

**计算机网络**

**课程实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 实验名称 | 可靠数据传输协议-停等协议/ GBN协议的设计与实现 | | | | | |
| 姓名 | 何思远 | | 院系 | 计算学部 | | |
| 班级 | 2303103 | | 学号 | 2023212224 | | |
| 任课教师 | 聂兰顺 | | 指导教师 | 聂兰顺 | | |
| 实验地点 | 格物楼213 | | 实验时间 | 2025.10.21 | | |
| 实验课表现 | 出勤、表现得分(10) |  | 实验报告  得分(40) |  | 实验总分 |  |
| 操作结果得分(50) |  |
| 教师评语 | | | | | | |
|  | | | | | | |

****

|  |
| --- |
| 实验目的： |
| 理解可靠数据传输的基本原理；掌握停等协议的工作原理；掌握基 于UDP设计并实现一个停等协议的过程与技术。理解滑动窗口协议的基本原理；掌握 GBN 的工作原理；掌握基于 UDP设计并实现一个GBN协议的过程与技术。 |
| 实验内容： |
| 1) 基于UDP设计一个简单的停等协议，实现单向可靠数据传输（服 务器到客户的数据传输）。  2) 模拟引入数据包的丢失，验证所设计协议的有效性。  3) 改进所设计的停等协议，支持双向数据传输；  4）基于所设计的停等协议，实现一个C/S结构的文件传输应用。  1) 基于UDP 设计一个简单的GBN 协议，实现单向可靠数据传输（服  务器到客户的数据传输）。  2) 模拟引入数据包的丢失，验证所设计协议的有效性。  3) 改进所设计的 GBN 协议，支持双向数据传输  4）将所设计的 GBN 协议改进为 SR 协议。 |
| 实验过程： |
| **停等协议实验**  数据分组格式：DataFrame    seq: 序列号,使用0和1两个值交替  data: 传输的数据内容  flag: 标志位,0表示普通数据帧,1表示最后一帧  确认分组格式：ACK帧结构    ack: 确认序列号,值为0或1  协议两端程序流程图      协议典型交互过程  正常情况：    ACK丢失情况：      数据分组丢失验证模拟方法：  实际上收到了信息但假装没有收到，即不发送相应的 ack，并等待接下来的操作。  Server 端代码：  #include <iostream>  #include <winsock2.h>  #include <ws2tcpip.h>  #include <string>  #include <cstring>  #include <ctime>  #pragma comment(lib, "ws2\_32.lib")  using namespace std;  // 服务器配置  #define SERVER\_PORT 12340  #define BUFFER\_SIZE 1024  #define TIMEOUT\_MS 2000 // 超时时间2秒  // 数据帧结构  struct DataFrame {  unsigned char seq; // 序列号 (0或1)  char data[BUFFER\_SIZE]; // 数据  unsigned char flag; // 标志位: 0=数据帧, 1=结束帧  };  // ACK帧结构  struct AckFrame {  unsigned char ack; // ACK序列号  };  // 全局变量  float ackLossRate = 0.0f; // ACK丢包率  // 模拟ACK丢包  bool simulateAckLoss() {  if (ackLossRate <= 0.0f) return false;  float random = (float)rand() / RAND\_MAX;  return random < ackLossRate;  }  // 获取当前时间字符串  string getCurrentTime() {  time\_t now = time(0);  char\* dt = ctime(&now);  string timeStr(dt);  timeStr.pop\_back(); // 移除换行符  return timeStr;  }  /\*\*  \* @brief 发送数据帧并等待ACK  \* @param sock 套接字  \* @param clientAddr 客户端地址  \* @param frame 数据帧  \* @param expectedAck 期望的ACK序列号  \* @param isLastFrame 是否是最后一个数据帧  \*/  bool sendDataFrame(SOCKET sock, sockaddr\_in& clientAddr, DataFrame& frame, int expectedAck, bool isLastFrame = false) {  char buffer[sizeof(DataFrame)]; // 数据帧缓冲区  memcpy(buffer, &frame, sizeof(DataFrame)); // 复制数据帧到缓冲区  int attempts = 0; // 重传次数  const int MAX\_ATTEMPTS = 5;  while (attempts < MAX\_ATTEMPTS) {  // 发送数据帧  int sendResult = sendto(sock, buffer, sizeof(DataFrame), 0,  (sockaddr\*)&clientAddr, sizeof(clientAddr));  if (sendResult == SOCKET\_ERROR) {  cout << "[错误] 发送数据失败" << endl;  return false;  }  cout << "[发送] Seq=" << (int)frame.seq << ", 数据大小=" << strlen(frame.data)  << " 字节, 尝试次数=" << (attempts + 1) << endl;  // 等待ACK  sockaddr\_in fromAddr;  int fromLen = sizeof(fromAddr);  AckFrame ackFrame;  fd\_set readSet; // 读集合,创建一个“监视列表”  FD\_ZERO(&readSet); // 初始化读集合  FD\_SET(sock, &readSet); // 将套接字加入读集合  timeval timeout; // 超时结构体  timeout.tv\_sec = TIMEOUT\_MS / 1000; // 秒  timeout.tv\_usec = (TIMEOUT\_MS % 1000) \* 1000; // 微秒  // select 在不阻塞程序的情况下，监视一个或多个 Socket，看它们是否准备好进行某种操作（比如读或写），并设置一个最长等待时间（超时）  int selectResult = select(0, &readSet, NULL, NULL, &timeout);  if (selectResult > 0) {  // 有数据可读，接收ACK  int recvLen = recvfrom(sock, (char\*)&ackFrame, sizeof(AckFrame), 0,  (sockaddr\*)&fromAddr, &fromLen);  if (recvLen > 0) {  cout << "[接收] ACK=" << (int)ackFrame.ack << endl;  // 模拟ACK丢失(服务器收到ACK但假装没收到)  // 注意: 最后一个数据包不模拟ACK丢失，因为客户端已经结束接收  if (!isLastFrame && simulateAckLoss()) {  cout << "[模拟] ACK丢失! 假装未收到ACK" << endl;  attempts++; // 计入重传次数  continue; // 继续等待下一个ACK或超时重传  }  if (ackFrame.ack == expectedAck) {  cout << "[成功] 收到正确ACK" << endl;  return true;  } else {  cout << "[警告] ACK序列号不匹配, 期望=" << expectedAck  << ", 实际=" << (int)ackFrame.ack << endl;  }  }  } else if (selectResult == 0) {  // 超时  cout << "[超时] 未收到ACK, 准备重传..." << endl;  } else {  cout << "[错误] select调用失败" << endl;  }  attempts++;  }  cout << "[失败] 达到最大重传次数" << endl;  return false;  }  int main() {  srand((unsigned int)time(NULL)); // 初始化随机数种子  // 初始化Winsock  WSADATA wsaData;  int result = WSAStartup(MAKEWORD(2, 2), &wsaData);  if (result != 0) {  cout << "WSAStartup失败: " << result << endl;  return 1;  }  // 创建UDP socket  SOCKET serverSocket = socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP);  if (serverSocket == INVALID\_SOCKET) {  cout << "创建socket失败: " << WSAGetLastError() << endl;  WSACleanup();  return 1;  }  // 绑定地址  sockaddr\_in serverAddr;  serverAddr.sin\_family = AF\_INET;  serverAddr.sin\_addr.s\_addr = INADDR\_ANY;  serverAddr.sin\_port = htons(SERVER\_PORT);  if (bind(serverSocket, (sockaddr\*)&serverAddr, sizeof(serverAddr)) == SOCKET\_ERROR) {  cout << "绑定失败: " << WSAGetLastError() << endl;  closesocket(serverSocket);  WSACleanup();  return 1;  }  cout << "======================================" << endl;  cout << "停等协议服务器已启动" << endl;  cout << "监听端口: " << SERVER\_PORT << endl;  cout << "======================================" << endl;  // 设置ACK丢包率  cout << "\n请输入ACK丢包率 (0.0-1.0, 默认0.2): ";  string input;  getline(cin, input);  if (!input.empty()) {  try {  ackLossRate = stof(input);  if (ackLossRate < 0.0f) ackLossRate = 0.0f;  if (ackLossRate > 1.0f) ackLossRate = 1.0f;  } catch (...) {  cout << "[警告] 无效的丢包率, 使用默认值0.2" << endl;  ackLossRate = 0.2f;  }  } else {  ackLossRate = 0.2f;  }  cout << "\n[配置] ACK模拟丢包率: " << (ackLossRate \* 100) << "%" << endl;  cout << "======================================" << endl;  // 主循环  while (true) {  char recvBuffer[BUFFER\_SIZE];  sockaddr\_in clientAddr;  int clientAddrLen = sizeof(clientAddr);  cout << "\n等待客户端请求..." << endl;  // 接收客户端请求  int recvLen = recvfrom(serverSocket, recvBuffer, BUFFER\_SIZE, 0,  (sockaddr\*)&clientAddr, &clientAddrLen);  if (recvLen > 0) {  recvBuffer[recvLen] = '\0';  string command(recvBuffer); // 将接收到的数据(C 字符数组)转换为字符串(C++ string)  char clientIP[INET\_ADDRSTRLEN]; // 存储客户端IP地址的字符串  inet\_ntop(AF\_INET, &(clientAddr.sin\_addr), clientIP, INET\_ADDRSTRLEN); // 将二进制 IP 地址(clientAddr.sin\_addr)转换为字符串形式存入 clientIP  // ntohs 将网络字节序的端口号转换为主机字节序  cout << "[请求] 来自 " << clientIP << ":" << ntohs(clientAddr.sin\_port)  << " - " << command << endl;  // 处理命令  if (command == "-time") {  // 返回当前时间  string timeStr = getCurrentTime();  sendto(serverSocket, timeStr.c\_str(), timeStr.length(), 0,  (sockaddr\*)&clientAddr, clientAddrLen);  cout << "[响应] 发送时间: " << timeStr << endl;  }  else if (command == "-quit") {  // 客户端退出  string goodbye = "Good bye!";  sendto(serverSocket, goodbye.c\_str(), goodbye.length(), 0,  (sockaddr\*)&clientAddr, clientAddrLen);  cout << "[响应] 客户端断开连接" << endl;  }  else if (command == "-test") {  // 测试停等协议  cout << "\n[开始] 停等协议数据传输测试" << endl;  cout << "======================================" << endl;  // 准备测试数据  const char\* testData[] = {  "这是第一个数据包",  "这是第二个数据包",  "这是第三个数据包",  "停等协议测试数据",  "最后一个数据包"  };  int dataCount = 5;  // 使用停等协议发送数据  unsigned char currentSeq = 0; // 当前序列号 (0或1)  bool success = true;  for (int i = 0; i < dataCount; i++) {  DataFrame frame;  frame.seq = currentSeq;  strcpy\_s(frame.data, testData[i]);  frame.flag = (i == dataCount - 1) ? 1 : 0; // 最后一帧标记为结束  bool isLastFrame = (i == dataCount - 1); // 是否是最后一帧  cout << "\n--- 发送数据包 " << (i + 1) << "/" << dataCount << " ---" << endl;  if (!sendDataFrame(serverSocket, clientAddr, frame, currentSeq, isLastFrame)) {  cout << "[失败] 数据包发送失败" << endl;  success = false;  break;  }  // 切换序列号 (0->1, 1->0)  currentSeq = 1 - currentSeq;  }  cout << "\n======================================" << endl;  if (success) {  cout << "[完成] 所有数据包已成功发送" << endl;  } else {  cout << "[失败] 数据传输失败" << endl;  }  cout << "======================================" << endl;  }  else {  // 回显数据  sendto(serverSocket, recvBuffer, recvLen, 0,  (sockaddr\*)&clientAddr, clientAddrLen);  cout << "[响应] 回显数据: " << command << endl;  }  }  }  // 清理  closesocket(serverSocket);  WSACleanup();  return 0;  }  Client 端代码：  #include <iostream>  #include <winsock2.h>  #include <ws2tcpip.h>  #include <string>  #include <cstring>  #include <cstdlib>  #include <ctime>  #pragma comment(lib, "ws2\_32.lib")  using namespace std;  // 服务器配置  #define SERVER\_PORT 12340  #define SERVER\_IP "127.0.0.1"  #define BUFFER\_SIZE 1024  // 数据帧结构  struct DataFrame {  unsigned char seq; // 序列号 (0或1)  char data[BUFFER\_SIZE]; // 数据  unsigned char flag; // 标志位: 0=数据帧, 1=结束帧  };  // ACK帧结构  struct AckFrame {  unsigned char