个参数就可以创建套接字了，操作系统会自动推演出协议类型，除非遇到这样的情况：有两种不同的协议支持同一种地址类型和数据传输类型。如果我们不指明使用哪种协议，操作系统是没办法自动推演的。

5) Linux 下使用 read() / write() 函数读写，而 Windows 下使用 recv() / send() 函数发送和接收。

6) 关闭 socket 时，Linux 使用 close() 函数，而 Windows 使用 closesocket() 函数。

bind() 函数

- socket() 函数用来创建套接字，确定套接字的各种属性，然后服务器端要用 bind() 函数将套接字与特定的 IP 地址和端口绑定起来，只有这样，流经该 IP 地址和端口的数据才能交给套接字处理。类似地，客户端也要用 connect() 函数建立连接。
- bind() 函数的原型为:
  - int bind(int sock, struct sockaddr *addr, socklen_t addrlen); //Linux
  - int bind(SOCKET sock, const struct sockaddr *addr, int addrlen); //Windows

- sockaddr_in 结构体

```
struct sockaddr_in{
sa_family_t sin_family;   //地址族（Address Family），也就是地址类型
uint16_t sin_port;        //16位的端口号
struct in_addr sin_addr; //32位IP地址
char sin_zero[8];         //不使用，一般用0填充
};
```

- 1) sin_family 和 socket() 的第一个参数的含义相同，取值也要保持一致。

  2) sin_prot 为端口号。uint16_t 的长度为两个字节，理论上端口号的取值范围为 0~65536，但 0~1023 的端口一般由系统分配给特定的服务程序，例如 Web 服务的端口号为 80，FTP 服务的端口号为 21，所以我们的程序要尽量在 1024~65536 之间分配端口号。

  端口号需要用 htons() 函数转换。

  3) sin_addr 是 struct in_addr 结构体类型的变量，下面会详细讲解。

  4) sin_zero[8] 是多余的8个字节，没有用，一般使用 memset() 函数填充为 0。上面的代码中，先用 memset() 将结构体的全部字节填充为 0，再给前 3个成员赋值，剩下的 sin_zero 自然就是 0 了。

- in_addr 结构体

```
struct in_addr{
in_addr_t s_addr;  //32位的IP地址
};
```

- in_addr_t 在头文件 <netinet/in.h> 中定义，等价于 unsigned long，长度为4个字节。也就是说，s_addr 是一个整数，而IP地址是一个字符串，所以需要 inet_addr() 函数进行转换。

- 为什么使用 sockaddr_in 而不使用 sockaddr？
  - bind() 第二个参数的类型为 sockaddr，而代码中却使用 sockaddr_in，然后再强制转换为 sockaddr，这是为什么呢？
- sockaddr 结构体

```
struct sockaddr{
sa_family_t sin_family;  //地址族（Address Family），也就是地址类型
char sa_data[14];        //IP地址和端口号
};
```

- sockaddr 和 sockaddr_in 的长度相同，都是16字节，只是将IP地址和端口号合并到一起，用一个成员 sa_data 表示。要想给 sa_data 赋值，必须同时指明IP地址和端口号，例如“127.0.0.1:80”，遗憾的是，没有相关函数将这个字符串转换成需要的形式，也就很难给 sockaddr 类型的变量赋值，所以使用 sockaddr_in 来代替。这两个结构体的长度相同，强制转换类型时不会丢失字节，也没有多余的字节。
- sockaddr 是一种通用的结构体，可以用来保存多种类型的IP地址和端口号，而 sockaddr_in 是专门用来保存 IPv4 地址的结构体。

- sockaddr_in6，用来保存 IPv6 地址

```
struct sockaddr_in6 {
sa_family_t sin6_family;      //(2)地址类型，取值为AF_INET6
in_port_t sin6_port;          //(2)16位端口号
uint32_t sin6_flowinfo;       //(4)IPv6流信息
struct in6_addr sin6_addr;  //(4)具体的IPv6地址
uint32_t sin6_scope_id;     //(4)接口范围ID
};
```

- Connection() 函数原型：
  - int connect(int sock, struct sockaddr *serv_addr, socklen_t addrlen); //Linux
  - int connect(SOCKET sock, const struct sockaddr *serv_addr, int addrlen); //Windows

