Socket网络编程实验报告

1. 实验原理

1.1. UDP协议概述

1.1.1. 概述

1️⃣数据报：UDP的段

2️⃣UDP完成工作：有且仅有多路复用/分解+差错检测，也就是传输层最基本的工作，直接和IP交互

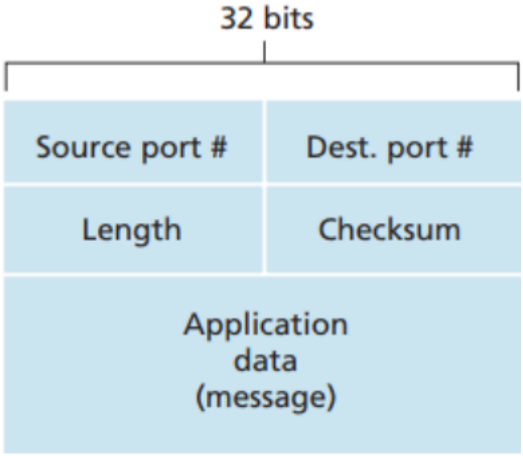
3️⃣UDP的特点

1. 简单无连接：传输数据前双方不事先连接
2. 是best effort服务：尽最大努力传输，但不保证可靠性/顺序
3. 速度快
4. 首部小：只有8字节，开销小

4️⃣UDP的应用：NFS，流式多媒体，DNS

1.1.2. UDP报文段结构

段头+数据，其中段头只有32bit，含源端口号+目的端口号+长度+校验



1.1.3. UDP校验&checksum

1️⃣目的：检测UDP报文从源到目的过程中，是否发生改变

2️⃣特点：检错能力很弱

3️⃣检错步骤：将段的内容看作16为二进制数集合

- 发送端：
    1. 获得校验和：所有16位二进制数相加→截取低16位→结果按位取反
    2. 将校验和输入UDP校验和字段Checksum
- 接收端
    1. 对段内容所有16位二进制相加，截取低16位

2. 将结果+Checksum，如果全为1111111111111111就没出错

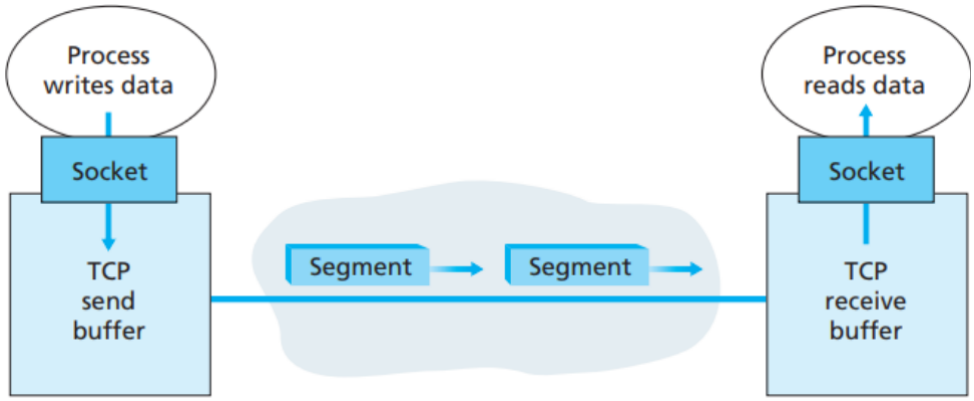
1.2. TCP协议概述

1.2.1. TCP概览

1 特点

1. 点对点：发送/接收方都只能有一个
2. 可靠的
3. 无报文边界：字节以有序流发送数据，而不是分成独立报文
4. 流水线式：通过设置窗口大小，实现拥塞/流量控制
5. 全双工：数据可以同时双向传输，UDP同样

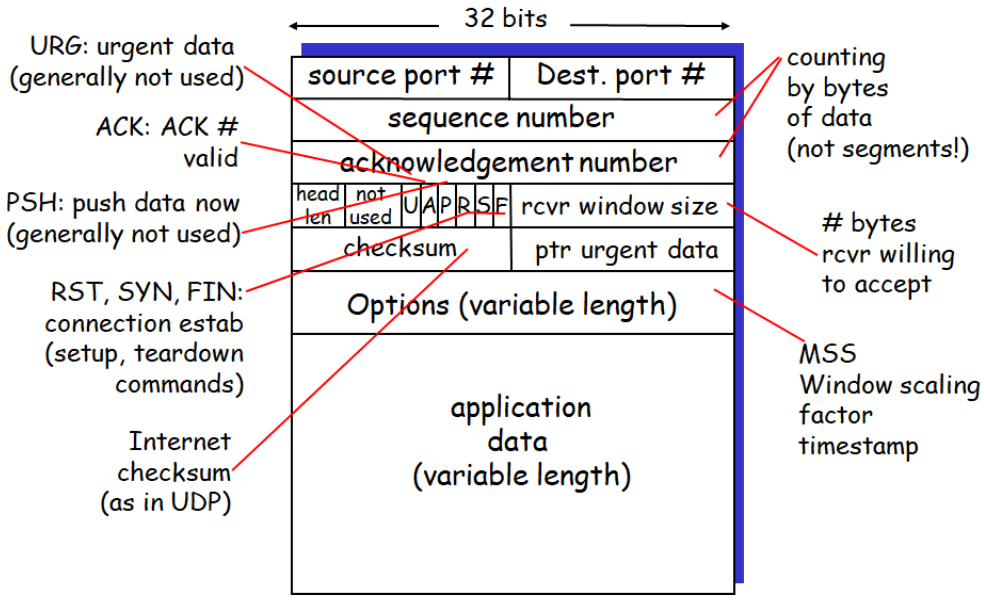
2 结构概览



3 功能：三控一管，可靠性/流量/拥塞控制，连接管理

1.2.2. TCP段文

1.2.2.1. 段文格式：一共五行20字节



1 第一行：源端+目的端口号

2 第二行：序列号，TCP的传输是一个个字节流送的，要给每个字节编号保证按序交付

3️⃣ 第三行：确认号，TCP有确认机制(接收端$\xrightarrow{ACK}$发送端，当ACK=N则N-1及以前的数据都已收到

4️⃣ 第四行：

1. 首部长度
2. 保留字段：占6位，忽略不计
3. 标志位

标志位	标志位=0	标志位=1
U(URG紧急)	紧急指针字段无效	有效，报文中有紧急数据，优先级高
A(ACK确认)	确认号字段无效	有效，TCP连接建立后，所有ACK=1
P(PSH推送)	\	收到PSH=1的报文，会优先上交给应用进程
R(RST复位)	\	当RST=1时，说明TCP连接崩溃，需要释放连接重传
S(SYN同步)	\	当SYN=1时，表示整个报文是一个连接请求/连接接收报文
F(FIN终止)	\	当FIN=1时，表示数据传输结束，就地释放连接

4. 窗口字段：**明确指定了目前允许对方发送的数据量**

5️⃣ 第五行

1. 校验和字段：检测范围包括首部+数据，计算校验和时需要在TCP报头前加上12B伪首部
2. 紧急指针字段：前面已说，由URG控制

6️⃣ 第六行开始：选项字段，长度可变，内容可选(最早的内容为MSS)

7️⃣ 最尾部：填充字段，填充使整个首部长度为32bit整数倍

1.2.2.2. 其他

1️⃣ 最大段大小(MSS)：536字节

2️⃣ TCP段文$\xleftrightarrow{\text{协同工作}}$IP数据报

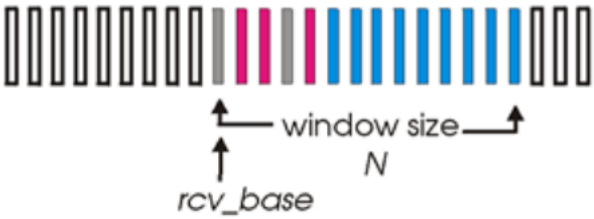
1. TCP报文首部不包含IP地址，IP地址在IP数据报首部
2. IP层负责处理目的IP：TCP协议在发送数据时，依赖于IP协议来确定数据报的目的地

1.2.3. TCP的可靠性控制：丢包重传

1.2.3.1. 发送窗口(发送缓冲区)