ack; // ACK序列号  };  // 全局变量  float lossRate = 0.0f; // 数据包丢包率  float ackLossRate = 0.0f; // ACK丢包率  // 模拟数据包丢包  bool simulateLoss() {  if (lossRate <= 0.0f) return false;  float random = (float)rand() / RAND\_MAX;  return random < lossRate;  }  // 模拟ACK丢包  bool simulateAckLoss() {  if (ackLossRate <= 0.0f) return false;  float random = (float)rand() / RAND\_MAX;  return random < ackLossRate;  }  int main() {  srand((unsigned int)time(NULL));  // 初始化Winsock  WSADATA wsaData;  int result = WSAStartup(MAKEWORD(2, 2), &wsaData);  if (result != 0) {  cout << "WSAStartup失败: " << result << endl;  return 1;  }  // 创建UDP socket  SOCKET clientSocket = socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP);  if (clientSocket == INVALID\_SOCKET) {  cout << "创建socket失败: " << WSAGetLastError() << endl;  WSACleanup();  return 1;  }  // 设置服务器地址  sockaddr\_in serverAddr;  serverAddr.sin\_family = AF\_INET;  serverAddr.sin\_port = htons(SERVER\_PORT);  inet\_pton(AF\_INET, SERVER\_IP, &serverAddr.sin\_addr);  cout << "======================================" << endl;  cout << "停等协议客户端" << endl;  cout << "服务器地址: " << SERVER\_IP << ":" << SERVER\_PORT << endl;  cout << "======================================" << endl;  cout << "\n可用命令:" << endl;  cout << " -time 获取服务器时间" << endl;  cout << " -test [数据丢包率] [ACK丢包率] 测试停等协议" << endl;  cout << " 丢包率范围0.0-1.0" << endl;  cout << " 默认: 数据丢包率=0.2, ACK丢包率=0.1" << endl;  cout << " -quit 退出程序" << endl;  cout << " 其他文本 回显测试" << endl;  cout << "======================================\n" << endl;  // 主循环  while (true) {  cout << "\n请输入命令: ";  string input;  getline(cin, input);  if (input.empty()) continue;  // 解析命令  if (input == "-time") {  // 请求时间  sendto(clientSocket, input.c\_str(), input.length(), 0,  (sockaddr\*)&serverAddr, sizeof(serverAddr));  char recvBuffer[BUFFER\_SIZE];  sockaddr\_in fromAddr; // 服务器地址  int fromLen = sizeof(fromAddr); // 地址长度  // 消息接收，recvBuffer存放接收到的数据, fromAddr存放发送方地址，fromLen存放地址长度, 返回接收数据的长度  int recvLen = recvfrom(clientSocket, recvBuffer, BUFFER\_SIZE, 0,  (sockaddr\*)&fromAddr, &fromLen);  if (recvLen > 0) {  recvBuffer[recvLen] = '\0';  cout << "[服务器时间] " << recvBuffer << endl;  }  }  else if (input == "-quit") {  // 退出  sendto(clientSocket, input.c\_str(), input.length(), 0,  (sockaddr\*)&serverAddr, sizeof(serverAddr));  char recvBuffer[BUFFER\_SIZE];  sockaddr\_in fromAddr;  int fromLen = sizeof(fromAddr);  int recvLen = recvfrom(clientSocket, recvBuffer, BUFFER\_SIZE, 0,  (sockaddr\*)&fromAddr, &fromLen);  if (recvLen > 0) {  recvBuffer[recvLen] = '\0';  cout << "[服务器] " << recvBuffer << endl;  }  break;  }  else if (input.substr(0, 5) == "-test") {  // 测试停等协议  lossRate = 0.2f; // 默认数据丢包率  ackLossRate = 0.1f; // 默认ACK丢包率  // 解析参数  if (input.length() > 6) {  size\_t firstSpace = input.find(' ', 6);  size\_t secondSpace = input.find(' ', firstSpace + 1);  // 解析第一个参数(数据丢包率)  try {  string firstParam = (secondSpace != string::npos)  ? input.substr(6, secondSpace - 6)  : input.substr(6);  lossRate = stof(firstParam);  if (lossRate < 0.0f) lossRate = 0.0f;  if (lossRate > 1.0f) lossRate = 1.0f;  } catch (...) {  cout << "[警告] 无效的数据丢包率, 使用默认值0.2" << endl;  lossRate = 0.2f;  }  // 解析第二个参数(ACK丢包率)  if (secondSpace != string::npos && secondSpace + 1 < input.length()) {  try {  string secondParam = input.substr(secondSpace + 1);  ackLossRate = stof(secondParam);  if (ackLossRate < 0.0f) ackLossRate = 0.0f;  if (ackLossRate > 1.0f) ackLossRate = 1.0f;  } catch (...) {  cout << "[警告] 无效的ACK丢包率, 使用默认值0.1" << endl;  ackLossRate = 0.1f;  }  }  }  cout << "\n[开始] 停等协议测试" << endl;  cout << "数据包丢包率: " << (lossRate \* 100) << "%" << endl;  cout << "ACK丢包率: " << (ackLossRate \* 100) << "%" << endl;  cout << "======================================" << endl;  // 发送测试命令  string testCmd = "-test";  sendto(clientSocket, testCmd.c\_str(), testCmd.length(), 0,  (sockaddr\*)&serverAddr, sizeof(serverAddr));  // 接收数据  unsigned char expectedSeq = 0; // 期望的序列号  int receivedCount = 0; // 已接收数据包计数  int duplicateCount = 0; // 重复数据包计数  while (true) {  DataFrame frame;  sockaddr\_in fromAddr;  int fromLen = sizeof(fromAddr);  // 设置接收超时  DWORD timeout = 10000; // 10秒  setsockopt(clientSocket, SOL\_SOCKET, SO\_RCVTIMEO,  (char\*)&timeout, sizeof(timeout)); // SO\_RCVTIMEO 设置接收超时  int recvLen = recvfrom(clientSocket, (char\*)&frame, sizeof(DataFrame), 0,  (sockaddr\*)&fromAddr, &fromLen);  if (recvLen > 0) {  cout << "\n[接收] Seq=" << (int)frame.seq  << ", 数据=\"" << frame.data << "\"" << endl;  // 模拟丢包  if (simulateLoss()) {  cout << "[模拟] 数据包丢失! 