- 对于服务器端程序，使用 bind() 绑定套接字后，还需要使用 listen() 函数让套接字进入被动监听状态，再调用 accept() 函数，就可以随时响应客户端的请求了。
  - 通过 listen() 函数可以让套接字进入被动监听状态，它的原型为：
    - int listen(int sock, int backlog); //Linux
    - int listen(SOCKET sock, int backlog); //Windows
  - sock 为需要进入监听状态的套接字，backlog 为请求队列的最大长度。
  - 所谓被动监听，是指当没有客户端请求时，套接字处于“睡眠”状态，只有当接收到客户端请求时，套接字才会被“唤醒”来响应请求。
  - 注意：listen() 只是让套接字处于监听状态，并没有接收请求。接收请求需要使用 accept() 函数。
  - 当套接字正在处理客户端请求时，如果有新的请求进来，套接字是没法处理的，只能把它放进缓冲区，待当前请求处理完毕后，再从缓冲区中读取出来处理。如果不断有新的请求进来，它们就按照先后顺序在缓冲区中排队，直到缓冲区满。这个缓冲区，就称为请求队列（Request Queue）。
  - 缓冲区的长度（能存放多少个客户端请求）可以通过 listen() 函数的 backlog 参数指定，但究竟为多少并没有什么标准，可以根据你的需求来定，并发量小的话可以是10或者20。
  - 如果将 backlog 的值设置为 SOMAXCONN，就由系统来决定请求队列长度，这个值一般比较大，可能是几百，或者更多。
  - 当请求队列满时，就不再接收新的请求，对于 Linux，客户端会收到 ECONNREFUSED 错误，对于 Windows，客户端会收到 WSAECONNREFUSED 错误。

accept() 函数

- 当套接字处于监听状态时，可以通过 accept() 函数来接收客户端请求。它的原型为：
  - int accept(int sock, struct sockaddr *addr, socklen_t *addrlen); //Linux
  - SOCKET accept(SOCKET sock, struct sockaddr *addr, int *addrlen); //Windows

- accept() 返回一个新的套接字来和客户端通信，addr 保存了客户端的IP地址和端口号，而 sock 是服务器端的套接字，大家注意区分。后面和客户端通信时，要使用这个新生成的套接字，而不是原来服务器端的套接字。
- listen() 只是让套接字进入监听状态，并没有真正接收客户端请求，listen() 后面的代码会继续执行，直到遇到 accept()。accept() 会阻塞程序执行（后面代码不能被执行），直到有新的请求到来。

Socket 缓冲区 (I)

- 每个 socket 被创建后，都会分配两个缓冲区，输入缓冲区和输出缓冲区。
- write()/send() 并不立即向网络中传输数据，而是先将数据写入输出缓冲区中，再由TCP协议将数据从输出缓冲区发送到目标机器。一旦将数据写入到缓冲区，函数就可以成功返回，不管它们有没有到达目标机器，也不管它们何时被发送到网络，这些都是TCP协议负责的事情。
  - write() 原型： ssize_t write(int fd, const void *buf, size_t nbytes); // Linux
  - send() 原型： int send(SOCKET sock, const char *buf, int len, int flags); // Window
  - TCP协议独立于 write()/send() 函数，数据有可能刚被写入缓冲区就发送到网络，也可能在缓冲区中不断积压，多次写入的数据被一次性发送到网络，这取决于当时的网络情况、当前线程是否空闲等诸多因素，不由程序员控制。

Socket 缓冲区 (II)

- read()/recv() 函数也是如此，也从输入缓冲区中读取数据，而不是直接从网络中读取。
  - read() 原型：ssize_t read(int fd, void *buf, size_t nbytes); // Linux
  - recv() 原型：int recv(SOCKET sock, char *buf, int len, int flags); // Window

- 这些I/O缓冲区特性如下：
  - I/O缓冲区在每个TCP套接字中单独存在；
  - I/O缓冲区在创建套接字时自动生成；
  - 即使关闭套接字也会继续传送输出缓冲区中遗留的数据；
  - 关闭套接字将丢失输入缓冲区中的数据。

- 输入输出缓冲区的默认大小一般都是 8K，可以通过 getsockopt() 函数获取：

阻塞模式 (I)

