Lab7 — 基于Socket接口实现自定义协议通信

Dr. Xiqun Lu
College of Computer Science
Zhejiang University

实验目的

- 学习如何设计网络应用协议
- 掌握Socket编程接口编写基本的网络应用软件

实验内容

- 根据自定义的协议规范，使用Socket编程接口编写基本的网络应用软件。
  - 掌握C语言形式的Socket编程接口用法，能够正确发送和接收网络数据包
  - 开发一个客户端，实现人机交互界面与服务器的通信
  - 开发一个服务端，实现并发处理多个客户端的请求
  - 程序界面不做要求，使用命令行或最简单的窗体即可

编程要求

- 本实验涉及到网络数据包发送部分**不能**使用任何的Socket封装类，只能使用最底层的C语言形式的Socket API。
- 本实验可组成小组，服务端和客户端可由不同人来完成

功能要求 (I)

- 传输层协议采用**TCP**协议；
- **客户端**采用交互菜单形式，用户可以选择以下功能：
  - 连接：请求连接到指定地址和端口的服务端
  - 断开连接：断开与服务端的连接
  - 获取时间: 请求服务端给出当前时间
  - 获取名字：请求服务端给出其机器的名称
  - 活动连接列表：请求服务端给出当前连接的所有客户端信息（编号、IP地址、端口等）
  - 发消息：请求服务端把消息转发给对应编号的客户端，该客户端收到后显示在屏幕上
  - 退出：断开连接并退出客户端程序

功能要求 (II)

- **服务端**接收到客户端请求后，根据客户端传过来的指令完成特定任务：
  - 向客户端传送服务端所在机器的当前时间
  - 向客户端传送服务端所在机器的名称
  - 向客户端传送当前连接的所有客户端信息
  - 将某客户端发送过来的内容转发给指定编号的其他客户端
  - 采用**异步多线程编程模式**，正确处理多个客户端同时连接，同时发送消息的情况

IP ADDRESS, MAC ADDRESS, PORT NUMBER

IP 地址，MAC 地址和端口号 (I)

- 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 (32 bits) or IPv6 (128 bits) 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** (48 bits)：每个网卡的 MAC 地址在全世界都是独一无二的。多数现实的情况是，一个局域网往往才能拥有一个独立的IP地址；换句话说，IP 地址只能定位到一个局域网，无法定位到具体的一台计算机。
  - 数据包中除了会附带对方的 IP 地址，还会附带对方的 MAC 地址，当数据包达到局域网以后，路由器/交换机会根据数据包中的 MAC 地址找到对应的计算机，然后把数据包转交给它，这样就完成了数据的传递。

IP 地址、MAC 地址和端口号 (II)

- **端口号** (16 bits)：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应用程序来处理。为了区分不同的网络应用程序，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。可以将某台计算机看成是一个居民楼，而每个网络应用程序是居住在这个楼的居民，为了能把信件投送到家，必须在投送地址上写明房间号。
  - 注意一般0-1023端口是分配给一些特定的网络应用服务，如
    - http 服务的端口号是 80，https(TCP)服务的端口号是443
    - DNS(UDP)服务的端口号是 53
    - FTP 服务的端口号是 21
    - SMTP 服务的端口号是 25 (POP3 — 101)
  - 我们实验中端口号请选择1024-65535之间的数值（以自己学号的后四位作为服务器端的监听端口号，如果后四位中的第一位为零，则采用1XXXX）。

SOCKET

什么是socket？[1, 4]

- 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。
  - Sockets were first released as part of the Berkeley UNIX 4.2BSD software distribution in1983.

- **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 winsocket？

- 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。

流格式套接字 SOCK_STREAM

- 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（是一个字符数组），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)

数据报格式套接字 SOCK_DGRAM

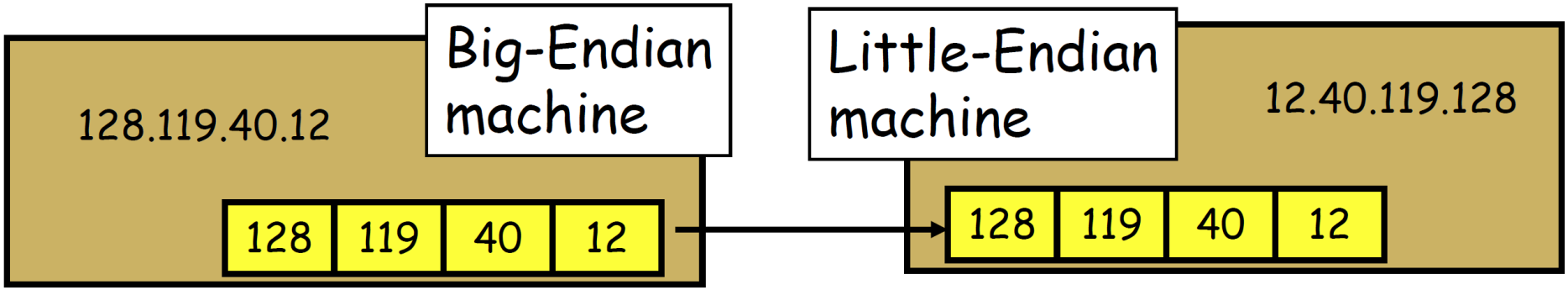
- **数据报格式套接字（SOCK_DGRAM）**计算机只管传输数据，不作数据校验，如果数据在传输中损坏，或者没有到达另一台计算机，是没有办法补救的。也就是说，数据错了就错了，无法重传。
  - 强调快速传输而非传输顺序；
  - 传输的数据可能丢失也可能损毁；
  - 限制每次传输的数据大小；
  - 数据的发送和接收是同步的
  - Example: QQ视频和语音聊天 (UDP)