不发送ACK" << endl;  continue;  }  // 检查序列号  if (frame.seq == expectedSeq) {  cout << "[正确] 序列号匹配, 准备发送ACK=" << (int)expectedSeq << endl;  // 模拟ACK丢失  if (simulateAckLoss()) {  cout << "[模拟] ACK丢失! 不发送ACK" << endl;  continue;  }  // 发送ACK  AckFrame ackFrame;  ackFrame.ack = expectedSeq;  sendto(clientSocket, (char\*)&ackFrame, sizeof(AckFrame), 0,  (sockaddr\*)&serverAddr, sizeof(serverAddr));  cout << "[发送] ACK=" << (int)expectedSeq << endl;  receivedCount++;  // 更新期望的序列号  expectedSeq = 1 - expectedSeq;  // 检查是否为最后一帧  if (frame.flag == 1) {  cout << "\n[完成] 收到结束标志, 传输完成!" << endl;  cout << "======================================" << endl;  cout << "统计信息:" << endl;  cout << " 成功接收数据包: " << receivedCount << " 个" << endl;  cout << " 重复数据包: " << duplicateCount << " 个" << endl;  cout << "======================================" << endl;  break;  }  } else {  cout << "[重复] 序列号不匹配! 期望=" << (int)expectedSeq  << ", 实际=" << (int)frame.seq << endl;  cout << "[重复] 这是重复的数据包, 丢弃数据" << endl;  duplicateCount++;  // 模拟ACK丢失  if (simulateAckLoss()) {  cout << "[模拟] ACK丢失! 不重发ACK" << endl;  continue;  }  cout << "[重发] 重发上一个ACK=" << (int)(1 - expectedSeq) << endl;  // 重发上一个ACK  AckFrame ackFrame;  ackFrame.ack = 1 - expectedSeq;  sendto(clientSocket, (char\*)&ackFrame, sizeof(AckFrame), 0,  (sockaddr\*)&serverAddr, sizeof(serverAddr));  }  } else if (recvLen == SOCKET\_ERROR) {  int error = WSAGetLastError();  if (error == WSAETIMEDOUT) {  cout << "\n[超时] 未收到更多数据, 测试结束" << endl;  cout << "======================================" << endl;  cout << "统计信息:" << endl;  cout << " 成功接收数据包: " << receivedCount << " 个" << endl;  cout << " 重复数据包: " << duplicateCount << " 个" << endl;  cout << "======================================" << endl;  break;  } else {  cout << "[错误] 接收失败: " << error << endl;  break;  }  }  }  cout << "======================================" << endl;  }  else {  // 回显测试  sendto(clientSocket, input.c\_str(), input.length(), 0,  (sockaddr\*)&serverAddr, sizeof(serverAddr));  char recvBuffer[BUFFER\_SIZE];  sockaddr\_in fromAddr;  int fromLen = sizeof(fromAddr);  int recvLen = recvfrom(clientSocket, recvBuffer, BUFFER\_SIZE, 0,  (sockaddr\*)&fromAddr, &fromLen);  if (recvLen > 0) {  recvBuffer[recvLen] = '\0';  cout << "[回显] " << recvBuffer << endl;  }  }  }  // 清理  closesocket(clientSocket);  WSACleanup();  cout << "\n客户端已关闭" << endl;  return 0;  }  选做内容1：双向停等协议 改进要点与单向协议的区别  | 特性 | 单向协议 | 双向协议 | | --- | --- | --- | | 数据传输方向 | 服务器→客户端 | 双向 | | 序列号管理 | 单一序列号 | 发送和接收各维护序列号 | | 帧类型 | 数据帧、ACK帧 | 数据帧、ACK帧(统一格式) | | 复杂度 | 简单 | 中等 |  关键改进  1. **双序列号机制**  * unsigned char sendSeq = 0; // 发送序列号 unsigned char recvSeq = 0; // 接收序列号   + 发送和接收独立维护序列号   + 避免冲突和混淆  1. **统一帧格式**  * struct DataFrame {  unsigned char seq; // 序列号  char data[1024]; // 数据  unsigned char flag; // 0=数据, 1=结束, 2=ACK };   + 使用flag字段区分帧类型   + 数据帧和ACK帧使用相同结构  1. **双向传输能力**    * 服务器可以向客户端发送数据    * 客户端可以向服务器发送数据    * 支持同时进行双向通信   选做内容2: 文件传输应用 协议设计数据帧格式 struct DataFrame {  unsigned char seq; // 序列号 (0或1)  unsigned int fileSize; // 文件总大小(仅首帧)  unsigned int offset; // 数据在文件中的偏移  unsigned short dataLen; // 当前帧数据长度  char data[1020]; // 数据内容  unsigned char flag; // 帧类型标志 }; 帧类型  * **flag = 0**: 普通数据帧 * **flag = 1**: 最后一个数据帧 * **flag = 2**: ACK确认帧 * **flag = 3**: 文件列表帧(保留) * **flag = 4**: 文件信息帧(传输开始前)  传输流程  1. **建立连接**    * 客户端发送文件信息帧(文件名、大小)    * 服务器发送ACK确认 2. **数据传输**    * 客户端按DATA\_SIZE(1020字节)分割文件    * 每个数据帧包含：序列号、偏移、长度、数据    * 使用停等协议逐帧传输 3. **可靠传输**    * 每发送一帧等待ACK    * 超时未收到ACK则重传(最多5次)    * 序列号0和1交替使用 4. **传输完成**    * 最后一帧设置flag=1    * 服务器收到后返回ACK    * 显示传输统计信息   **GBN 协议实验** 协议数据分组格式数据帧（DataFrame）格式  | 字段名 | 类型 | 大小 | 说明 | | --- | --- | --- | --- | | seq | unsigned char | 1 字节 | 序列号（0-15） | | data | char[] | 1024 字节 | 数据内容 | | flag | unsigned char | 1 字节 | 标志位（0=普通帧，1=结束帧） |   **总大小：** 1026 字节  struct DataFrame {  unsigned char