- 对于TCP套接字（默认情况下），当使用 write()/send() 发送数据时：
  - 1) 首先会检查缓冲区，如果缓冲区的可用空间长度小于要发送的数据，那么 write()/send() 会被阻塞（暂停执行），直到缓冲区中的数据被发送到目标机器，腾出足够的空间，才唤醒 write()/send() 函数继续写入数据。
  - 2) 如果TCP协议正在向网络发送数据，那么输出缓冲区会被锁定，不允许写入，write()/send() 也会被阻塞，直到数据发送完毕缓冲区解锁，write()/send() 才会被唤醒。
  - 3) 如果要写入的数据大于缓冲区的最大长度，那么将分批写入。
  - 4) 直到所有数据被写入缓冲区 write()/send() 才能返回。

阻塞模式 (II)

- 当使用 read()/recv() 读取数据时：
  - 1) 首先会检查缓冲区，如果缓冲区中有数据，那么就读取，否则函数会被阻塞，直到网络上有数据到来。
  - 2) 如果要读取的数据长度小于缓冲区中的数据长度，那么就不能一次性将缓冲区中的所有数据读出，剩余数据将不断积压，直到有 read()/recv() 函数再次读取。
  - 3) 直到读取到数据后 read()/recv() 函数才会返回，否则就一直被阻塞。
- 这就是TCP套接字的阻塞模式。所谓阻塞，就是上一步动作没有完成，下一步动作将暂停，直到上一步动作完成后才能继续，以保持同步性。

TCP 粘包问题

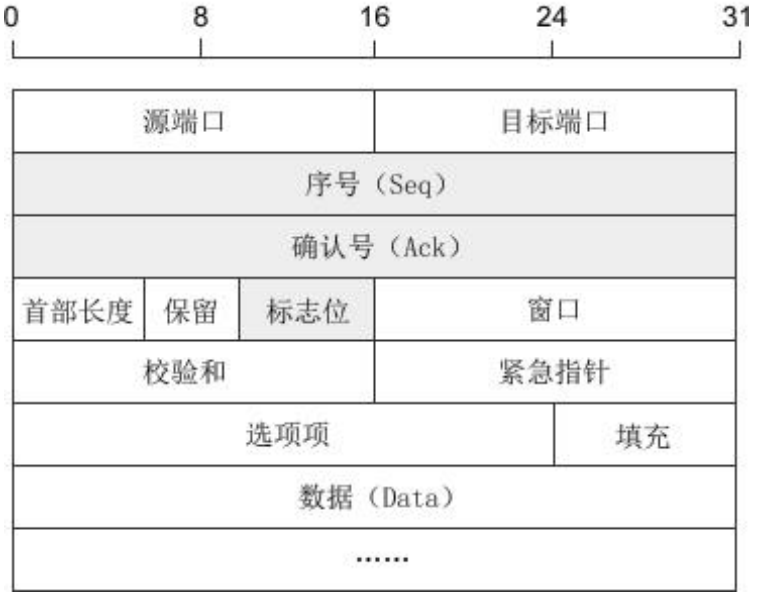
- 数据的“粘包”问题，客户端发送的多个数据包被当做一个数据包接收，也称数据的无边界性，read()/recv() 函数不知道数据包的开始或结束标志（实际上也没有任何开始或结束标志），只把它们当做连续的数据流来处理。
- 例如：假设我们希望客户端每次发送一位学生的学号，让服务器端返回该学生的姓名、住址、成绩等信息，这时候可能就会出现问题，服务器端不能区分学生的学号。例如第一次发送 1，第二次发送 3，服务器可能当成 13 来处理，返回的信息显然是错误的。
- 例如，write()/send() 重复执行三次，每次都发送字符串"abc"，那么目标机器上的 read()/recv() 可能分三次接收，每次都接收"abc"；也可能分两次接收，第一次接收"abcab"，第二次接收"cabc"；也可能一次就接收到字符串"abcabcabc"。
  - read()/recv() 和 write()/send() 的执行次数可能不同。

TCP数据报结构 (I)