1. 数据结构：开始指针send_base + 窗口大小n + 下一个序列号nextseqnum
2. 发送窗口的行为
    1. 应用层请求通过传输层发送数据，先检查nextseqnum是否有效
    2. 有效的话，就将nextseqnum封装成TCP段发送，同时nextseqnum++
    3. 一直移动到nextseqnum - send_base = N(窗口满)，无法继续发应用层塞的数据了
    4. 直到接收方发回ACK，根据ACK，send_base右移，跳过已确认收到的段
3. 丢失段和发送窗口
    1. 发送窗口一定包含丢失段
    2. 如何判定窗口中的段已丢失：发回的ACK包含了段序号，超时后ACK还没某段序号就重传

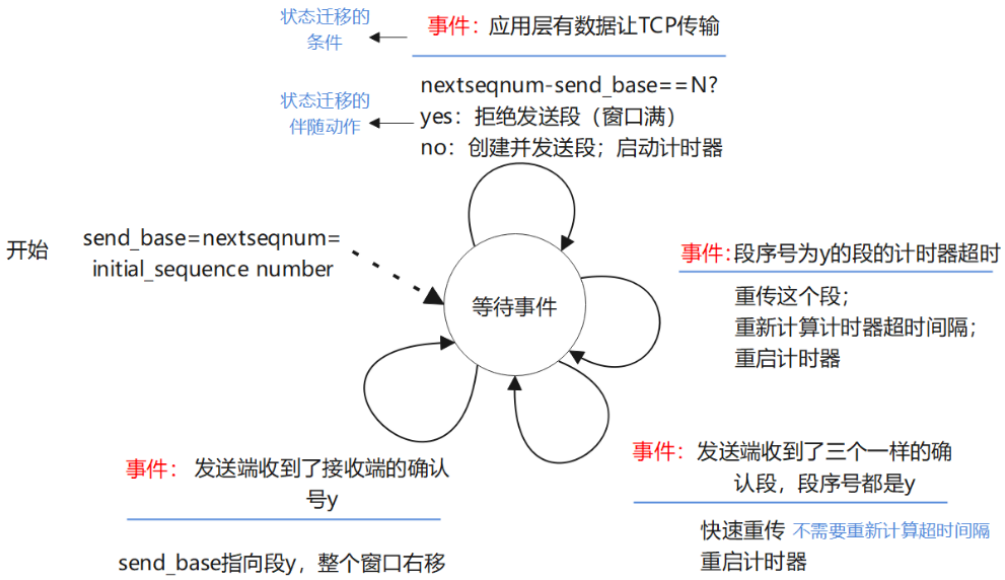
1.2.3.2. 接收窗口(接收缓冲区)



1. 数据结构：开始指针rcv_base + 窗口大小n，其中rcv_base指向
2. 核心机制：**通过发送期待接收下一个段的序列号，以此来确认收到当前段**
3. 接收窗口的行为
    1. rcv_base指向接收方期待收到的段的段号(第一个灰色)
    2. 串口内不断收到段(红色)，在期待段到达前，先将红色的缓存起来(蓝色的为空闲接收缓存)
    3. 期待段到后，连通所有红段都一起上交应用层
    4. rcv_base来到下一个期待收到的段(第二个灰色处)

1.2.3.3. 发送端的可靠性控制

1. 控制机制：
    1. 只有一个状态即等事件，状态迁移结尾等事件→等事件
    2. 假设无流量/拥塞控制，应用层给到的数据长度小于MSS，数据传输单向



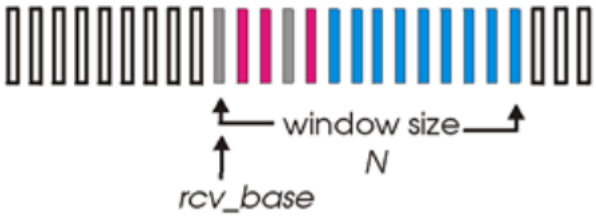
2 快速重传：代表了轻度拥塞(超时对应重度拥塞)

1. 接收端没收到期待包，不论接下来收到了什么都只会返回期待包的ACK，因为TCP要求数据有序
2. 当返回相同ACK三次，就可认为这个ACK所代表的包已经丢了
3. 启动快速重传

3 伪代码

```
send_base = init_sequence number
nextseqnum = init_sequence number
loop(永远){
    switch(事件)
    事件:应用层有数据让TCP传输
        if(nextseqnum-send_base<N){
            创建段序号为nextseqnum的段
            启动计时器
            将段发给IP层
            nextseqnum = nextseqnum +数据长度/*段序号是跳跃式的*/
        }else{
            拒绝发送段
        }
    事件:段序号为y的段的计时器超时
        重传这个段y
        重新计算计时器超时间隔
        重启计时器
    事件:接收到ACK，字段值为y
        if(y>send_base){/*段在发送窗口内*/
            取消掉段y之前所有的段的计时器
            send_base = y/*窗口右移*/
        }else{/*这里指的是y=send_base，接收端还没有收到y*/
            对ACK字段为y的计数器+1
            if(计数器的值==3){
                快速重传段y
                重启段y的计时器
            }
        }
}
```

1.2.3.4. 接收端的可靠性控制



收到段的特征	TCP接收端动作
有序到达 无间隙 其他段都已确认	延迟等待下一个段0.5s 1. 期间如果下一段来了则二者一起确认 2. 没来的话就发送ACK
有序到达 无间隙 有一个ACK在做延时	就是上述"下一段"，二者立刻一起确认ACK
乱序到达 有间隙 (如上图红色部分)	立即发ACK，内容为期待段的段号
乱序到达 但填满了某些间隙	目的在于补齐gap，间隙成因有两种(期待段空缺+其他乱序段间空缺) 1. 收到期待段(左灰)：连同右边连着的红段送给应用层，窗口右移 2. 收到其他段(右灰)：收到段变红，返回ACK(内容为期待段段号)
收到段位于窗口左侧	丢弃段

1.2.3.5. TCP往返时间和超时

1 一些概念

1. 超时时间间隔：数据发送后，在这个时间内还未收到ACK，就要重传
2. RTT：数据包的往返时间，理想超时时间间隔应该略大于RTT
3. 安全边际：略大于RTT，略大于的这部分就是安全边际

2 如何预测RTT：一般$\alpha = 1/8, \beta = 1/4$

1. $\text{sample}RTT$：策略从传输→收到ACK的时间差，缺点是波动大
2. $\text{estimate}RTT$：用EWMA平滑估计RTT

   $$\text{estimate}RTT_n = (1-\alpha)\text{estimate}RTT_{n-1} + \alpha\text{sample}RTT$$
3. deviation：估计前两者的差距，来反映波动性大小

   $$\text{deviation}_n = (1-\beta)\text{deviation}_{n-1} + \beta|\text{sample}RTT - \text{estimate}RTT|$$

3 超时时间间隔$= \text{estimate}RTT + 4 * \text{deviation}$

1.2.4. TCP流量控制

1.2.4.1. 概述

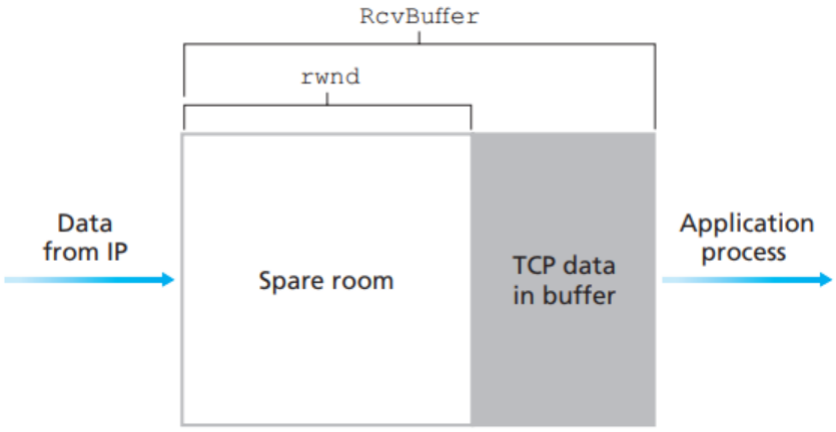
1️⃣背景：发送太快，接收端应用程序读取太慢，就会造成接收窗口溢出

2️⃣流量控制的含义：让发送方发送速率=接收方应用程序读取速率，防止溢出

3️⃣核心机制：

1. TCP段头中有一个字段表示接收窗口大小
2. 接收窗口会给发送方指明，接收方还有多少可用缓存

1.2.4.2. 控制过程



1️⃣接收端行为：A通过TCP连接向B发文件，B为该连接分配接收缓存，B的应用不断从缓存取走数据

2️⃣接收有关变量

1. `LBRead`：缓存中被读走的最后一个段的段号
2. `LBRcvd`：缓存中刚收到的段的段号
3. `RcvBuffer`：缓存大小，满足`LBRcvd-LBRead ≤ RcvBuffer`
4. `rwnd`：接收窗口(空闲缓存)，等于`RcvBuffer-[LBRcvd-LBRead]`，初始为`rwnd=RcvBuffer`