seq; // 序列号  char data[1024]; // 数据  unsigned char flag; // 标志位: 0=普通帧, 1=结束帧 }; 确认帧（AckFrame）格式  | 字段名 | 类型 | 大小 | 说明 | | --- | --- | --- | --- | | ack | unsigned char | 1 字节 | ACK序列号（累积确认） | | flag | unsigned char | 1 字节 | 预留标志位 |   **总大小：** 2 字节  struct AckFrame {  unsigned char ack; // ACK序列号  unsigned char flag; // 预留标志位 };  服务器端流程图    客户端流程图    GBN 协议典型交互过程    服务端代码 server.cpp  #include <iostream>  #include <winsock2.h>  #include <ws2tcpip.h>  #include <string>  #include <cstring>  #include <ctime>  #include <vector>  #include <fstream>  #pragma comment(lib, "ws2\_32.lib")  using namespace std;  // 服务器配置  #define SERVER\_PORT 12340  #define BUFFER\_LENGTH 1026  #define SEND\_WIND\_SIZE 15 // 发送窗口大小(最大为2^n-1=15)  #define SEQ\_SIZE 16 // 序列号个数(n=4, 2^4=16, 序列号范围0-15)  #define TIMEOUT\_THRESHOLD 50 // 超时阈值  #define MAX\_RETRIES 10 // 最大重传次数  // 数据帧结构  struct DataFrame {  unsigned char seq; // 序列号  char data[1024]; // 数据  unsigned char flag; // 标志位: 0=普通帧, 1=结束帧  };  // ACK帧结构  struct AckFrame {  unsigned char ack; // ACK序列号  unsigned char flag; // 预留标志位  };  // 全局变量  BOOL ack[SEQ\_SIZE]; // 收到ACK情况  int curSeq; // 当前数据包的seq  int curAck; // 当前等待确认的ack  int totalSeq; // 总的已发送数据包数  int totalPacket; // 需要发送的包总数  vector<DataFrame> packets; // 数据包缓冲区  // 获取当前时间  void getCurTime(char\* ptime) {  time\_t now = time(0);  strcpy(ptime, ctime(&now));  ptime[strlen(ptime) - 1] = '\0'; // 移除换行符  }  // 判断当前序列号是否可用  bool seqIsAvailable() {  int step = (curSeq + SEQ\_SIZE - curAck) % SEQ\_SIZE; // 计算窗口内已发送但未确认的包数  return step < SEND\_WIND\_SIZE;  }  // 打印发送窗口状态  void printWindow() {  cout << "[窗口状态] curAck=" << curAck << ", curSeq=" << curSeq  << ", 窗口大小=" << SEND\_WIND\_SIZE << endl;  cout << " [";  for (int i = 0; i < SEQ\_SIZE; i++) {  if (i == curAck) cout << "|";  if (i >= curAck && i < (curAck + SEND\_WIND\_SIZE) % SEQ\_SIZE) {  cout << (ack[i] ? "√" : "×");  } else {  cout << " ";  }  if (i == (curAck + SEND\_WIND\_SIZE - 1) % SEQ\_SIZE) cout << "|";  }  cout << "]" << endl;  }  // 超时处理:重传窗口内所有未确认的包  void timeoutHandler(SOCKET sockServer, sockaddr\_in& clientAddr) {  cout << "\n[超时] 检测到超时,重传窗口内所有数据包..." << endl;  int start = curAck;  int end = curSeq;  for (int i = start; i != end; i = (i + 1) % SEQ\_SIZE) {  if (i < packets.size()) {  sendto(sockServer, (char\*)&packets[i], sizeof(DataFrame), 0,  (sockaddr\*)&clientAddr, sizeof(clientAddr));  cout << "[重传] Seq=" << i << ", 数据=\"" << packets[i].data << "\"" << endl;  }  }  }  // 处理收到的ACK  void ackHandler(unsigned char c) {  cout << "[ACK处理] 收到ACK=" << (int)c << endl;  // 计算ACK序列号与curAck的关系（考虑循环）  int ackDist = (c + SEQ\_SIZE - curAck) % SEQ\_SIZE;  int windowSize = (curSeq + SEQ\_SIZE - curAck) % SEQ\_SIZE;  // 判断ACK是否在有效范围内（窗口内的已发送包）  if (ackDist < windowSize) {  // 有效ACK，累积确认到c  int i = curAck;  while (i != (c + 1) % SEQ\_SIZE) {  ack[i] = TRUE;  i = (i + 1) % SEQ\_SIZE;  }  // 移动窗口：滑动到第一个未确认的位置  while (ack[curAck] && curAck != curSeq) {  ack[curAck] = FALSE;  curAck = (curAck + 1) % SEQ\_SIZE;  }  cout << "[窗口滑动] 新的curAck=" << curAck << endl;  } else {  // 重复ACK或过期ACK，忽略  cout << "[忽略] 重复或无效的ACK" << endl;  }  printWindow();  }  int main() {  // 初始化Winsock  WSADATA wsaData;  int err = WSAStartup(MAKEWORD(2, 2), &wsaData);  if (err != 0) {  cout << "WSAStartup失败: " << err << endl;  return 1;  }  // 创建socket  SOCKET sockServer = socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP);  if (sockServer == INVALID\_SOCKET) {  cout << "创建socket失败" << endl;  WSACleanup();  return 1;  }  // 设置非阻塞模式  u\_long iMode = 1;  ioctlsocket(sockServer, FIONBIO, &iMode); // File I/O Non-Blocking I/O  // 绑定地址  sockaddr\_in addrServer;  addrServer.sin\_family = AF\_INET;  addrServer.sin\_addr.s\_addr = INADDR\_ANY;  addrServer.sin\_port = htons(SERVER\_PORT);  if (bind(sockServer, (sockaddr\*)&addrServer, sizeof(addrServer)) == SOCKET\_ERROR) {  cout << "绑定失败: " << WSAGetLastError() << endl;  closesocket(sockServer);  WSACleanup();  return 1;  }  cout << "======================================" << endl;  cout << "GBN协议服务器已启动" << endl;  cout << "监听端口: " << SERVER\_PORT << endl;  cout << "窗口大小: " << SEND\_WIND\_SIZE << endl;  cout << "序列号范围: 0-" << (SEQ\_SIZE - 1) << endl;  cout << "======================================" << endl;  sockaddr\_in addrClient;  int addrClientLen = sizeof(addrClient);  char recvBuffer[BUFFER\_LENGTH];  // 主循环  while (true) {  int recvLen = recvfrom(sockServer, recvBuffer, BUFFER\_LENGTH, 0,  (sockaddr\*)&addrClient, &addrClientLen);  if (recvLen > 0) {  recvBuffer[recvLen] = '\0';  string command(recvBuffer);  char clientIP[INET\_ADDRSTRLEN];  inet\_ntop(AF\_INET, &(addrClient.sin\_addr), clientIP, INET\_ADDRSTRLEN);  cout << "\n[接收] 来自 " << clientIP << ":" << ntohs(addrClient.sin\_port)  << " - " << command << endl;  // 处理命令  if (command == "-time") {  char timeStr[100];  getCurTime(timeStr);  sendto(sockServer, timeStr, strlen(timeStr), 0,  (sockaddr\*)&addrClient, addrClientLen);  cout << "[响应] 发送时间: " << timeStr << endl;  }  else if (command == "-quit") {  string goodbye = "Good bye!";  sendto(sockServer, goodbye.c\_str(), goodbye.length(), 0,  (sockaddr\*)&addrClient, addrClientLen);  cout << "[响应] 客户端断开" << endl;  }  else if (command == "-testgbn") {  cout << "\n[开始] GBN协议测试" << endl;  cout << "======================================" << endl;  // 初始化GBN参数  for (int i = 0; i < SEQ\_SIZE; i++) {  ack[i] = FALSE;  }  curSeq = 0;  curAck = 0;  totalSeq = 0;  packets.clear();  // 准备测试数据  const char\* testData[] = {  "数据包1:GBN协议测试开始",  "数据包2:Go-Back-N测试",  "数据包3:滑动窗口协议",  "数据包4:累积确认机制",  "数据包5:超时重传测试",  "数据包6:窗口大小为10",  "数据包7:序列号0-19",  "数据包8:可靠数据传输",  "数据包9:网络协议实验",  "数据包10:计算机网络",  "数据包11:传输层协议",  "数据包12:UDP封装",  "数据包13:丢包模拟",  "数据包14:ACK确认",  "数据包15:最后的测试数据"  };  totalPacket = 15;  // 准备所有数据包  for (int i = 0; i < totalPacket; i++) {  DataFrame frame;  frame.seq = i % SEQ\_SIZE; // 配置序列号  strcpy(frame.data, testData[i]);  frame.flag = (i == totalPacket - 1) ? 1 : 0;  packets.push\_back(frame); // 存入缓冲区  }  cout << "准备发送 " << totalPacket << " 个数据包" << endl;  printWindow();  // GBN发送循环  int timer = 0;  int retryCount = 0; // 连续超时重传计数  bool transferComplete = false;  while (!transferComplete) {  // 在窗口允许的范围内发送数据  // curSeq < totalPacket 防止在 totalPacket 较小时访问越界  while (seqIsAvailable() && curSeq < totalPacket) {  sendto(sockServer, (char\*)&packets[curSeq], sizeof(DataFrame), 0,  (sockaddr\*)&addrClient, addrClientLen);  cout << "[发送] Seq=" << curSeq  << ", 数据=\"" << packets[curSeq].data << "\"" << endl;  curSeq = (curSeq + 1) % SEQ\_SIZE; // curSeq 更新为下一个待发送的序列号  totalSeq++; // 统计已发送的数据包数    // 不要在发送时重置计时器  // 计时器只在收到ACK或超时重传后重置  }  // 尝试接收ACK  AckFrame ackFrame;  sockaddr\_in fromAddr;  int fromLen = sizeof(fromAddr);  int ackLen = recvfrom(sockServer, (char\*)&ackFrame, sizeof(AckFrame), 0,  (sockaddr\*)&fromAddr, &fromLen);  if (ackLen > 0) {  // 收到ACK,处理并重置计时器  ackHandler(ackFrame.ack);  timer = 0;  retryCount = 0; // 收到ACK，重置重传计数  // 检查是否所有数据都已确认  if (curAck == totalPacket % SEQ\_SIZE && totalSeq == totalPacket) {  transferComplete = true;  }  } else {  // 没有收到ACK,计时器递增  timer++;  if (timer >= TIMEOUT\_THRESHOLD) {  retryCount++;  cout << "[超时] 第 " << retryCount << " 次重传" << endl;    // 检查是否超过最大重传次数  if (retryCount >= MAX\_RETRIES) {  cout << "\n[警告] 达到最大重传次数，可能传输已完成但ACK丢失" << endl;  cout << "[判断] 已发送所有数据包 (totalSeq=" << totalSeq  << ", totalPacket=" << totalPacket << ")" << endl;    // 如果所有包都已发送，假设传输完成  if (totalSeq >= totalPacket) {  cout << "[假定] 传输可能已成功完成" << endl;  transferComplete = true;  } else {  cout << "[失败] 传输失败" << endl;  transferComplete = true;  }  } else {  // 超时,重传  timeoutHandler(sockServer, addrClient);  timer = 0;  }  }  Sleep(1); // 短暂休眠  }  }  cout << "\n======================================" << endl;  cout << "[完成] GBN协议测试完成!" << endl;  cout << "总共发送: " << totalPacket << " 个数据包" << endl;  cout << "======================================" << endl;  }  else {  // 回显  sendto(sockServer, recvBuffer, recvLen, 0,  (sockaddr\*)&addrClient, addrClientLen);  cout << "[响应] 回显数据" << endl;  }  }  Sleep(1);  }  // 清理  closesocket(sockServer);  WSACleanup();  return 0;  }  客户端代码 client.cpp  #include <iostream>  #include <winsock2.h>  #include <ws2tcpip.h>  #include <string>  #include <cstring>  #include <cstdlib>  #include <ctime>  #pragma comment(lib, "ws2\_32.lib")  using namespace std;  // 客户端配置  #define SERVER\_PORT 12340  #define SERVER\_IP "127.0.0.1"  #define