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

CONSTRUCTING MESSAGES: BYTE ORDERING

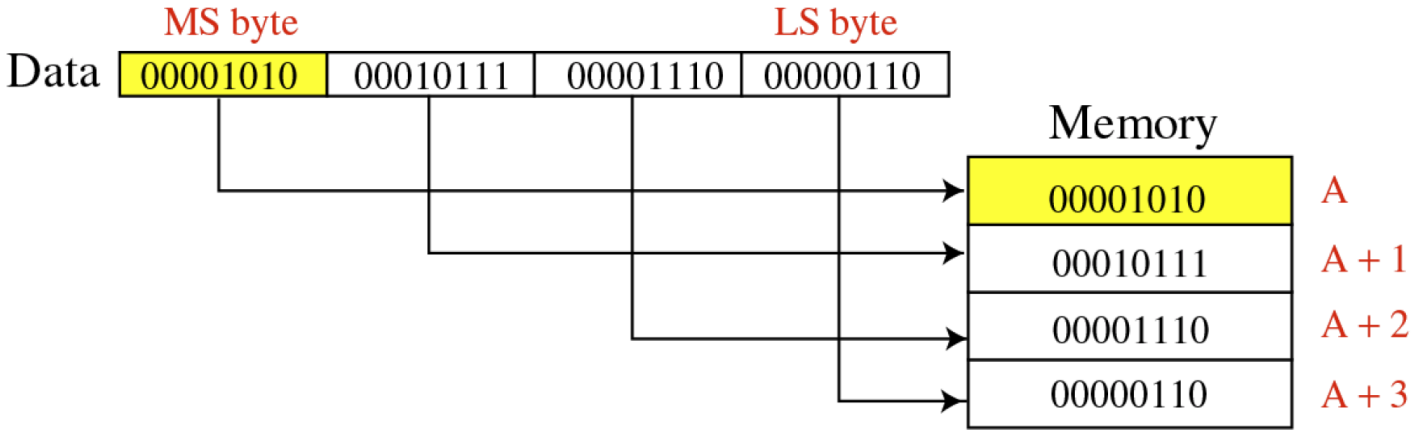
Constructing Message: Byte Ordering [4]

- Problems:
  - Different machines / OS's use different word orderings
    - **Little-endian**: lower bytes first
    - **Big-endian**: higher bytes first
  - These machines may communicate with one another over the network

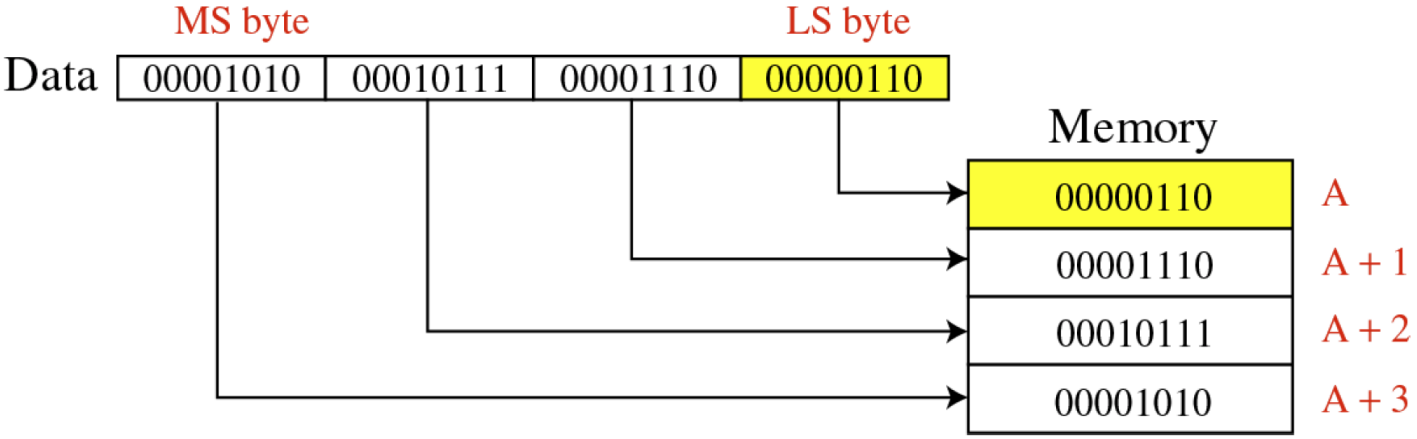


Constructing Message: Byte Ordering

- Big-Endian:



- Little-Endian:

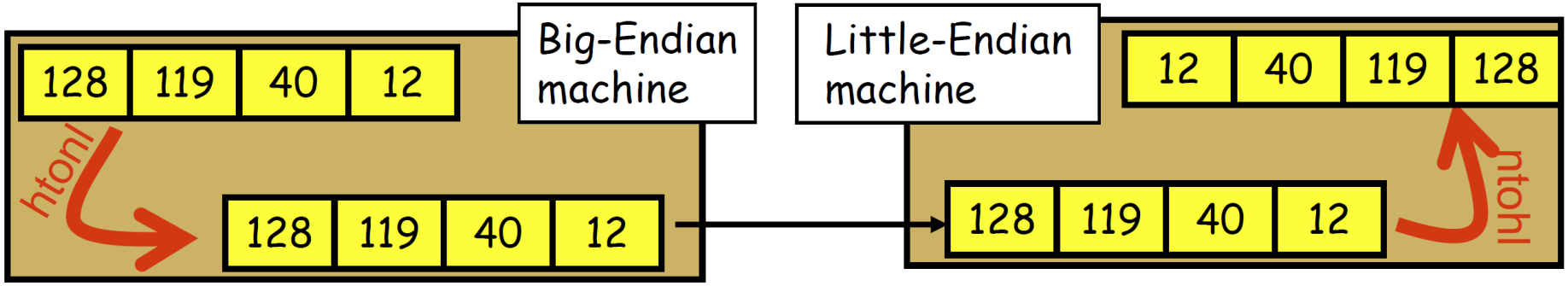


Constructing Message: Byte Ordering [4]

- **Host Byte-Ordering**: the byte ordering used by a host (big or little, little-endian is popular in X-86 systems)
- **Network Byte-Ordering**: the byte ordering used by the network — always big-endian

```
u_long   htonl(u_long x);    u_long   ntohl(u_long x);
u_short  htons(u_short x);   u_short  ntohs(u_short x);
```