⚠️主机A需要明白B的`rwnd`还有多大，通过将`rwnd`放到B传回给A的报文的接收窗口字段即可

3️⃣发送端有关变量

1. `LBSent`：最后一个被送出的字节
2. `LBAck`：最后一个被确认的字节，`LBAck-LBSent`就是发出但未收到确认的数据

4️⃣流量控制的核心：在主机A的整个生命周期，保证LBAck-LBSent ≤ rwnd

1.2.4.3. 零窗口探测

1️⃣Bug所在：

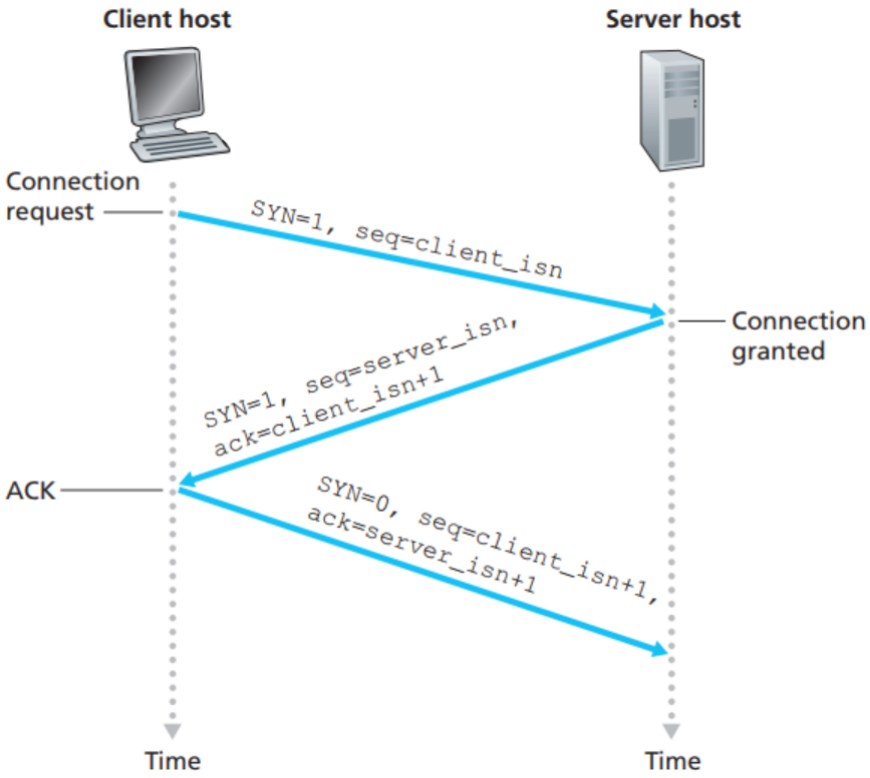
1. 当B端`rwnd=0`时，发送端A必定会停止发送
2. B然后不会向A反馈`rwnd=0`的变化，因为TCP只在发信/ACK时才发送报文
3. 之后B的应用程序会取走缓存使得`rwnd>0`，但是A就也无法得知了

2️⃣ Debug所在

1. 不论`rwnd=0`与否，A都定期发送只包含1字节的探测报文段
2. 接收了探测报文的B端，无论如何都不可能是零窗口了，即可恢复数据传输

1.2.5. TCP的连接管理

1.2.5.1. 开启连接：三次握手



(Client=发送端，Serve=接收端)

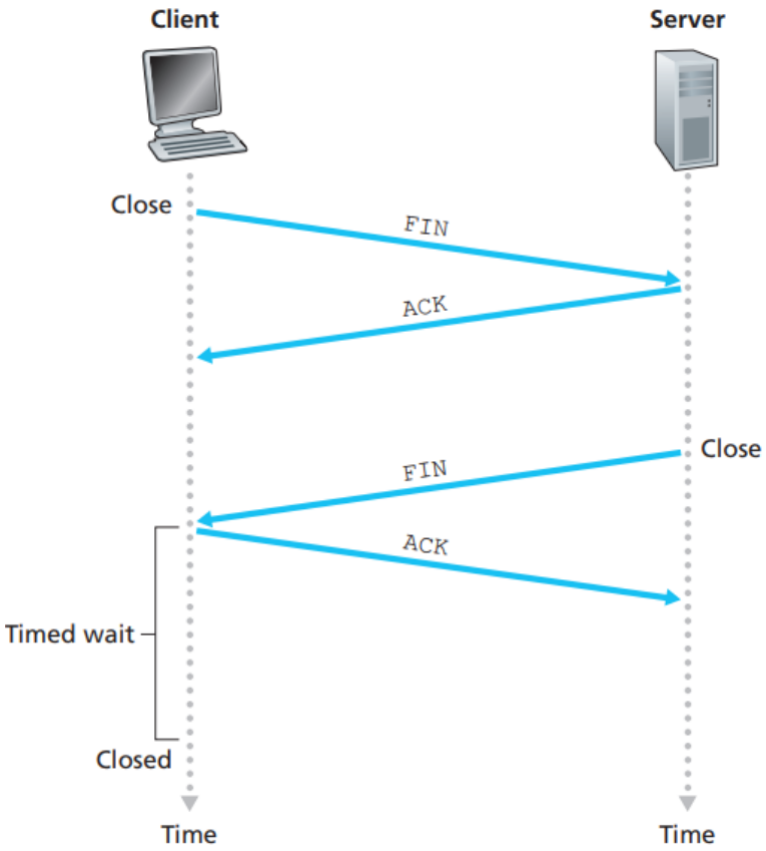
握手	方向	报文类型	Client初始序列号	Server初始序列号	确认号
第一次	C→S	SYN	`J`	\	\
第二次	S→C	SYN+ACK	\	`K`	`J+1`
第三次	C→S	ACK	\	\	`K+1`

1️⃣ 三次握手后TCP连接建立，Clinet开辟缓存，开始传输

2️⃣ 补充说明：`J`和`K`是随机生成，ACK/SYN/FIN报文的报头对应标志位为1

3️⃣ DDos攻击：永远吊在第三次握手未完成状态。服务器不断开辟内存，直到服务器崩溃

1.2.5.2. 关闭连接：四次握手

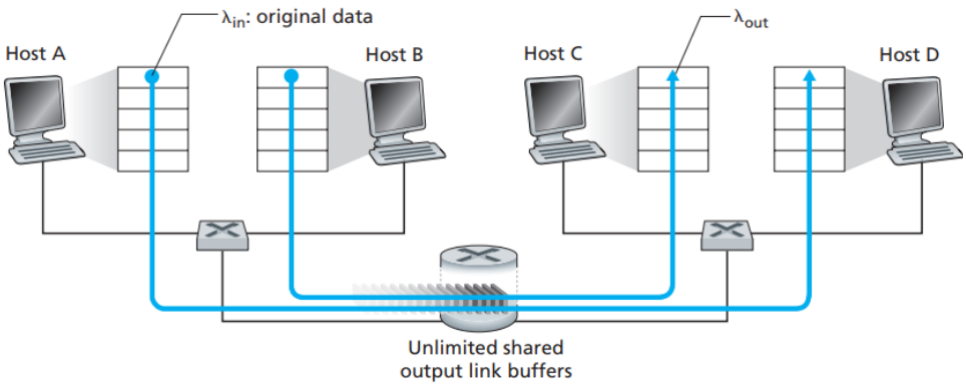


握手	方向	报文类型	C-S行为
第一次	C→S	FIN	所以爱会消失的，结束吧
第二次	S→C	ACK	好的
第三次	S→C	FIN	那就结束吧
第四次	C→S	ACK	好的，达成共识

四次握手后，要等一段时间TCP连接才正真关闭

1.2.6. 拥塞控制

1.2.6.1. 拥塞控制概述



1 拥塞：是指路由器的拥塞

2 表现：丢包(路由器缓冲区溢出)，长延迟(一直在路由器缓冲区中排队)

