表格

AI 生成的内容可能不正确。Wireshark包捕获的截图

**实现关键点**

* 自定义协议首部以模拟 TCP 的连接建立和可靠传输
* 定义不同类型的报文（SYN、SYN\_ACK、ACK、DATA、FIN）
* 设计序列号和确认号机制以跟踪数据传输状态

**代码解决方案**

// Packet类定义协议首部结构

public class Packet {

// 报文类型定义

public static final byte SYN = 0x1;

public static final byte SYN\_ACK = 0x2;

public static final byte ACK = 0x4;

public static final byte DATA = 0x8;

public static final byte FIN = 0x10;

// 首部字段

private byte type;

private int seqNumber; // 序列号（字节偏移）

private int ackNumber; // 确认号（字节偏移）

private long timestamp; // 发送时间戳

private byte[] payload; // 数据负载

// 序列化与反序列化方法

public byte[] serialize() {

// 构造字节数组，包含首部和数据负载

int payloadLength = (payload != null) ? payload.length : 0;

ByteBuffer buffer = ByteBuffer.allocate(17 + payloadLength);

buffer.put(type);

buffer.putInt(seqNumber);

buffer.putInt(ackNumber);

buffer.putLong(timestamp);

if (payload != null) {

buffer.put(payload);

}

return buffer.array();

}

public static Packet deserialize(byte[] data) {

// 从字节数组解析报文

ByteBuffer buffer = ByteBuffer.wrap(data);

byte type = buffer.get();

int seq = buffer.getInt();

int ack = buffer.getInt();

long timestamp = buffer.getLong();

byte[] payload = null;

if (buffer.remaining() > 0) {

payload = new byte[buffer.remaining()];

buffer.get(payload);

}

return new Packet(type, seq, ack, payload);

}

}

private byte type;

private int seqNumber; // 序列号（字节偏移）

private int ackNumber; // 确认号（字节偏移）

private long timestamp; // 发送时间戳

private byte[] payload; // 数据负载

// 序列化与反序列化方法

public byte[] serialize() {

// 构造字节数组，包含首部和数据负载

int payloadLength = (payload != null) ? payload.length : 0;

ByteBuffer buffer = ByteBuffer.allocate(17 + payloadLength);

buffer.put(type);

buffer.putInt(seqNumber);

buffer.putInt(ackNumber);

buffer.putLong(timestamp);

if (payload != null) {

buffer.put(payload);

}

return buffer.array();

}

public static Packet deserialize(byte[] data) {

// 从字节数组解析报文

ByteBuffer buffer = ByteBuffer.wrap(data);

byte type = buffer.get();

int seq = buffer.getInt();

int ack = buffer.getInt();

long timestamp = buffer.getLong();

byte[] payload = null;

if (buffer.remaining() > 0) {

payload = new byte[buffer.remaining()];

buffer.get(payload);

}

return new Packet(type, seq, ack, payload);

}

}

}

return new Packet(type, seq, ack, payload);

}

}

**连接建立过程模拟**

**实现关键点**

* 在无连接的 UDP 上模拟 TCP 的三次握手
* 实现 SYN-SYN\_ACK-ACK 的交互流程
* 处理连接建立过程中的超时和重传

**代码解决方案**

// 客户端连接建立

private void establishConnection() throws IOException {

// 发送SYN报文

Packet syn = new Packet(Packet.SYN, 0, 0, null);

sendPacket(syn);

// 等待SYN-ACK响应

byte[] buffer = new byte[1024];

DatagramPacket response = new DatagramPacket(buffer, buffer.length);

socket.receive(response);

Packet synAck = Packet.deserialize(Arrays.copyOf(response.getData(), response.getLength()));

if ((synAck.getType() & Packet.SYN\_ACK) == 0) {

throw new IOException("连接失败");

}

System.out.println("连接已建立");

}

// 服务器端处理SYN请求

private void handleSYN(Packet packet, InetAddress clientAddr, int clientPort) throws IOException {

System.out.println("收到来自 " + clientAddr + " 的连接请求");

// 发送SYN-ACK响应

Packet synAck = new Packet(Packet.SYN\_ACK, 0, packet.getSeqNumber() + 1, null);

sendPacket(synAck, clientAddr, clientPort);

}

**可靠数据传输机制**

**实现关键点**

* 基于 GBN 协议实现滑动窗口机制
* 实现超时重传机制，动态调整超时时间
* 服务器端模拟丢包场景（20% 丢包率）
* 客户端维护发送窗口和重传队列

**代码解决方案**

// 客户端发送窗口管理与超时重传

private void sendNextPacket() throws IOException {

// 确定包大小（40-80字节随机）

int packetSize = Math.min(

MIN\_PACKET\_SIZE + random.nextInt(MAX\_PACKET\_SIZE - MIN\_PACKET\_SIZE + 1),

TOTAL\_DATA - nextSeq

);

// 创建数据包并发送

byte[] payload = new byte[packetSize];

Packet packet = new Packet(Packet.DATA, nextSeq, 0, payload);

sendPacket(packet);

// 记录发送时间和包信息

sendTimes.put(nextSeq, System.currentTimeMillis());

sentPackets.put(nextSeq, packet);

// 输出发送信息

System.out.printf("第 %d 个（第 %d~%d 字节）client 端已经发送%n",

packetCounter, nextSeq, nextSeq + packetSize - 1);

nextSeq += packetSize;

}

// 超时检查机制

private void checkTimeouts() {

long now = System.currentTimeMillis();

int timeout = calculateTimeout();