  - Port numbers (16 bits)： htons (host to network short), ntohs (network to host short)
  - IP addresses (32 bits)： htonl (host to network long), ntohl (network to host long)
  - On big-endian machines, these routines do nothing
  - On little-endian machines, they reverse the byte order



Constructing Message: Alignment & Padding [5]

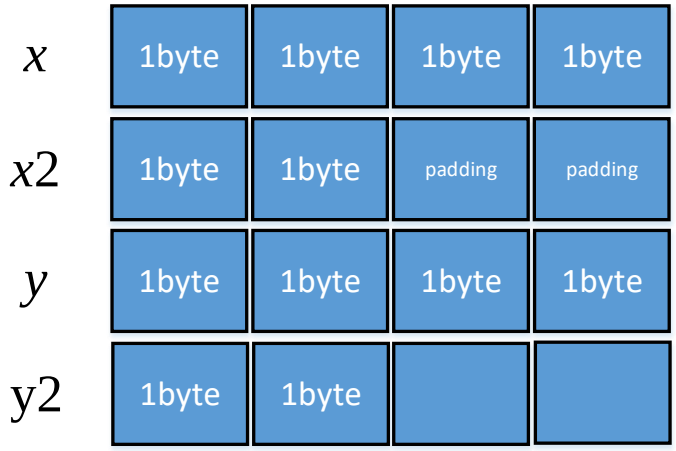
- **Data structure alignment** is the way data is arranged and accessed in computer memory.
  - *Data alignment* and *data structure padding* are two different issues but are related to each other and together known as Data Structure alignment.
- **Data alignment**: Data alignment means putting the data in memory at an address equal to some multiple of the word size. This increases the performance of the system due to the way the CPU handles memory.
- **Data Structure Padding**: Now, to align the data, it may be necessary to insert some extra bytes between the end of the last data structure and the start of the next data structure as the data is placed in memory as multiples of fixed word size. This insertion of extra bytes of memory to align the data is called data structure padding.

Constructing Message: Alignment & Padding

- Consider the following 12 bytes structure

```
struct {
  int x;
  short int x2;
  int y;
  short int y2;
} msg1Struct;
```

- After compilation it will be a 14 byte structure!
- Why? → Alignment & Padding



- Remember the following rules:
  - Data structures are maximally aligned, according to the size of the largest native integer
  - Other multibyte fields are aligned to their size, e.g., a four-byte integer's address will be divisible by four.

Constructing Message: Alignment & Padding

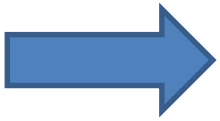
- Consider the following 12 bytes structure

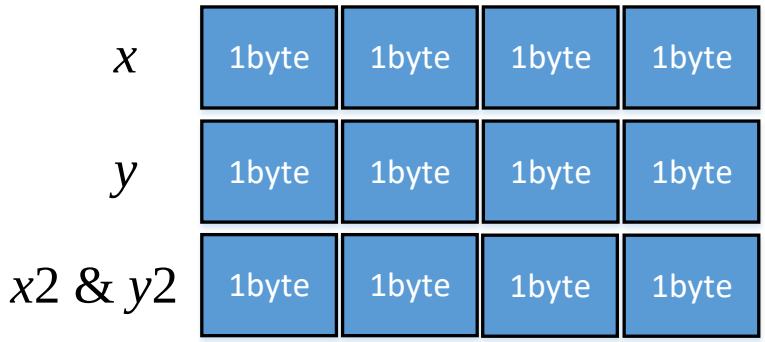
```
struct {
  int x;
  short int x2;
  int y;
  short int y2;
} msg1Struct;
```

- This can be avoided
  - Include padding to data structure
  - Reorder fields

```
struct {
  int x;
  short int x2;
  char padding[2];
  int y;
  short int y2;
} msg2Struct;
```

```
struct {
  int x;
  int y;
  short int x2;
  short int y2;
} msg3Struct;
```





Constructing Message: Framing & Parsing

- **Framing** is the problem of formatting the information so that the receiver can parse messages
- **Parsing** means to locate the beginning and the end of message.
- This is easy if the fields have fixed sizes!
  - e.g. msgStruct
- For text-string representation is harder
  - Solution: use of appropriate delimiters
  - Caution is needed since a call of “recv” may return the messages sent by multiple calls of “send”.

SOCKET PROGRAMMING WITH TCP

Socket Programming with TCP [3]

- Many network applications consists of a pair programs — a client program and a server program — residing in two different end systems
  - Web (visiting http://www.zju.edu.cn)
  - ftp (ftp://10.214.X.Y)
- When these two processes are executed, a client and a server process are created, and these processes communicate with each other by reading from and writing to **sockets**.
- During the development phase, one of the first decisions the developer must make is whether the application is run over **TCP** or over **UDP**.
  - TCP & UDP are the two most important protocols in the transport layer.
  - TCP is **connection oriented** and provides **a reliable byte-stream channel** through which data flows between two end systems.
  - UDP is **connectionless** and sends independent packets of data from one end system to the other, without any guarantees about delivery.