3 成因：

1. 长延时：发送端发送总速度>路由器交换能力，**延时太大了会被误以为丢包，而进行无意义重传**
2. 丢包：发送端塞给路由器的数据>路由器有限的缓存

4️⃣ 分类

1. 网络帮助的拥塞控制：交换机检测到拥塞后直接控制，可为交换机→发端，交换机→收端→发端
2. 端到端的拥塞控制：端系统功能强(路由器相对弱)，端根据网络反馈调节拥塞(超时/快速重传)

1.2.6.2. ATM(异步传输模式)的拥塞控制

Old Fashined

1️⃣ ATM业务类型

类型	名称	描述	特点
ABR	Available Bit Rate	有效位率服务，用于视频	可能丢包，保证最小带宽
CBR	Constant Bit Rate	用于实时语音通信	不丢包，不拥塞控制
VBR	Variable Bit Rate	变动位率服务	不丢包，不拥塞控制
UBR	Unspecified Bit Rate	有资源则使用，无资源则丢包	免费使用，无拥塞控制

2️⃣ 信元：ATM的数据单元

1. 数据信元
2. 资源管理信元：存放拥塞信息，同工厂几十个信元里就有一个资源管理信元

3️⃣ ATM的ABR拥塞控制方法

1. 信元头部加CI(拥塞指示)和NI位(不增加速率)：拥塞后CI=1发端会降低发送速率，NI用于让发端速率不再增加
2. ER设置：这是在资源管理信元的字段，告诉发端可以按多大速率发数据，有多个则取最小
3. EFCI：位于数据信元，检测到拥塞后置1

1.2.6.3. TCP的拥塞控制

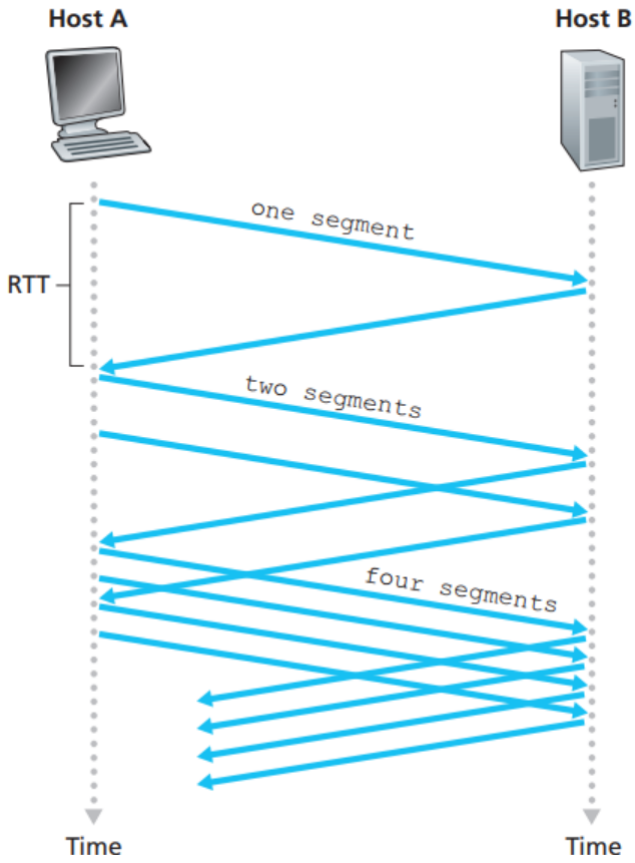
1️⃣ 两种判断：超时(重度拥塞)，收到三个相同ACK(轻度拥塞)

2️⃣ 探测拥塞：慢启动(不断*2)+拥塞避免(改为+1)

```
第1个RTT：发1个探测段，收到1个ACK则没拥塞
第2个RTT：发2个探测段，收到2个ACK则没拥塞
第3个RTT：发4个探测段，收到4个ACK则没拥塞
.....n次试探后还没拥塞.....
第n+1个RTT：发$2^n$+1个测试段
```

3️⃣ TCP慢启动

1. 慢启动过程



2. 慢启动伪代码

```
threshold=适当的值(10、20...不要太大) //阈值，区分慢启动和拥塞避免
Congwin=1 //拥塞控制时，使用的窗口大小
for(每个确认段) //每收到一个ACK窗口就+1，第一轮1个ACK，第二轮2个，第三轮4个...
    Congwin++
until(丢包&&ngWin>=threshold)
```

4 拥塞避免

1. Tahoe拥塞避免算法伪代码：不合理的点在于没区分轻度/重度拥塞，一股脑将Congwin=1

```
//慢启动结束
while (没有丢包) {
  每w个段被确认:
       Congwin++//每个RTT，窗口+1线性增加
}
//如果丢包则退出循环
threshold = Congwin/2
Congwin = 1
//重启慢启动
```

2. Reno拥塞算法伪代码：吞吐率更高，震荡更小

```
//慢启动结束
while (没有丢包) {
  每w个段被确认:
       Congwin++//每个RTT，窗口+1线性增加
```

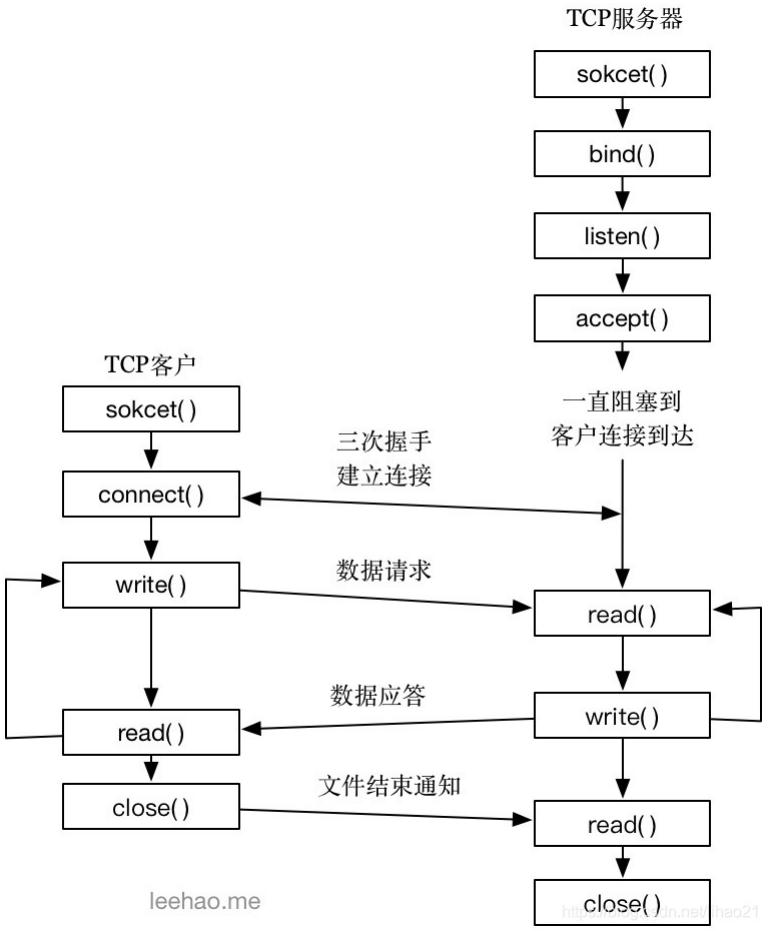
```
}
//如果丢包则退出循环
threshold = Congwin/2
if(因为超时丢包){//重度拥塞
    Congwin = 1
    重启慢启动
}
if(因为收到三个相同确认段丢包){//轻度拥塞，只需要快速恢复
    Congwin = Congwin/2
    goto: while循环
}
```

5 平均吞吐率≈ $\frac{1.22MSS}{RTT\sqrt{L}}$，$L$为丢包率，$MSS$为最大段大小

6 算法特点：

1. 线性增加窗口，丢包后指数减少窗口
2. 有效，收敛，公正，友好

1.3. Scoket(套接字)编程概述



1.3.1. 创建Socket

使用系统调用，创建一个Scoket，并返回其描述符

```
#include <sys/socket.h>
int fd = socket(int domain, int type, int protocol);
//当fd>0时，说明Scoket创建成功
```

1️⃣ `domain`：指定使用哪种协议族，`AF_UNIX`协议族用于本地Scoket，`AF_INET`用于网络Scoket

2️⃣ `type`：参与Socket的通信类型，有两种

1. `SOCK_STREAM`：流Scoket，基于TCP，提供可靠有序服务
2. `SOCK_DGRAM`：数据报Socket，基于UDP，提供不可靠无序服务