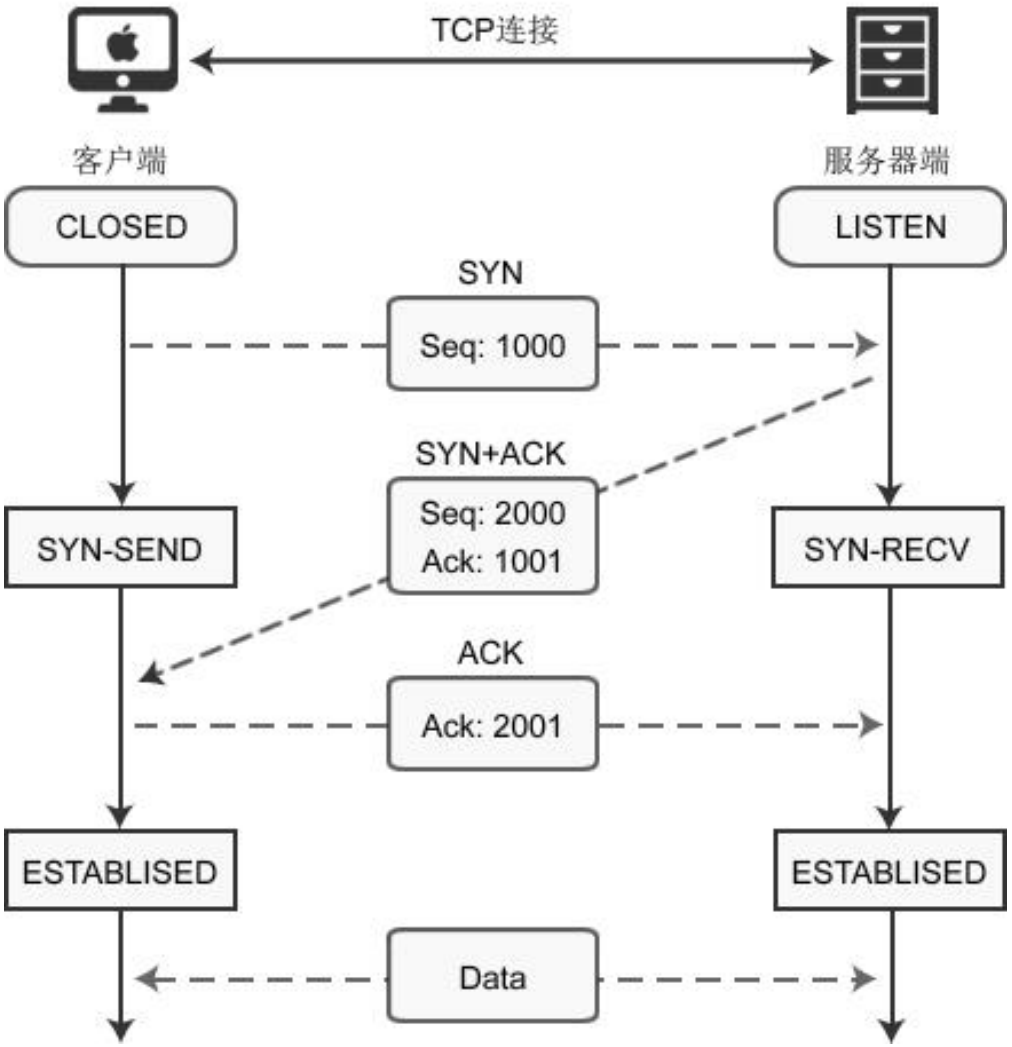
- TCP（Transmission Control Protocol，传输控制协议）是一种面向连接的、可靠的、基于字节流的通信协议，数据在传输前要建立连接，传输完毕后还要断开连接。
- 客户端在收发数据前要使用 connect() 函数和服务器建立连接。建立连接的目的是保证IP地址、端口、物理链路等正确无误，为数据的传输开辟通道。
- TCP建立连接时要传输三个数据包，俗称三次握手（**Three-way Handshaking**）。可以形象的比喻为下面的对话：
  - [Shake 1] 套接字A：“你好，套接字B，我这里有数据要传送给你，建立连接吧。”
  - [Shake 2] 套接字B：“好的，我这边已准备就绪。”
  - [Shake 3] 套接字A：“谢谢你受理我的请求。”

TCP数据报结构 (II)