Socket Programming with TCP [3]

- The socket is the door between the application process and TCP

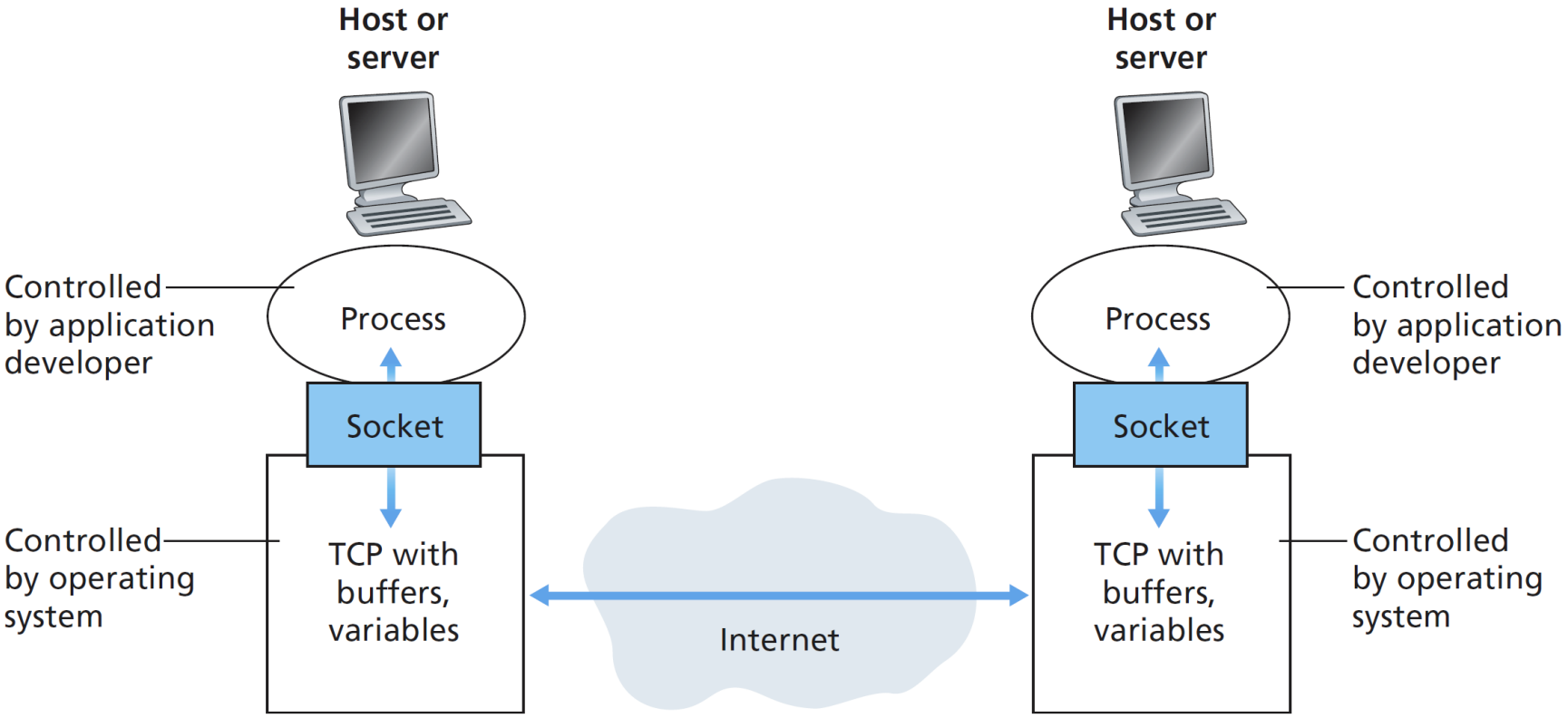


Figure 2.28 ♦ Processes communicating through TCP sockets

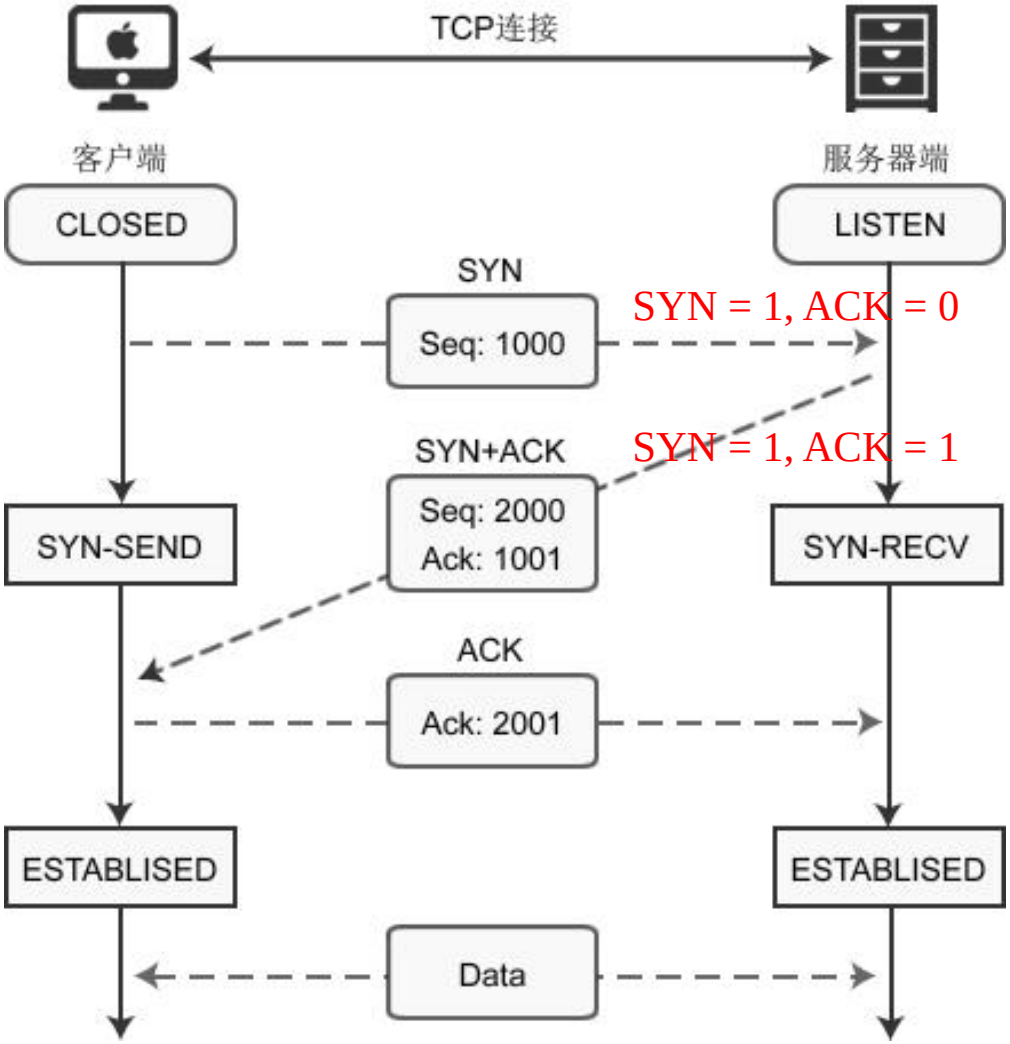
TCP数据报结构 (I) [4]