3️⃣ `proctol`：指定协议族中的一个具体

1.3.2. 命名Socket

1️⃣ 含义：给Socket关联一个IP+端口号

2️⃣ 目的：让创建的套接字可以被其他进程使用

3️⃣ 基于系统调用`bind`的实现：`*address`中的地址分配给`scoket`，地址长度为`address_len`

```
#include <sys/socket.h>
int bind(int socket, const struct sockaddr *address, size_t address_len);
```

4️⃣ `AF_INET`域的地址格式

```
struct sockaddr_in {
 short int sin_family;          // AF_INET
 unsigned short int sin_port;   // 端口号，无需指定时设为0
 struct in_addr  sin_addr;      // IP地址
};
```

1.3.3. 创建Socket队列

1️⃣ 目的：使用队列来保存服务器上未处理的请求

2️⃣ 基于`listen`系统调用的实现

```
#include <sys/socket.h>
int listen(int socket, int backlog);
//backlog是等待队列的最大值，再往后的请求就会被拒绝，常用为5
```

1.3.4. 客户请求连接服务器

基于`connect`的实现

```
#include <sys/socket.h>
int connect(int socket, const struct sockaddr *address, site_t address len);
```

1️⃣ 客户端Socket将连接到`address`指定的服务器Socket

2️⃣ `address`指定的结构长度由`address len`控制

3️⃣ 调用`connect`后会触发三次TCP握手

1.3.5. 服务器接受客户连接

基于`accept`的实现

```
#include <sys/socket.h>
int accept(int socket, struct sockaddr *address, socklen_t
*address_len);
```

1️⃣Scoket队列中没有未处理连接时，`accept`将阻塞直到有客户建立连接

2️⃣`accept`将创建一个新Socket来与客户通信，并返回Socket的描述符

1.3.6. 利用Socket收发字节数据

基于`read`和`write`系统调用来传输数据了

1️⃣向服务器发消息示例

```
const int MAXBUF = 256;
char buffer[MAXBUF] = "hello tcp";
int nbytes = write(client_fd, buffer, 10);
```

2️⃣服务器读取消息示例

```
const int MAXBUF = 256;
char buffer[MAXBUF];
int nbytes = read(client_fd, buffer, MAXBUF);
```

1.3.7. 关闭连接

触发TCP四次挥手

```
#include <unistd.h>
int close(int socket);
```

3. UDP协议的实现

3.1. `UDP-Client.cpp`

代码详细解释和代码如下

```
#include <iostream>
#include <string>
#include <thread>
#include <atomic>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <cstring>

/*定义三个atomic布尔变量，用于控制程序的不同状态
```

```
** 2. exit_flag用于指示程序何时退出
** 3. proof_flag用于指示用户是否通过了身份验证
** 4. recv_flag用于控制接收和发送数据的同步
*/
std::atomic<bool> exit_flag(false);
std::atomic<bool> proof_flag(false);
std::atomic<bool> recv_flag(true);

/*
** 等待退出命令的函数
** 不断检查exit_flag的状态，当exit_flag被设置为true时，关闭套接字并退出程序
*/
void wait_for_exit(int sock)
{
 while (!exit_flag.load()) {}
 close(sock);
 exit(0);
}

/*
** 接收来自服务器的数据
** 无限循环中执行，不断检查是否接收到从服务器发送的数据
*/
void CS_Server(int sock, sockaddr_in addr)
{
 /*
 ** 1. 用buffer作接收数据的缓冲区
 ** 2. 定义一个socklen_t类型的变量len，用于存储地址结构的长度，这对于recvfrom函数来说是必须的
 */
 char buffer[1024];
 socklen_t len = sizeof(addr);

 /*
 **不断检查和接收从服务器发送过来的数据
 */
 while (true)
 {
     /*
     ** 1. 检查退出标志exit_flag的状态。
     **    - 如果exit_flag为true，表示程序应当终止，这时候函数会返回，停止接收数据
     ** 2. 使用recvfrom函数从套接字sock接收数据。
     **    - buffer数组用于存储接收到的数据。
     **    - 1024定义了buffer的最大长度，即最多可以接收1024字节的数据。
     **    - (struct sockaddr *)&addr和&len提供了关于发送端(服务器)地址的信息。
     **    - recvfrom函数返回接收到的字节数，如果没有接收到数据则返回0。
     **    - 收到的数据被存储在buffer数组中。
     */
     if (exit_flag.load()) return;
     int received = recvfrom(sock, buffer, 1024, 0, (struct sockaddr *)&addr, &len);

     if (received > 0)
     {
```

```
64
65          /*
66          ** 1. 确保接收到的数据以空字符结束，形成一个有效的C字符串。
67          ** 2. 将接收到的数据转换为std::string类型，便于后续处理
68          */
69          buffer[received] = '\0';
70          std::string data(buffer);
71
72          /*
73          ** 1. 如果身份验证已经通过（proof_flag为true），
74          **    则在控制台输出从服务器接收到的数据，并用"服务器:"标签指示这些数
    据来自服务器
75          **    然后设置接收标志为true，表示客户端已准备好发送下一条数据
76          ** 2. 如果身份验证还未通过，直接输出接收到的数据
77          ** 3. 当接收到特定消息（"信息正确!\r\n"）时，
78          **    表明身份验证成功，设置proof_flag为true。
79          **    然后在控制台提示"客户端:"，表示现在可以从客户端发送数据。
80          */
81          if (proof_flag.load())
82          {
83              std::cout << "服务器: " << data << std::endl;
84              recv_flag.store(true);
85          }
86          else
87          {
88              std::cout << data;
89          }
90          if (data == "信息正确!\r\n")
91          {
92              proof_flag.store(true);
93              std::cout << "客户端: ";
94          }
95      }
96  }
97  }
98
99
100 /*
101 ** 持续在一个循环中运行，用于发送数据到服务器
102 */
103 void CS_Client(int sock, sockaddr_in addr)
104 {
105  //在无限循环中运行，持续监听用户输入并发送到服务器
106  while (true)
107  {
108      /*
109      ** 1. 检查退出标志exit_flag，如果为true，则终止函数
110      ** 2. 不退出的话，就创建一个可以存储和处理用户输入的字符串变量data
111      */
112      if (exit_flag.load()) return;
113      std::string data;
114
115      /*
116      ** 1. 检查身份验证标志proof_flag。
117      **    - 如果已验证(proof_flag为true)，则等待接收标志recv_flag为
    true。
```

```
     ** 2. 循环等待，直到recv_flag变为true
     **    - 表示客户端已经准备好接收用户的下一个输入
     ** 3. 打印提示信息，获取一行用户文本，将输入存储到字符串data中
     ** 4. 将recv_flag设置为false，表示当前的用户输入已经获取，并准备发送到服务器
     */
     if (proof_flag.load())
     {
         while (!recv_flag.load()) {}
         std::cout << "客户端: ";
         std::getline(std::cin, data);
         recv_flag.store(false);
     }

     /*
     ** 1. 如果用户还未通过身份验证(proof_flag为false)
     ** 2. 也获取用户输入的数据，然后准备发往服务器
     */
     else
     {
         std::getline(std::cin, data);
         recv_flag.store(false);
     }

     /*使用sendto函数将用户输入的数据发送到服务器
     **  - data.c_str()获取字符串的C风格表示，适用于sendto函数
     **  - data.length()提供要发送的数据长度
     **  - (struct sockaddr *)&addr和sizeof(addr)提供服务器的地址信息
     */
     sendto(sock, data.c_str(), data.length(), 0, (struct sockaddr *)&addr, sizeof(addr));

     /*检查用户是否输入了"exit"命令，以决定是否退出程序
     **  - 如果输入了"exit"，则设置退出标志exit_flag为true
     **  - 然后退出函数，进而终止客户端程序
     */
     if (data == "exit")
     {
         exit_flag.store(true);
         return;
     }
 }
}

int main() {
 /*
 ** 1. 创建UDP套接字。
 **    - AF_INET指定IPv4协议
 **    - SOCK_DGRAM指定数据报套接字(UDP)
 **    - 如果创建套接字失败(返回值小于0)，则输出错误信息并退出程序
 */
 int client_socket = socket(AF_INET, SOCK_DGRAM, 0);
 if (client_socket < 0) {
     std::cerr << "Socket creation failed." << std::endl;
     return -1;
 }
```

```
172
173  /*
174  ** 2. 设置服务器地址。
175  **    - sin_family设置为AF_INET，指定IPv4地址
176  **    - sin_port设置服务器端口号，htons(1212)将主机字节顺序转换为网络字节顺
     序
177  **    - inet_pton转换IP地址为网络字节顺序，这里使用的是本地地址(127.0.0.1)
178  */
179  sockaddr_in server_addr;
180  server_addr.sin_family = AF_INET;
181  server_addr.sin_port = htons(1212);
182  inet_pton(AF_INET, "127.0.0.1", &server_addr.sin_addr); // 使用
     localhost
183
184  /*
185  ** 3. 向服务器发送初始消息"Hello!"。
186  **    - sendto用于发送数据到指定的地址
187  */
188  sendto(client_socket, "Hello!", strlen("Hello!"), 0, (struct
     sockaddr *)&server_addr, sizeof(server_addr));
189  std::cout << "Connect Succeed!" << std::endl;
190
191  /*
192  ** 4. 启动三个线程处理不同的任务:
193  **    - exit_thread负责监听退出命令
194  **    - server_thread负责处理来自服务器的消息
195  **    - client_thread负责处理客户端用户的输入
196  */
197  std::thread exit_thread(wait_for_exit, client_socket);
198  std::thread server_thread(CS_Server, client_socket, server_addr);
199  std::thread client_thread(CS_Client, client_socket, server_addr);
200
201  /*
202  ** 5. join这三个线程，等待它们的完成
203  **    - 这保证了主函数会等待所有线程完成它们的任务
204  */
205  exit_thread.join();
206  server_thread.join();
207  client_thread.join();
208
209  return 0;
210 }
```