- 1) 序号：Seq（Sequence Number）序号占32位，用来标识从计算机A发送到计算机B的数据包的序号，计算机发送数据时对此进行标记。
- 2) 确认号：Ack（Acknowledge Number）确认号占32位，客户端和服务器端都可以发送，Ack = Seq + 1。
- 3) 标志位：每个标志位占用1Bit，共有6个，分别为 URG、ACK、PSH、RST、SYN、FIN，具体含义如下：
  - URG — 紧急指针（urgent pointer）有效。
  - ACK —确认序号有效。
  - PSH —接收方应该尽快将这个报文交给应用层。
  - RST —重置连接。
  - SYN —建立一个新连接。
  - FIN —断开一个连接。

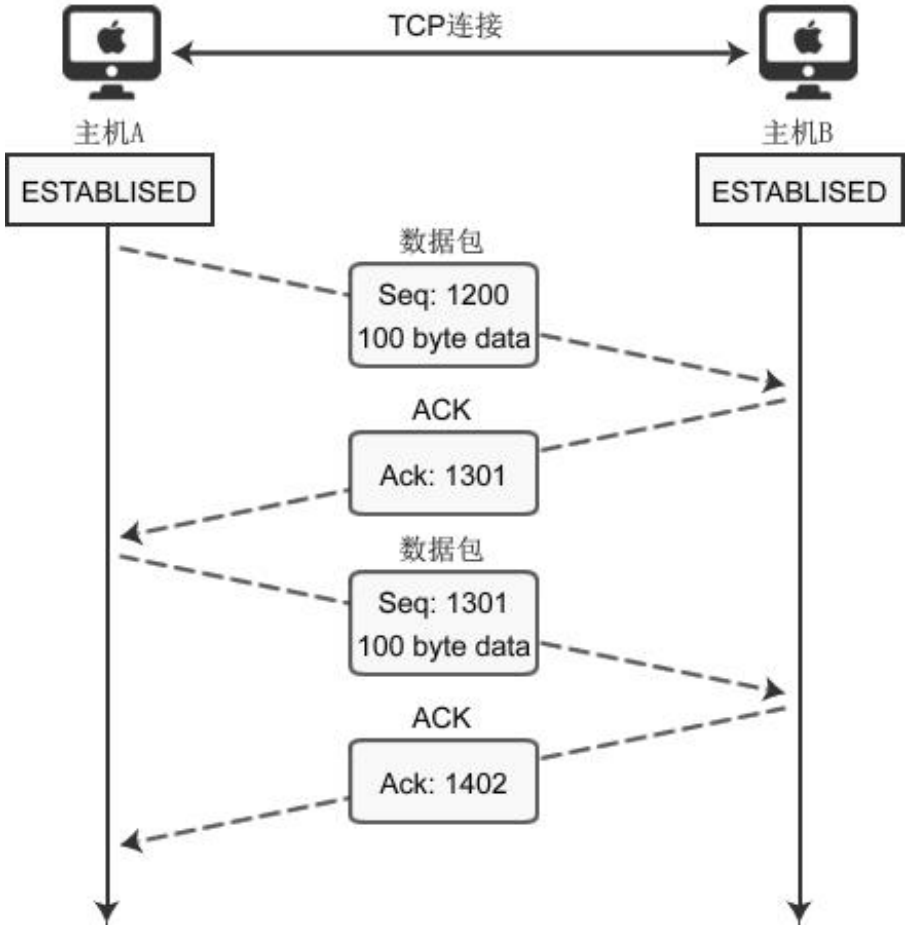


TCP三次握手



三次握手的关键是要确认对方收到了自己的数据包，这个目标就是通过“确认号（Ack）”字段实现的。计算机会记录下自己发送的数据包序号 Seq，待收到对方的数据包后，检测“确认号（Ack）”字段，看Ack = Seq + 1是否成立，如果成立说明对方正确收到了自己的数据包。

TCP数据传输过程 (I)