- 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)

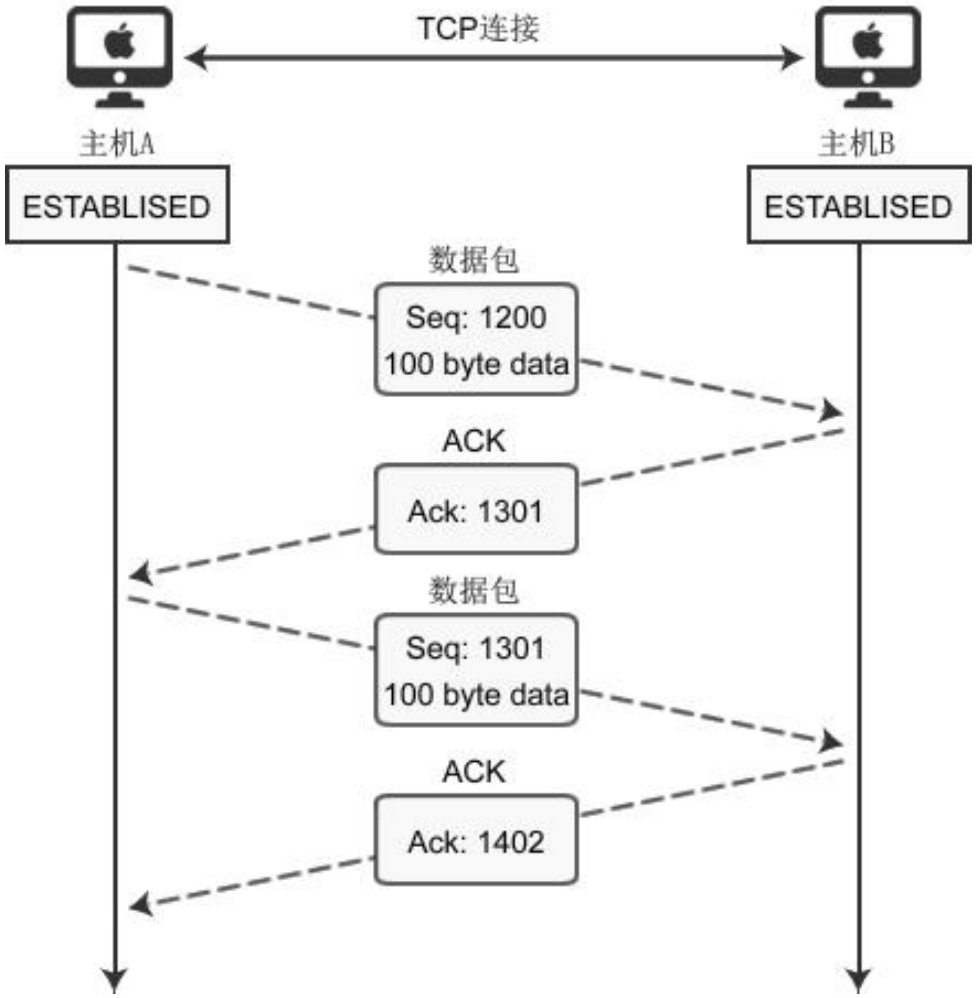
- 1) 序号：Seq（Sequence Number）序号占32位，用来标识从计算机A发送到计算机B的数据包的序号，计算机发送数据时对此进行标记。
- 2) 确认号：Ack（Acknowledge Number）确认号占32位，客户端和服务器端都可以发送，Ack = Seq + 1。
- 3) 标志位：每个标志位占用1Bit，共有6个，分别为URG、ACK、PSH、RST、SYN、FIN，具体含义如下：
  - URG — 紧急指针（urgent pointer）有效。
  - ACK —确认序号有效。
  - PSH —接收方应该尽快将这个报文交给应用层。
  - RST —重置连接。
 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