3.2. UDP-Server.cpp

代码详细解释和代码如下

```
1 #include <iostream>
2 #include <string>
3 #include <thread>
4 #include <vector>
5 #include <map>
6 #include <sys/socket.h>
7 #include <netinet/in.h>
8 #include <arpa/inet.h>
```

```cpp
#include <unistd.h>
#include <cstring>


/*
** 1. 定义用户结构体，存储用户消息(用户名+密码)
** 2. 定义User类型的向量，用于存储一系列的用户，并且初始化一个用户名DHY密码dhy的用户
** 3. 定义Client_Message，以一整型为关键字，存贮与特定客户端有关的消息
*/
struct User
{
std::string username;
std::string pwd;
};
std::vector<User> Users = {{"DHY", "dhy"}};
std::map<int, std::vector<std::string>> Client_Message;

/*
** 定义服务器套接字+服务器地址和客户端地址
*/
int server_socket;
sockaddr_in server_addr, client_addr;

/*
** 等待退出命令的函数
** 功能：等待用户在命令行输入exit，然后退出
*/
void wait_for_exit()
{
std::string input;
while (true) {
   std::cin >> input;
   if (input == "exit")
   {
       close(server_socket);
       exit(0);
   }
}
}

/*
** 在服务器端完成一个简单的数学运算然后传回客户端
** 客户端传来x时，服务器就返回给客户端x * x + 2 * x + 1
*/
float F(float x)
{
return x * x + 2 * x + 1;
}


/*
** 处理客户端请求的线程函数
*/
void CS_thread(sockaddr_in client_addr)
{
```

```
/*
** 1. 声明一个长1024的数组buffer，用于缓存从客户端接收的数据
** 2. 定义len变量，并初始化为客户端地址长，当作收发数据的地址结构长
** 3. 定于proof_flag用于判断用户是否通过了身份验证
*/
char buffer[1024];
socklen_t len = sizeof(client_addr);
bool proof_flag = false;

/* 当连接成功后
** 1. 向服务器的控制台打印连接成功的提示信息
** 2. 打印出连接的客户端的IP地址，inet_ntoa将二进制IP转化为十进制
** 3. 用sendto在客户端终端打印出欢迎信息
*/
std::cout << "Connect Succeed!" << std::endl;
std::cout << "Client Address: " << inet_ntoa(client_addr.sin_addr)
<< std::endl;
sendto(server_socket, "欢迎访问1212服务器!\r\n", strlen("欢迎访问1212服
务器!\r\n"), 0, (struct sockaddr *)&client_addr, len);

/*
** 负责处理客户端的身份验证
** 通过一个无限循环来实现这个过程
*/
while (true)
{
   /* 这一段是为了接收用户名
   ** 1. 通过sendto函数，服务器向客户端发送一个提示，要求输入用户名
   ** 2. 使用recvfrom函数接收客户端用户输入(发来)的用户名，并将用户名放在
buffer中
   ** 3. 在服务器端的控制台中，打印出从客户端发来的用户名
   */
   sendto(server_socket, "用户名: ", strlen("用户名: "), 0, (struct
sockaddr *)&client_addr, len);
   recvfrom(server_socket, buffer, 1024, 0, (struct sockaddr
*)&client_addr, &len);
   std::string username(buffer);
   std::cout << "用户名: " << username << std::endl;

   /*这一段是为了接收用户在客户端输入的密码
   ** 1. 服务器再次使用sendto函数向客户端发送一个提示，要求输入密码
   ** 2. 使用recvfrom函数来接收客户端发送的密码，密码也被存储在buffer数组中
   ** 3. 在服务器端的控制台中，打印出从客户端输入并且发来的密码
   */
   sendto(server_socket, "密码: ", strlen("密码: "), 0, (struct
sockaddr *)&client_addr, len);
   recvfrom(server_socket, buffer, 1024, 0, (struct sockaddr
*)&client_addr, &len);
   std::string pwd(buffer);
   std::cout << "密码: " << pwd << std::endl;

   /*这一段是为了身份验证
   ** 1. 循环遍历服务器维护的用户数据库向量User，这个向量中存储了有效的用户名和
密码组合
   ** 2. 检查客户端所提供的用户名和密码，是否和某个表项相符
```

```
    ** 3. 如果找到相符的，则更改proof_flag，在服务器端输出验证成功消息，然后通过sendto向客户端发送确认
    */
    for (const auto& user : Users)
    {
        if (user.username == username && user.pwd == pwd)
        {
            proof_flag = true;
            std::cout << "Proof Success!" << std::endl;
            sendto(server_socket, "信息正确!\r\n", strlen("信息正确!\r\n"), 0, (struct sockaddr *)&client_addr, len);
            break;
        }
    }

    /*身份验证有误时
    ** 1. 遍历所有表项后都没找到的话，说明输入账号密码有误
    ** 2. 先在服务器端输出错误提示
    ** 3. 通过sendto函数向客户端发送错误提示，要求重新输入用户名和密码
    */
    if (!proof_flag)
    {
        std::cout << "Proof Defeated!" << std::endl;
        sendto(server_socket, "用户名或密码错误,请再次输入!\r\n", strlen("用户名或密码错误,请再次输入!\r\n"), 0, (struct sockaddr *)&client_addr, len);
    }
    else
    {
        break;
    }
}

/*
** 登陆完成后，C-S就可进行交互了
** 以下程序负责处理从客户端接收到的请求
** 通过一个无限循环，用于不断接收和处理来自客户端的请求
*/
while (true)
{
    /*
    ** 1. 在接收客户端发来的数据之钱，用memset函数清空缓存buffer，保证每次接收新数据时缓冲区都是干净的
    ** 2. 使用recvfrom函数接收客户端发送的数据，接收到的数据存储在buffer中
    ** 3. 将缓存在buffer中的数据，转化为std::string类型字符asked
    */
    memset(buffer, 0, sizeof(buffer));
    recvfrom(server_socket, buffer, 1024, 0, (struct sockaddr *)&client_addr, &len);
    std::string asked(buffer);

    /*客户端请求断开连接
    ** 1. 检查asked是否为exit
    ** 2. 如果是，则在服务器控制台打印客户端断开连接的消息
    ** 3. 然后用return退出函数，即断开与客户端的连接
    */
```

```
161     if (asked == "exit")
162     {
163         std::cout << "Disconnect: " << inet_ntoa(client_addr.sin_addr) << std::endl;
164         return;
165     }
166 
167     /*
168     ** 不断开连接的情况下，页打印客户端的IP和客户端发送的数据
169     */
170     std::cout << "客户端 - " << inet_ntoa(client_addr.sin_addr) << ": " << asked << std::endl;
171 
172 
173     /*try-catch是异常处理结构，用于处理可能在执行代码时发生的错误
174     ** 1. try模块在正常运行时总是被执行
175     ** 2. catch模块在try块内的代码抛出异常时执行
176     ** 3. 在此处，这个所谓的异常是指，asked包含非数字字符，从而转化为浮点数失败
177     */
178     try
179     {
180         /*
181         ** 1. 将客户端发送的字符串asked转为浮点数，通过F计算后结果存储在answer中
182         ** 2. 打印answer计算结果，显示是来自于哪个客户端的请求
183         ** 3. 将answer计算结果转为字符串
184         ** 4. 将得到的字符串送回客户端
185         */
186         float answer = F(std::stof(asked));
187         std::cout << "服务器 - " << inet_ntoa(client_addr.sin_addr) << ": " << answer << std::endl;
188         std::string response = std::to_string(answer);
189         sendto(server_socket, response.c_str(), response.length(), 0, (struct sockaddr *)&client_addr, len);
190     }
191     catch (const std::exception& e)
192     {
193         /*
194         ** 1. 先设置打印/响应错误输入的消息
195         ** 2. 打印输错误消息，以及客户端的IP等信息
196         ** 3. 通过sendto函数将错误消息返回给客户端
197         */
198         std::string response = "Please check your input!";
199         std::cout << "服务器 - " << inet_ntoa(client_addr.sin_addr) << ": " << response << std::endl;
200         sendto(server_socket, response.c_str(), response.length(), 0, (struct sockaddr *)&client_addr, len);
201     }
202 }
203 }
204 
205 
206 int main()
207 {
208 /*创建套接字
```

```
** 1. 创建一个UDP套接字，AF_INET表示使用IPv4，SOCK_DGRAM指定套接字类型为数据报，即无连接的UDP
** 2. 检查套接字是否创建成功，失败的话就返回失败信息
*/
server_socket = socket(AF_INET, SOCK_DGRAM, 0);
if (server_socket < 0)
{
   std::cerr << "Socket creation failed." << std::endl;
   return -1;
}

/*设置服务器地址
** 1. 设置地址族为 IPv4
** 2. 指定服务器套接字绑定到所有可用的接口上，htonl和INADDR_ANY确保地址格式正确
** 3. 设置服务器端口为 1212，htons确保端口号格式正确
*/
server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
server_addr.sin_port = htons(1212);

/*绑定套接字到地址
** 1. 将之前创建的套接字绑定到指定的地址(IP和端口)
** 2. 如果绑定失败，则打印错误信息并返回-1，绑定失败基本就是因为端口号已经被占用了
** 3. 用buffer来缓存收到的客户端的数据
*/
if (bind(server_socket, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0)
{
   std::cerr << "Bind failed." << std::endl;
   return -1;
}
char buffer[1024];
/*当总是因为端口号被占用而返回失败时的解决方案
** 1. 用sudo netstat -tulnp | grep 1212命令查找占用1212端口的进程PID
** 2. 暴力杀掉这个进程，sudo kill -9 <进程PID>
*/

/*启动一个新线程(对象exitThread)来处理退出命令
** 1. 在新线程中运行wait_for_exit，后者监控输入，以便在特定输入(exit)时关闭服务器
** 2. 将新创建线程从对象中分离，转而在后台运行
*/
std::thread exitThread(wait_for_exit);
exitThread.detach();


/*
** 持续监听和处理来自客户端的数据
*/
while (true)
{
  /*
  ** 1. 始化一个变量len来存储客户端地址结构的大小
  ** 2. 使用recvfrom函数从套接字server_socket接收数据
```

```
  **    接收到的数据被存储在buffer数组中，最多可接收1024字节
  */
  socklen_t len = sizeof(client_addr);
  int received = recvfrom(server_socket, buffer, 1024, 0, (struct sockaddr *)&client_addr, &len);

   /*
   **如果有接收到数据，则做如下处理
   */
   if (received > 0)
   {
       //不论如何都将收到的数据以空字符结尾，以形成有效的字符串
       buffer[received] = '\0';

       /*
       ** 1. 定义it变量，用于检查客户端是否已连接
       ** 2. 如果找到了客户端的地址(即客户端已经连接过)
       **    则将缓存在buffer中的已收到消息添加到客户端的消息列表中
       */
       auto it = Client_Message.find(client_addr.sin_addr.s_addr);
       if (it != Client_Message.end())
       {
           it->second.push_back(std::string(buffer));
       }

       /*
       ** 1. 若客户端是新连接的，则在新客户端的Client_Message中创建一个新消息列表
       ** 2. 为新的客户端创建一个线程，该线程调用CS_thread，负责处理与客户端的进一步通信与交互
       ** 3. 最后将新线程从对象中分离，使之可以独立运行
       */
       else
       {
           Client_Message[client_addr.sin_addr.s_addr] = std::vector<std::string>(1, std::string(buffer));
           std::thread clientThread(CS_thread, client_addr);  // 定义并启动客户端线程
           clientThread.detach();
       }
   }

   /*
   **如果有接收到数据或者数据有错误，则返回错误信息
   */
   else
   {
       std::cerr << "Error in receiving data." << std::endl;
   }
}

return 0;
}
```