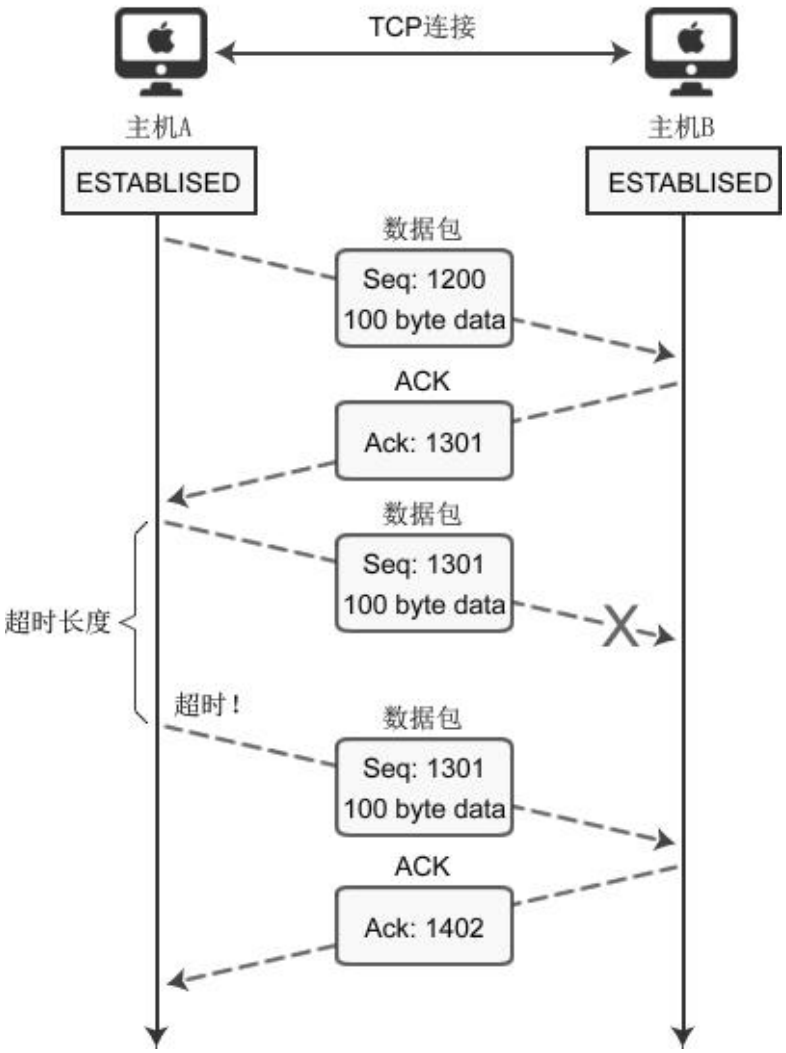


此时 Ack 号为 1301 而不是 1201，原因在于 Ack 号的增量为传输的数据字节数。假设每次 Ack 号不加传输的字节数，这样虽然可以确认数据包的传输，但无法明确100字节全部正确传递还是丢失了一部分，比如只传递了80字节。因此按如下的公式确认 Ack 号：

Ack号 = Seq号 + 传递的字节数 + 1

与三次握手协议相同，最后加 1 是为了告诉对方要传递的 Seq 号。

TCP数据传输过程 (II)



为了完成数据包的重传，TCP套接字每次发送数据包时都会启动定时器，如果在一定时间内没有收到目标机器传回的 ACK 包，那么定时器超时，数据包会重传。 左图演示的是数据包丢失的情况，也会有 ACK 包丢失的情况，一样会重传。

重传超时时间（RTO, Retransmission Time Out）
这个值太大了会导致不必要的等待，太小会导致不必要的重传，理论上最好是网络 RTT 时间，但又受制于网络距离与瞬态时延变化，所以实际上使用自适应的动态算法（例如 Jacobson 算法和 Karn 算法等）来确定超时时间。