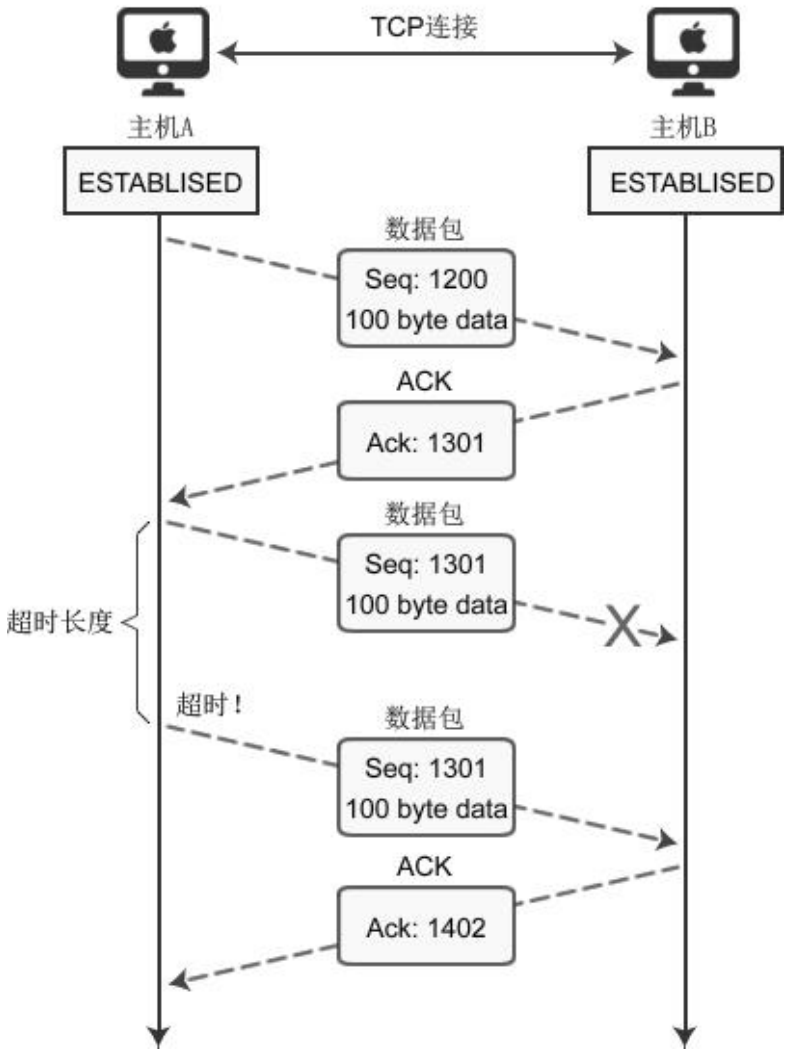


此时 Ack 号为 1301 而不是 1201，原因在于 Ack 号的增量为对方传输过来的数据字节数。假设每次 Ack 号不加传输的字节数，这样虽然可以确认数据包的传输，但无法明确100字节全部正确传递还是丢失了一部分，比如只传递了80字节。因此按如下的公式确认 Ack 号：

Ack号 = Seq号 + 传递的字节数 + 1

与三次握手协议相同，最后加 1 是为了告诉对方接下来要传递的 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。

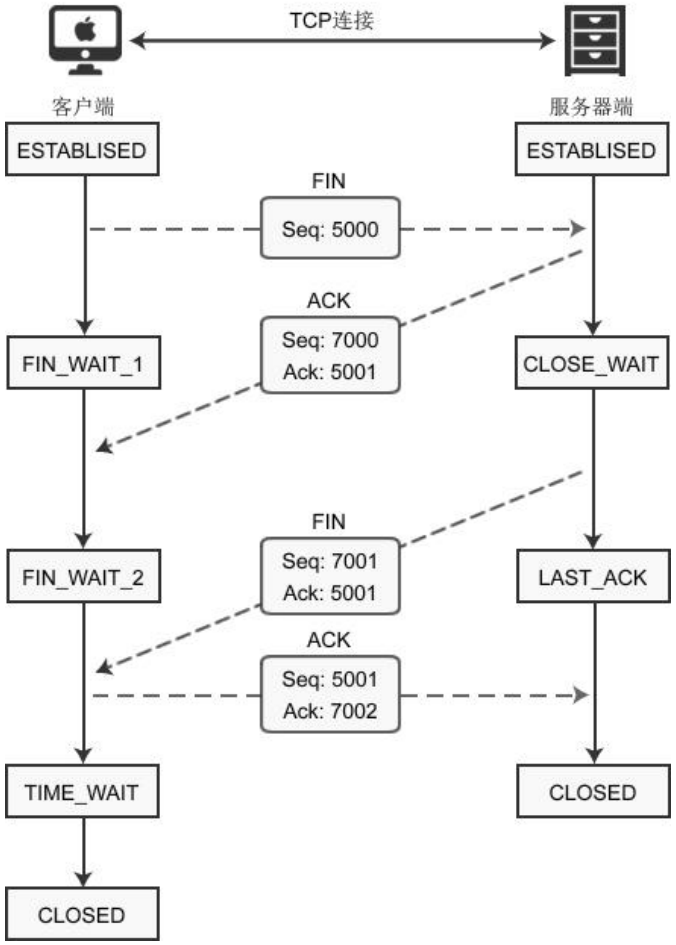
重传次数
TCP数据包重传次数根据系统设置的不同而有所区别。有些系统，一个数据包只会被重传3次，如果重传3次后还未收到该数据包的 ACK 确认，就不再尝试重传。但有些要求很高的业务系统，会不断地重传丢失的数据包，以尽最大可能保证业务数据的正常交互。

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，而网络是动态不稳定的，数据随时可能被会毁坏或丢弃，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。

SOCKET PROGRAMMING WITH TCP IN LINUX

Socket Primitives for TCP [4]

Primitive	Meaning
SOCKET	Create a new communication endpoint
BIND	Associate a local address with a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Passively establish an incoming connection
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

Figure 6-5. The socket primitives for TCP.

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。
    - AF_INET 表示 IPv4 地址 (INET是“Internet”的简写)
    - AF_INET6表示IPv6地址
    - AF_UNIX (local channel, similar to pipes)
    - AF_ISO (ISO protocols)
  - 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 ~ TCP） 和 SOCK_DGRAM（数据报套接字/无连接的套接字 ~ UDP）
  - protocol 表示传输协议，常用的有 IPPROTO_TCP 和 IPPROTO_UDP，分别表示 TCP 传输协议和 UDP 传输协议。Usually set to 0 (i.e., use default protocol). 有可能多种协议使用同一种数据传输方式，所以在socket编程中，需要同时指明数据传输方式和协议。(This is useful in cases where some families may have more than one protocol to support a given type of service.)
  - Note: socket call does not specify where data will be coming from, nor where it will be going to — it just creates the interface!

Address Structure [6]

- Socket functions like connect(), accept(), and bind() require the use of specifically defined address structures to hold IP address, port number, and protocol type.
- The difficulty is that you can use sockets to program network applications using different protocols. For example, we can use IPv4, IPv6, Unix local, etc. Here is the problem: Each different protocol uses a different address structure to hold its addressing information, yet they all use the same functions connect(), accept(), and bind() etc.
- How do we pass these different structures to a given socket function that requires an address structure?
- This is how it is done: there is a **generic** address structure: struct sockaddr

struct sockaddr

```
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
};
```

- This is the address structure which must be passed to all of the socket functions requiring an address structure.
- This means that you must type cast your specific protocol dependent address structure to the generic address structure when passing it to these socket functions, like connect(), accept(), and bind() etc.