3.3. 运行：WSL环境

1️⃣安装C++多线程库

```
sudo apt --fix-broken install
sudo apt-get install libstdc++6
```

2️⃣编译程序

```
g++ -o UDP-Server UDP-Server.cpp -lpthread -static-libstdc++
g++ -o UDP-Client UDP-Client.cpp -lpthread -static-libstdc++
```

报错解决：度过尝试打开服务器时返回`Bind failed.`则大概率是因为端口1212被占用，需要杀死占用端口的进程

```
sudo netstat -tulnp | grep 1212   #找到占用1212端口的进程PID
sudo kill -9 <PID>  #以最高权限杀死这个进程
```

3️⃣运行：在两个终端，先打开服务器再打开客户端，完成输入

```
dann_hiroaki@DESKTOP-QANEDCT:~/socket/UDP$ ./UDP-Client
Connect Succeed!
欢迎访问1212服务器!

用户名: DHY

密码: DHY
用户名或密码错误,请再次输入!
a
用户名: DHY
a
密码: dhy
信息正确!
```

```
dann_hiroaki@DESKTOP-QANEDCT:~/socket/UDP$ ./UDP-Server
Connect Succeed!
Client Address: 127.0.0.1
Error in receiving data.
用户名: DHY
Error in receiving data.
密码: DHY
Proof Defeated!
用户名: DHY
密码: dhy
Proof Success!
```

4. TCP协议的实现

4.1. `TCP-Client.cpp`

代码逻辑和DUP的实现差不多

```cpp
#include <iostream>
#include <string>
#include <thread>
#include <atomic>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>

std::atomic<bool> exit_flag(false);
std::atomic<bool> proof_flag(false);
std::atomic<bool> recv_flag(true);
int client_socket;

void wait_for_exit()
{
 while (!exit_flag.load()) {}
 close(client_socket);
 exit(0);
}

void CS_Server()
{
 char buffer[1024];
 while (true)
 {
     if (exit_flag.load()) return;

     int received = recv(client_socket, buffer, 1024, 0);
     if (received <= 0) continue;

     buffer[received] = '\0'; // 确保字符串以空字符结束
     std::string data(buffer);

     if (proof_flag.load())
     {
         std::cout << "服务器: " << data << std::endl;
         recv_flag.store(true);
     }
     else
     {
         std::cout << data;
         if (data == "信息正确!\r\n")
         {
             proof_flag.store(true);
             std::cout << "客户端: ";
         }
     }
 }
}

void CS_Client()
{
 while (true)
```

```
 {
     std::string data;
     if (proof_flag.load())
     {
         while (!recv_flag.load()) {}
         std::cout << "客户端: ";
         std::getline(std::cin, data);
         recv_flag.store(false);
     }
     else
     {
         std::getline(std::cin, data);
         recv_flag.store(false);
     }

     send(client_socket, data.c_str(), data.length(), 0);
     if (data == "exit")
     {
         exit_flag.store(true);
         return;
     }
 }
}

int main()
{
 client_socket = socket(AF_INET, SOCK_STREAM, 0);
 if (client_socket < 0)
 {
     std::cerr << "Socket creation failed." << std::endl;
     return -1;
 }

 sockaddr_in server_addr;
 server_addr.sin_family = AF_INET;
 server_addr.sin_port = htons(1212);
 inet_pton(AF_INET, "127.0.0.1", &server_addr.sin_addr); // 使用
localhost

 /*
 **
 */
 if (connect(client_socket, (struct sockaddr *)&server_addr,
sizeof(server_addr)) < 0)
 {
     std::cerr << "Connection failed." << std::endl;
     return -1;
 }

 std::cout << "Connect Succeed!" << std::endl;

 std::thread exit_thread(wait_for_exit);
 std::thread server_thread(CS_Server);
 std::thread client_thread(CS_Client);

 exit_thread.detach();
```

```
 server_thread.detach();
 client_thread.join();

 return 0;
}
```