// 检查所有未确认包是否超时

for (Map.Entry<Integer, Long> entry : new HashMap<>(sendTimes).entrySet()) {

int seq = entry.getKey();

long sendTime = entry.getValue();

if (now - sendTime > timeout) {

int retries = retryCounts.getOrDefault(seq, 0);

if (retries < MAX\_RETRIES) {

// 重发包

Packet packet = sentPackets.get(seq);

sendPacket(packet);

sendTimes.put(seq, now);

retryCounts.put(seq, retries + 1);

totalResent++;

// 输出重传信息

int[] range = packetRanges.get(seq);

System.out.printf("重传第 %d 个（第 %d~%d 字节）数据包%n",

getPacketNumber(seq), range[0], range[1]);

}

}

}

}

// 服务器端模拟丢包

private void handleData(Packet packet, InetAddress clientAddr, int clientPort) throws IOException {

int seq = packet.getSeqNumber();

// 对新包应用丢包率（20%）

if (!processedPackets.contains(seq) && random.nextDouble() < LOSS\_RATE) {

System.out.printf("模拟丢包: 字节 %d~%d%n", seq, seq + packet.getLength() - 1);

return;

}

// 处理正常接收的包

processedPackets.add(seq);

if (seq == expectedSeq) {

expectedSeq += packet.getLength();

processBufferedPackets(); // 处理乱序包缓存

} else {

outOfOrder.put(seq, packet.getPayload()); // 缓存乱序包

}

// 发送累积确认

Packet ack = new Packet(Packet.ACK, 0, expectedSeq, null);

sendPacket(ack, clientAddr, clientPort);

System.out.printf("确认字节: %d (累积确认)%n", expectedSeq);

}

long now = System.currentTimeMillis();

int timeout = calculateTimeout();

// 检查所有未确认包是否超时

for (Map.Entry<Integer, Long> entry : new HashMap<>(sendTimes).entrySet()) {

int seq = entry.getKey();

long sendTime = entry.getValue();

if (now - sendTime > timeout) {

int retries = retryCounts.getOrDefault(seq, 0);

if (retries < MAX\_RETRIES) {

// 重发包

Packet packet = sentPackets.get(seq);

sendPacket(packet);

sendTimes.put(seq, now);

retryCounts.put(seq, retries + 1);

totalResent++;

// 输出重传信息

int[] range = packetRanges.get(seq);

System.out.printf("重传第 %d 个（第 %d~%d 字节）数据包%n",

getPacketNumber(seq), range[0], range[1]);

}

}

}

}

// 服务器端模拟丢包

private void handleData(Packet packet, InetAddress clientAddr, int clientPort) throws IOException {

int seq = packet.getSeqNumber();

// 对新包应用丢包率（20%）

if (!processedPackets.contains(seq) && random.nextDouble() < LOSS\_RATE) {

System.out.printf("模拟丢包: 字节 %d~%d%n", seq, seq + packet.getLength() - 1);

return;

}

// 处理正常接收的包

processedPackets.add(seq);

if (seq == expectedSeq) {

expectedSeq += packet.getLength();

processBufferedPackets(); // 处理乱序包缓存

} else {

outOfOrder.put(seq, packet.getPayload()); // 缓存乱序包

}

// 发送累积确认

Packet ack = new Packet(Packet.ACK, 0, expectedSeq, null);

sendPacket(ack, clientAddr, clientPort);

System.out.printf("确认字节: %d (累积确认)%n", expectedSeq);

}

**实验涉及的知识点**

**计算机网络基础**

* **UDP 协议**：无连接、不可靠的传输层协议特点
* **TCP 协议**：面向连接、可靠传输的核心机制（三次握手、滑动窗口、超时重传、累积确认等）
* **应用层协议设计**：自定义协议首部字段和交互流程

**Socket 编程**

* **UDP Socket API**：DatagramSocket、DatagramPacket 的使用
* **网络字节序**：数据在网络传输中的字节顺序处理
* **客户端 - 服务器架构**：命令行参数传递（服务器 IP 和端口）

**可靠传输机制**

* **GBN 协议**：滑动窗口、累积确认、超时重传
* **RTT 估算**：往返时间测量与超时时间动态调整
* **丢包模拟**：在应用层实现数据包随机丢弃

**多线程编程**

* **并发处理**：接收 ACK 线程与发送线程的并行执行
* **定时任务**：使用 ScheduledExecutorService 实现超时检查
* **线程安全**：使用 ConcurrentHashMap 等线程安全集合类

**性能统计与分析**

* **描述性统计**：平均值、最大值、最小值、标准差的计算
* **丢包率计算**：重传次数与总发送次数的比率
* **数据记录与输出**：运行时状态信息打印与汇总统计

**通过本次实验增长点**

**TCP 可靠传输机制在 UDP 上的实现**

本实验的核心在于理解 TCP 可靠传输机制的本质，并在无连接的 UDP 之上通过应用层协议实现这些机制。主要包括：

* 连接建立与释放的状态管理
* 序列号与确认号机制保证数据有序性
* 超时重传与滑动窗口处理丢包和网络延迟
* 累积确认机制提高传输效率

**应用层协议设计思想**

自定义协议首部是实现可靠传输的基础，需要考虑：

* 字段设计：类型、序列号、确认号、时间戳等必要字段
* 协议交互流程：连接建立、数据传输、连接释放的状态转移
* 兼容性与扩展性：首部格式的可解析性和未来功能扩展空间

**网络性能测量与分析**

通过 RTT 和丢包率等指标理解网络传输质量：

* RTT 变化反映网络延迟波动
* 丢包率反映网络拥塞程度
* 统计分析方法用于量化网络性能

**理论与实践的结合**

本实验将计算机网络理论知识（如 GBN 协议、TCP 可靠传输）转化为实际代码实现，关键在于：

* 理解协议原理与代码实现的映射关系
* 处理边界情况（如包大小随机、乱序到达、丢包重传）
* 性能优化与错误处理机制设计

代码运行截图：