IPv4 address structure: struct sockaddr_in

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

- 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_port 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 0~65535，但 0~1023 的端口一般由系统分配给特定的服务程序。
- 3) 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。
- 4) 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。

struct sockaddr & struct sockaddr_in

- 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。
  - connect(), bind()和accept()函数中第二个参数的类型为 sockaddr

- sockaddr 和 sockaddr_in 的长度相同，都是16字节，sockaddr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。

IPv6 Address Structure

```
struct in6_addr{
    Union{//It is realized by the union structure on my system
        uint8_t __u6_addr8[16];//unsigned char
        uint16_t __u6_addr16[8];//unsigned short
        uint32_t __u6_addr32[4];//unsigned int
    }__in6_u;
};

struct sockaddr_in6{
    sa_family_t sin6_familiy;
    int_port_t sin6_port;
    uint32_t sin6_flowinfo;
    struct in6_addr sin6_addr;
    uint32_t sin6_scope_id;
};
```

Establish a Socket at the Server Side [6]

- The steps involved in establishing a socket on **the server side** are as following:
  - 1) Create a socket with the socket() system call.
  - 2) Bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number on the host machine.
  - 3) Listen for connections with the listen() system call.
  - 4) Accept a connection with the accept() system call. This call typically **blocks** until a client connects with the server.
  - 5) Send and receive data

Linux下的socket演示程序
server.c

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

int main(){
    //创建套接字，it just creates interface
    int serv_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP); //如果serv_sock < 0, 则创建失败，最好有检查
    //将套接字和IP、端口绑定
    struct sockaddr_in serv_addr;                              //IPv4地址结构体
    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");        //服务器IP地址(这里使用的是本机地址，INADDR_ANY)
    serv_addr.sin_port = htons(1234);                          //端口
    bind(serv_sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)); //这里地址强制转换
    //进入监听状态，等待用户发起请求
    listen(serv_sock, 20);                          //20表示服务器端队列长度，可以有20个clients等待连接
    //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);
    //向客户端发送数据
    char str[] = "Hello Client!";
    write(clnt_sock, str, sizeof(str));
    //关闭套接字
    close(clnt_sock); close(serv_sock); return 0;
}
```

More about accept()

```
int clnt_sock = accept(serv_sock, (struct sockaddr *)&clnt_addr, &clnt_addr_size);
```

- 1) accept() returns a new socket file descriptor for the purpose of reading and writing to the client. The original file descriptor is used usually for listening for new incoming connections.
- 2) It dequeues the next connection request on the queue for this socket of the server. If queue is empty, this function **blocks** until a connection request arrives.
- 3) Note that the last parameter of this function is a pointer. You are not specifying the length, the kernel is and returning the value to your application, the same with the `clnt_addr`. After a connection with a client is established the address of the client must be made available to your server, otherwise how could you communicate back with the client? Therefore, the accept function call fills in the address structure and length of the address structure for your use.

More about accept()

```
int clnt_sock = accept(serv_sock, (struct sockaddr *)&clnt_addr, &clnt_addr_size);
```

- 4) When a server receives (accepts) the client's connection request ⇒ it forks a copy of itself and lets the child handle the client. (make sure your remember these Operating Systems concepts) Therefore on the server machine, listening socket is distinct from the connect socket.

write() and read()

```
int write(int file_descriptor, const void *buf, size_t message_length);
int  read(int file_descriptor, char *buffer, size_t buffer_length);
```

- The return value of write() function is the number of bytes written, and –1 for failure.
- What this function does is transfer the data from your application to a buffer in the kernel on your machine, it does **NOT** directly transmit the data over the network. TCP is in complete control of sending the data and this is implemented inside the kernel.
- The value returned by the read() function is the number of bytes read which may not be buffer length. It returns –1 for failure.
- read() only transfers data from a buffer in the kernel to your application, you are not directly reading the byte stream from the remote host, but rather TCP is in control and buffers the data for your application.

Specifying Address

- 把ip地址转化为用于网络传输的二进制数值
  - int inet_aton(const char *cp, struct in_addr *inp);
  - inet_aton() 转换网络主机地址ip(如192.168.1.10)为二进制数值，并存储在struct in_addr结构中，即第二个参数*inp, 函数返回非零值表示cp主机地址有效，返回0表示主机地址无效。
  - 这个转换完后不能用于网络传输，还需要调用htonl函数才能将主机字节顺序转化为网络字节顺序。

- 将网络传输的二进制数值转化为成点分十进制的ip地址
  - char *inet_ntoa(struct in_addr in);
  - inet_ntoa 函数转换网络字节排序的地址为标准的ASCII以点分开的地址,该函数返回指向点分开的字符串地址（如192.168.1.10)的指针，该字符串的空间为静态分配的，这意味着在第二次调用该函数时，上一次调用将会被重写（复盖），所以如果需要保存该串最好复制出来自己管理！

Establish a Socket at the Client Side [6]

- The steps involved in establishing a socket on **the client side** are as following:
  - 1) Create a socket with the socket() system call
  - 2) Connect the socket to **the address of the server** using the connect() system call
  - 3) Send and receive data. There are a number of ways to do this, but the simplest is to use the read() and write() system calls.

- Notice that the client needs to know of the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established.
- Notice also that once a connection is established, both sides can send and receive information.

Linux下的socket演示程序
client.c

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

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

    //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");//服务器IP地址(这里是本机地址)
    serv_addr.sin_port = htons(1234);                            //端口
    //connect()函数执行TCP三次握手建立联接
    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);
    printf("Message form server: %s\n", buffer);
    //关闭套接字
    close(sock); return 0;
}
```

SOCKET PROGRAMMING WITH TCP IN WINDOWS

Windows 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。

- Linux下socket函数
- int socket(int af, int type, int protocol);

Window:

Generic:

```
struct SOCKADDR {
    unsigned short sa_family;
    char           sa_data[14];
};
```