4.2. TCP-Server.cpp

代码逻辑和DUP的实现差不多

```
#include <iostream>
#include <string>
#include <thread>
#include <vector>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>

struct User
{
 std::string username;
 std::string pwd;
};

std::vector<User> Users = {{"DHY", "dhy"}};
int server_socket;

void wait_for_exit()
{
 std::string input;
 while (true)
 {
     std::cin >> input;
     if (input == "exit")
     {
         close(server_socket);
         exit(0);
     }
 }
}

float F(float x)
{
 return x * x + 2 * x + 1;
}

void CS_thread(int client_socket, sockaddr_in addr)
{
 char buffer[1024];
 bool proof_flag = false;
 std::cout << "Connect Succeed!" << std::endl;
 std::cout << "Client Address: " << inet_ntoa(addr.sin_addr) <<
std::endl;
```

```
 send(client_socket, "欢迎访问1212服务器!\r\n", strlen("欢迎访问1212服务器!\r\n"), 0);

 while (!proof_flag)
 {
     // Username
     send(client_socket, "用户名: ", strlen("用户名: "), 0);
     int received = recv(client_socket, buffer, 1024, 0);
     buffer[received] = '\0';
     std::string username(buffer);
     std::cout << "用户名: " << username << std::endl;

     // Password
     send(client_socket, "密码: ", strlen("密码: "), 0);
     received = recv(client_socket, buffer, 1024, 0);
     buffer[received] = '\0';
     std::string pwd(buffer);
     std::cout << "密码: " << pwd << std::endl;

     // Authentication
     for (const auto& user : Users)
     {
         if (user.username == username && user.pwd == pwd)
         {
             proof_flag = true;
             std::cout << "Proof Success!" << std::endl;
             send(client_socket, "信息正确!\r\n", strlen("信息正确!\r\n"), 0);
             break;
         }
     }

     if (!proof_flag)
     {
         std::cout << "Proof Defeated!" << std::endl;
         send(client_socket, "用户名或密码错误,请再次输入!\r\n", strlen("用户名或密码错误,请再次输入!\r\n"), 0);
     }
 }

 // Connected
 while (true)
 {
     int received = recv(client_socket, buffer, 1024, 0);
     if (received <= 0)
     {
         std::cout << "Disconnect: " << inet_ntoa(addr.sin_addr) << std::endl;
         close(client_socket);
         return;
     }

     buffer[received] = '\0';
     std::string asked(buffer);
```

```
96      std::cout << "客户端 - " << inet_ntoa(addr.sin_addr) << ": " << asked << std::endl;
97
98      try
99      {
100         float answered = F(std::stof(asked));
101         std::cout << "服务器 - " << inet_ntoa(addr.sin_addr) << ": " << answered << std::endl;
102         std::string response = std::to_string(answered);
103         send(client_socket, response.c_str(), response.length(), 0);
104     } catch (const std::exception& e)
105     {
106         std::string response = "Please check your input!";
107         std::cout << "服务器 - " << inet_ntoa(addr.sin_addr) << ": " << response << std::endl;
108         send(client_socket, response.c_str(), response.length(), 0);
109     }
110 }
111 }
112
113 int main()
114 {
115  server_socket = socket(AF_INET, SOCK_STREAM, 0);
116  if (server_socket < 0)
117  {
118      std::cerr << "Socket creation failed." << std::endl;
119      return -1;
120  }
121
122  sockaddr_in server_addr;
123  server_addr.sin_family = AF_INET;
124  server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
125  server_addr.sin_port = htons(1212);
126
127  if (bind(server_socket, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0)
128  {
129      std::cerr << "Bind failed." << std::endl;
130      return -1;
131  }
132
133  listen(server_socket, 5);
134  std::thread exitThread(wait_for_exit);
135  exitThread.detach();
136
137  while (true)
138  {
139      sockaddr_in client_addr;
140      socklen_t len = sizeof(client_addr);
141      int client_socket = accept(server_socket, (struct sockaddr *)&client_addr, &len);
142      if (client_socket < 0)
143      {
144          continue;
```

```
        }
        std::thread(CS_thread, client_socket, client_addr).detach();
    }

    return 0;
}
```

4.3. 代码层面TCP/UDP的区别

```
/*套字建立
** 1. TCP使用socket(AF_INET, SOCK_STREAM, 0)创建流式套接字
**     - SOCK_STREAM指定了套接字类型为流式，适用于TCP协议
**     - 流式套接字就像是电话通话，建立稳定的连接，保证数据按顺序、完整地传输
** 2. UDP使用socket(AF_INET, SOCK_DGRAM, 0)创建数据报套接字
**     - SOCK_DGRAM指定了数据报套接字，适用于UDP协议
**     - 这类似于邮件系统，每个数据包独立发送，不保证送达顺序或送达本身
*/

/*连接建立
** 1. TCP连接建立
**     - 客户端使用connect()与服务器建立连接
**     - 服务器端使用bind(), listen()和accept()来接受连接
** 2. UDP连接建立
**     - 客户端和服务器使用bind()将套接字与特定的地址和端口关联
**     - 但不需要listen()和accept()步骤
*/

/*数据传输
** 1. TCP数据传输
**     - 数据传输使用send()和recv()函数，确保数据按顺序、完整地传输
**     - 因此TCP适用于需要高可靠性的应用，如文件传输、网页浏览
** 2. UDP数据传输
**     - 不需要建立连接，使用sendto()和recvfrom()函数直接发送和接收数据
**     - 这些函数需要指定目标服务器的地址
**     - UDP传输快速但不保证可靠性或数据顺序，适用于对速度敏感的应用，如实时视频或音频流
*/

/*数据接收处理
** 1. TCP:
        - 服务器端通常在一个单独的线程或进程中处理每个连接
 ** 2. UDP:
        -服务器端通常在一个单独的线程中处理所有来自不同客户端的数据，因为UDP不维护连接状态
 */

 /*连接终止
 ** 1. TCP连接管理
 **     - 在通信结束时，使用close()函数关闭连接，确保资源得到合理释放
 **     - TCP保证数据的可靠性和顺序性，适合那些需要确保数据完整性的应用
 ** 2. UDP连接管理
 **     - UDP是无连接的，发送和接收数据时不需要建立和维护一个持续的连接
 **     - 每个数据包独立发送，不保证顺序或可靠性
 */
```

3.3. 运行：WSL环境

1️⃣安装C++多线程库

```
sudo apt --fix-broken install
sudo apt-get install libstdc++6
```

2️⃣编译程序

```
g++ -o TCP-Server TCP-Server.cpp -lpthread -static-libstdc++
g++ -o TCP-Client TCP-Client.cpp -lpthread -static-libstdc++
```

报错解决：度过尝试打开服务器时返回`Bind failed.`则大概率是因为端口1212被占用，需要杀死占用端口的进程

```
sudo netstat -tulnp | grep 1212   #找到占用1212端口的进程PID
sudo kill -9 <PID>  #以最高权限杀死这个进程
```

3️⃣运行：在两个终端，先打开服务器再打开客户端，完成输入

```
dann_hiroaki@DESKTOP-QANEDCT:~/socket/TCP$ ./TCP-Client
Connect Succeed!
欢迎访问1212服务器!
用户名: DHY

密码: as
用户名或密码错误,请再次输入!

用户名: DHY

密码: dhy
信息正确!

^C
dann_hiroaki@DESKTOP-QANEDCT:~/socket/TCP$
```

```
dann_hiroaki@DESKTOP-QANEDCT:~/socket/TCP$ ./TCP-Server
Connect Succeed!
Client Address: 127.0.0.1
用户名: DHY
密码: as
Proof Defeated!
用户名: DHY
密码: dhy
Proof Success!
Disconnect: 127.0.0.1
^C
dann_hiroaki@DESKTOP-QANEDCT:~/socket/TCP$
```