BUFFER\_LENGTH 1026  #define SEQ\_SIZE 16 // 序列号个数(n=4, 2^4=16, 序列号范围0-15)  // 数据帧结构  struct DataFrame {  unsigned char seq; // 序列号  char data[1024]; // 数据  unsigned char flag; // 标志位: 0=普通帧, 1=结束帧  };  // ACK帧结构  struct AckFrame {  unsigned char ack; // ACK序列号  unsigned char flag; // 标志位  };  // 全局变量  float packetLossRate = 0.0f; // 数据包丢失率  float ackLossRate = 0.0f; // ACK丢失率  // 模拟丢包  bool simulateLoss(float lossRate) {  if (lossRate <= 0.0f) return false;  float random = (float)rand() / RAND\_MAX;  return random < lossRate;  }  int main() {  srand((unsigned int)time(NULL));  // 初始化Winsock  WSADATA wsaData;  int result = WSAStartup(MAKEWORD(2, 2), &wsaData);  if (result != 0) {  cout << "WSAStartup失败: " << result << endl;  return 1;  }  // 创建socket  SOCKET sockClient = socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP);  if (sockClient == INVALID\_SOCKET) {  cout << "创建socket失败" << endl;  WSACleanup();  return 1;  }  // 设置服务器地址  sockaddr\_in addrServer;  addrServer.sin\_family = AF\_INET;  addrServer.sin\_port = htons(SERVER\_PORT);  inet\_pton(AF\_INET, SERVER\_IP, &addrServer.sin\_addr);  cout << "======================================" << endl;  cout << "GBN协议客户端" << endl;  cout << "服务器地址: " << SERVER\_IP << ":" << SERVER\_PORT << endl;  cout << "======================================" << endl;  cout << "\n可用命令:" << endl;  cout << " -time 获取服务器时间" << endl;  cout << " -testgbn [X] [Y] 测试GBN协议" << endl;  cout << " X: 数据包丢失率(0.0-1.0,默认0.2)" << endl;  cout << " Y: ACK丢失率(0.0-1.0,默认0.2)" << endl;  cout << " -quit 退出程序" << endl;  cout << " 其他文本 回显测试" << endl;  cout << "======================================\n" << endl;  // 主循环  while (true) {  cout << "\n请输入命令: ";  string input;  getline(cin, input);  if (input.empty()) continue;  // 解析命令  if (input == "-time") {  // 请求时间  sendto(sockClient, input.c\_str(), input.length(), 0,  (sockaddr\*)&addrServer, sizeof(addrServer));  char recvBuffer[BUFFER\_LENGTH];  sockaddr\_in fromAddr;  int fromLen = sizeof(fromAddr);  int recvLen = recvfrom(sockClient, recvBuffer, BUFFER\_LENGTH, 0,  (sockaddr\*)&fromAddr, &fromLen);  if (recvLen > 0) {  recvBuffer[recvLen] = '\0';  cout << "[服务器时间] " << recvBuffer << endl;  }  }  else if (input == "-quit") {  // 退出  sendto(sockClient, input.c\_str(), input.length(), 0,  (sockaddr\*)&addrServer, sizeof(addrServer));  char recvBuffer[BUFFER\_LENGTH];  sockaddr\_in fromAddr;  int fromLen = sizeof(fromAddr);  int recvLen = recvfrom(sockClient, recvBuffer, BUFFER\_LENGTH, 0,  (sockaddr\*)&fromAddr, &fromLen);  if (recvLen > 0) {  recvBuffer[recvLen] = '\0';  cout << "[服务器] " << recvBuffer << endl;  }  break;  }  else if (input.substr(0, 8) == "-testgbn") {  // 测试GBN协议  packetLossRate = 0.2f;  ackLossRate = 0.2f;  // 解析参数  size\_t pos = 8;  if (input.length() > pos) {  // 跳过空格  while (pos < input.length() && input[pos] == ' ') pos++;  // 解析第一个参数  if (pos < input.length()) {  size\_t nextSpace = input.find(' ', pos);  string param1;  if (nextSpace != string::npos) {  param1 = input.substr(pos, nextSpace - pos);  pos = nextSpace + 1;  } else {  param1 = input.substr(pos);  pos = input.length();  }  try {  packetLossRate = stof(param1);  if (packetLossRate < 0.0f) packetLossRate = 0.0f;  if (packetLossRate > 1.0f) packetLossRate = 1.0f;  } catch (...) {  cout << "[警告] 无效的数据包丢失率,使用默认值0.2" << endl;  }  // 跳过空格  while (pos < input.length() && input[pos] == ' ') pos++;  // 解析第二个参数  if (pos < input.length()) {  string param2 = input.substr(pos);  try {  ackLossRate = stof(param2);  if (ackLossRate < 0.0f) ackLossRate = 0.0f;  if (ackLossRate > 1.0f) ackLossRate = 1.0f;  } catch (...) {  cout << "[警告] 无效的ACK丢失率,使用默认值0.2" << endl;  }  }  }  }  cout << "\n[开始] GBN协议测试" << endl;  cout << "数据包丢失率: " << (packetLossRate \* 100) << "%" << endl;  cout << "ACK丢失率: " << (ackLossRate \* 100) << "%" << endl;  cout << "======================================" << endl;  // 发送测试命令  string testCmd = "-testgbn";  sendto(sockClient, testCmd.c\_str(), testCmd.length(), 0,  (sockaddr\*)&addrServer, sizeof(addrServer));  // 接收数据  int expectedSeq = 0; // 期望的序列号  int receivedCount = 0; // 接收计数  int lostPacketCount = 0; // 丢包计数  int lostAckCount = 0; // ACK丢失计数  // 设置接收超时  DWORD