IPv4 address struct:

```
struct sockaddr_in {
     short          sin_family;
     unsigned short sin_port;
     struct in_addr sin_addr;
     char           sin_zero[8];
};
struct in_addr {
   union {
      struct {unsigned char s_b1, s_b2, s_b3, s_b4;} S_un_b;
      struct {unsigned short s_w1, s_w2;} S_un_w;
      unsigned long S_addr;
   } S_un;
};
```

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

```
#include <stdio.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);
    //创建套接字
    SOCKET servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    //绑定套接字
    struct 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");       //可以改成具体的服务器IP地址
    sockAddr.sin_port = htons(1234);                         //端口
    bind(servSock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));
    //进入监听状态
    listen(servSock, 20);
    //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);
    //关闭套接字
    closesocket(clntSock); closesocket(servSock);
    //终止 DLL 的使用
    WSACleanup(); return 0;
}
```

Window下的socket演示程序客户端 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);
    //创建套接字
    SOCKET sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    //向服务器发起请求
    struct 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");    //这里要填服务器的ip地址!
    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。

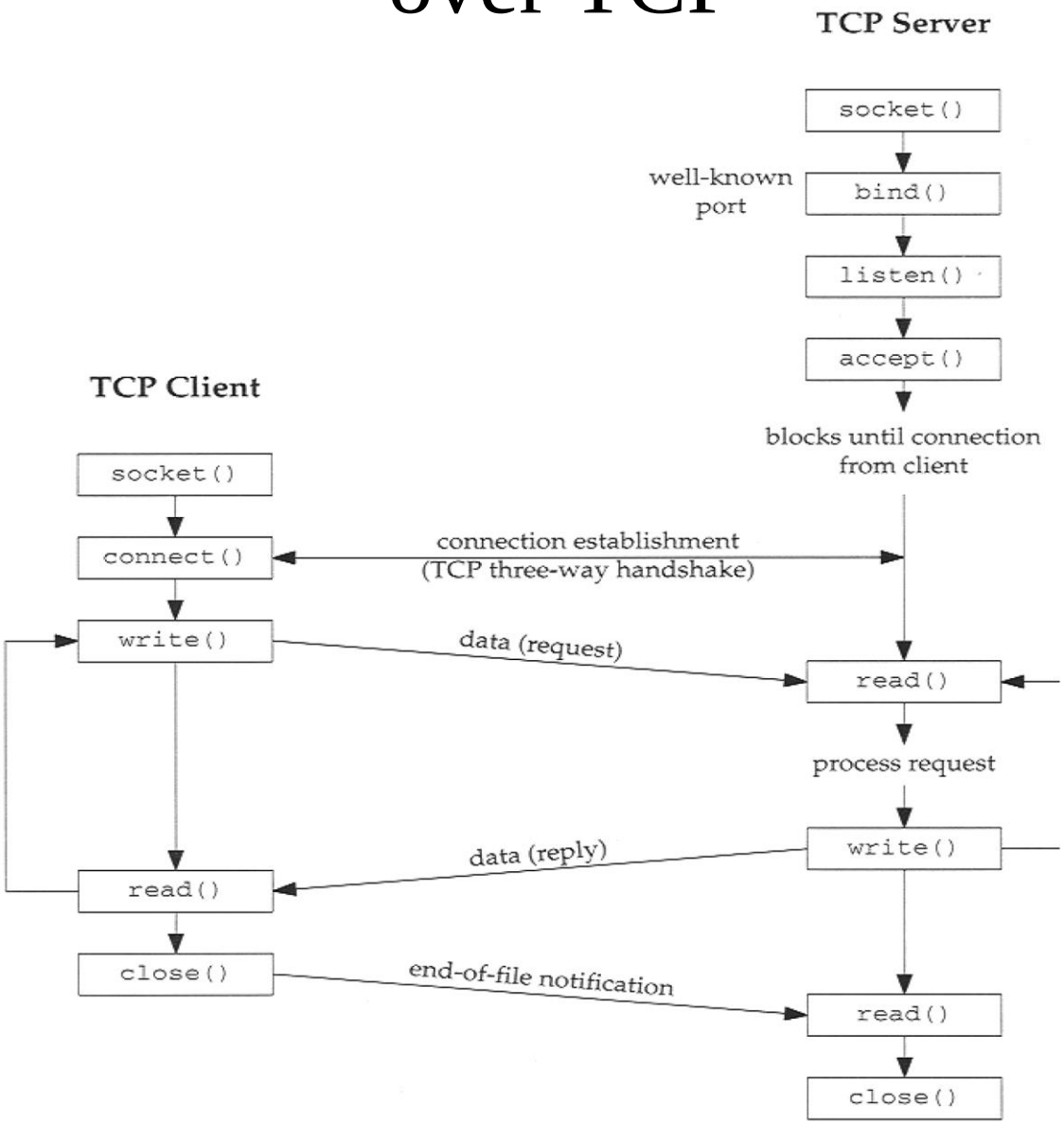
int socket(int af, int type, int protocol);

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。

Outline of a Client-Server Network Interaction over TCP



Client-Server Communication

- Communication via sockets necessitates existence of the 4-tuple:
  - Local IP address
  - Local Port#
  - Foreign IP address
  - Foreign Port#

- Server
  - Passively waits for and responds to clients
  - Passive socket

- Client
  - Initiates the communication
  - Must know the address and the port of the server
  - Active socket

Last but not Least

- The server executes first and waits to receive;
- The client executes second and sends the first network packet to the server.
- After initial contact, either the client or the server is capable of sending and receiving data.

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] J. Kurose and K. Ross, Computer Networking — A top-down approach, 5th edition, Pearson Education Inc., 2010. (§2.7 §2.8 pp.160-178)
- [3] A.S. Tanenbaum and D.J. Wetherall, Computer Networks, 5th edition, Prentice Hall, 2011.
- [4] https://docs.microsoft.com/en-us/windows/win32/api/winsock/nf-winsock-ntohl
- [5] https://www.geeksforgeeks.org/data-structure-alignment-how-data-is-arranged-and-accessed-in-computer-memory/
- [6] http://alumni.cs.ucr.edu/~ecegelal/TAw/socketTCP.pdf