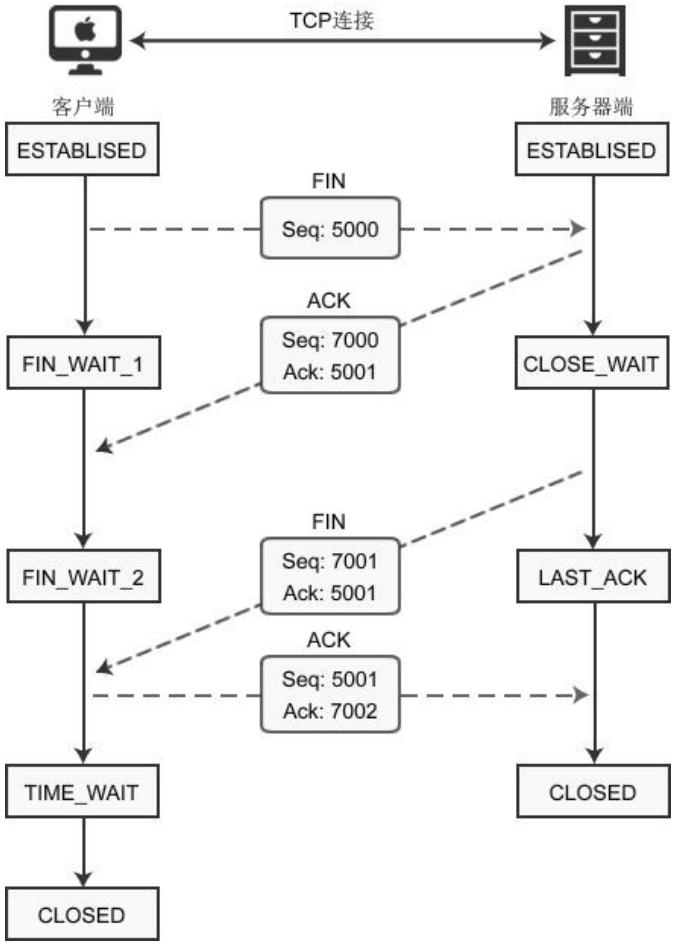
重传次数
TCP数据包重传次数根据系统设置的不同而有所区别。有些系统，一个数据包只会被重传3次，如果重传3次后还未收到该数据包的 ACK 确认，就不再尝试重传。但有些要求很高的业务系统，会不断地重传丢失的数据包，以尽最大可能保证业务数据的正常交互。

TCP四次握手断开连接

- 建立连接非常重要，它是数据正确传输的前提；断开连接同样重要，它让计算机释放不再使用的资源。如果连接不能正常断开，不仅会造成数据传输错误，还会导致套接字不能关闭，持续占用资源，如果并发量高，服务器压力堪忧。

- 建立连接需要三次握手，断开连接需要四次握手，可以形象的比喻为下面的对话：
  - [Shake 1] 套接字A：“任务处理完毕，我希望断开连接。”
  - [Shake 2] 套接字B：“哦，是吗？请稍等，我准备一下。”
  - 等待片刻后……
  - [Shake 3] 套接字B：“我准备好了，可以断开连接了。”
  - [Shake 4] 套接字A：“好的，谢谢合作。”

TCP四次握手断开连接



注意：服务器收到请求后并不是立即断开连接，而是先向客户端发送“确认包”，告诉它我知道了，我需要准备一下才能断开连接。

- 客户端最后一次发送 ACK包后进入 TIME_WAIT 状态，而不是直接进入 CLOSED 状态关闭连接，这是为什么呢？
  - TCP 是面向连接的传输方式，必须保证数据能够正确到达目标机器，不能丢失或出错，而网络是不稳定的，随时可能会毁坏数据，所以机器A每次向机器B发送数据包后，都要求机器B“确认”，回传ACK包，告诉机器A我收到了，这样机器A才能知道数据传送成功了。如果机器B没有回传ACK包，机器A会重新发送，直到机器B回传ACK包。
- 客户端最后一次向服务器回传ACK包时，有可能会因为网络问题导致服务器收不到，服务器会再次发送 FIN 包，如果这时客户端完全关闭了连接，那么服务器无论如何也收不到ACK包了，所以客户端需要等待片刻、确认对方收到ACK包后才能进入CLOSED状态。那么，要等待多久呢？
  - 数据包在网络中是有生存时间的，超过这个时间还未到达目标主机就会被丢弃，并通知源主机。这称为报文最大生存时间（MSL，Maximum Segment Lifetime）。TIME_WAIT 要等待 2MSL 才会进入 CLOSED 状态。ACK 包到达服务器需要 MSL 时间，服务器重传 FIN 包也需要 MSL 时间，2MSL 是数据包往返的最大时间，如果 2MSL 后还未收到服务器重传的 FIN 包，就说明服务器已经收到了 ACK 包。

References

- [1] 陆魁军，计算机网络实践基础教程，第二章，清华大学出版社，2005.
- [2] https://blog.csdn.net/hou09tian/article/details/82782247
- [3] https://blog.csdn.net/dyxcome/article/details/81783911 (初始化winsock的几种方法)