timeout = 15000; // 15秒  setsockopt(sockClient, SOL\_SOCKET, SO\_RCVTIMEO,  (char\*)&timeout, sizeof(timeout));  while (true) {  DataFrame frame;  sockaddr\_in fromAddr;  int fromLen = sizeof(fromAddr);  int recvLen = recvfrom(sockClient, (char\*)&frame, sizeof(DataFrame), 0,  (sockaddr\*)&fromAddr, &fromLen);  if (recvLen > 0) {  cout << "\n[接收] Seq=" << (int)frame.seq  << ", 数据=\"" << frame.data << "\"";  // 模拟数据包丢失  if (simulateLoss(packetLossRate)) {  cout << " -> [丢弃] 模拟数据包丢失!" << endl;  lostPacketCount++;  continue;  }  cout << endl;  // GBN协议:只接受期望的序列号  if (frame.seq == expectedSeq) {  cout << "[接受] 序列号正确,发送ACK=" << (int)expectedSeq << endl;  // 发送累积确认ACK  AckFrame ackFrame;  ackFrame.ack = expectedSeq;  ackFrame.flag = 0;  // 模拟ACK丢失  if (simulateLoss(ackLossRate)) {  cout << "[丢失] 模拟ACK丢失!" << endl;  lostAckCount++;  } else {  sendto(sockClient, (char\*)&ackFrame, sizeof(AckFrame), 0,  (sockaddr\*)&addrServer, sizeof(addrServer));  cout << "[发送] ACK=" << (int)expectedSeq << endl;  }  receivedCount++;  expectedSeq = (expectedSeq + 1) % SEQ\_SIZE;  // 检查是否为最后一帧  if (frame.flag == 1) {  cout << "\n[完成] 收到结束标志!" << endl;  cout << "======================================" << endl;  cout << "接收统计:" << endl;  cout << " 成功接收: " << receivedCount << " 个数据包" << endl;  cout << " 丢弃数据包: " << lostPacketCount << " 个" << endl;  cout << " 丢失ACK: " << lostAckCount << " 个" << endl;  cout << "======================================" << endl;  break;  }  } else {  // 收到非期望序列号,发送上一个ACK(累积确认)  int prevAck = (expectedSeq - 1 + SEQ\_SIZE) % SEQ\_SIZE;  cout << "[拒绝] 序列号不匹配! 期望=" << expectedSeq  << ", 实际=" << (int)frame.seq << endl;  cout << "[重发] 发送上一个ACK=" << prevAck << endl;  AckFrame ackFrame;  ackFrame.ack = prevAck;  ackFrame.flag = 0;  // 模拟ACK丢失  if (simulateLoss(ackLossRate)) {  cout << "[丢失] 模拟ACK丢失!" << endl;  lostAckCount++;  } else {  sendto(sockClient, (char\*)&ackFrame, sizeof(AckFrame), 0,  (sockaddr\*)&addrServer, sizeof(addrServer));  }  }  } else if (recvLen == SOCKET\_ERROR) {  int error = WSAGetLastError();  if (error == WSAETIMEDOUT) {  cout << "\n[超时] 未收到更多数据,测试结束" << endl;  cout << "======================================" << endl;  cout << "接收统计:" << endl;  cout << " 成功接收: " << receivedCount << " 个数据包" << endl;  cout << " 丢弃数据包: " << lostPacketCount << " 个" << endl;  cout << " 丢失ACK: " << lostAckCount << " 个" << endl;  cout << "======================================" << endl;  break;  } else {  cout << "[错误] 接收失败: " << error << endl;  break;  }  }  }  }  else {  // 回显测试  sendto(sockClient, input.c\_str(), input.length(), 0,  (sockaddr\*)&addrServer, sizeof(addrServer));  char recvBuffer[BUFFER\_LENGTH];  sockaddr\_in fromAddr;  int fromLen = sizeof(fromAddr);  int recvLen = recvfrom(sockClient, recvBuffer, BUFFER\_LENGTH, 0,  (sockaddr\*)&fromAddr, &fromLen);  if (recvLen > 0) {  recvBuffer[recvLen] = '\0';  cout << "[回显] " << recvBuffer << endl;  }  }  }  // 清理  closesocket(sockClient);  WSACleanup();  cout << "\n客户端已关闭" << endl;  return 0;  }  选做实验：  SR协议实现： SR vs GBN 对比  | 特性 | GBN协议 | SR协议 | | --- | --- | --- | | 接收方缓冲 | 无缓冲，只接受按序到达的包 | 有缓冲，可接收乱序包 | | 重传策略 | 超时重传整个窗口 | 只重传丢失的包 | | ACK机制 | 累积确认 | 独立确认每个包 | | 效率 | 丢包时效率低 | 丢包时效率高 | | 复杂度 | 实现简单 | 实现复杂 |  主要改进1. 发送方改进  * **独立计时器**：每个包有独立的超时计时 * **选择性重传**：只重传超时的包，不影响其他已确认的包 * **窗口管理**：使用map存储每个包的状态（是否确认、发送时间、重传次数）  2. 接收方改进  * **接收缓冲区**：使用map缓存乱序到达的数据包 * **接收窗口**：维护一个接收窗口，接受窗口内的任何包 * **按序交付**：缓存的数据按序交付给应用层 * **独立ACK**：对每个正确接收的包都发送ACK  3. 关键数据结构 **发送方窗口状态：**  struct SendWindow {  bool acked; // 是否已确认  clock\_t sendTime; // 发送时间(用于超时判断)  int retryCount; // 重传次数 }; map<int, SendWindow> sendWindow; // 序列号 -> 状态  **接收方缓冲区：**  struct RecvBuffer {  DataFrame frame; // 数据帧  bool received; // 是否已接收 }; map<int, RecvBuffer> recvWindow; // 序列号 -> 缓冲 |
| 实验结果： |
| 采用演示截图、文字说明等方式，给出本次实验的实验结果。  Lab2-1 Basic 停等协议实现：    选做1 双向传输      文件传输：    Lab2-2 GBN 协议实现    SR协议实现（文件传输）： |
| 问题讨论： |
| 实验过程中，我发现一个写代码时没有想到的情况：当服务端给客户端发最后一个数据块时，客户端成功接收，但ACK意外丢失，此时客户端已经完成接收结束了这次任务，并且不知道ACK丢了，但服务端会误以为客户端没有丢，从而不断重传。面对这种情况，可以对最后一个数据块增加一个超时控制，限定时间内没有收到ACK就结束任务，并且报告客户端已经正确接受只是ACK中途丢失了。 |
| 心得体会： |
| 熟练掌握 socket 编程，能够利用 udp 协议实现基本的停等协议、GBN协议、SR协议在服务端和客户端之间传输数据。  经过这次实验，更加深刻地理解了三种协议的工作流程，并且直观体会到了三种协议的性能差距。 |