高级计算机网络

实验报告

学　　员：_____　　学　　号：_____

专　　业：_____　　年　　级：_____

所属学院：_____　　授课教员：_____

国防科技大学计算机学院

College of Computer Science and Technology

目 录

【摘　要】 实验在有三台主机的网络环境中展开，三台连接在同一个网络上的 Linux 主机，分别为源（Source）节点、目标（Target）节点和监控（Monitor）节点。三台主机处于同一网段中，且未在网络中部署防火墙。监控节点可以通过 SSH 登录到源节点和目标节点上。作品程序的服务端运行在源节点上，客户端运行在目标节点上，而作为监控的工具软件运行在监控节点上，负责测试数据生成、正确性检查和实验运行时间的测量等工作。我们编写一套基于 UDP 协议能尽快地通过不确定网络正确传输一组确定数据的网络程序。我们在原有的 UDP 协议上对数据报数据进行封装，然后根据封装的格式实现了逐帧传输，回退 n 帧和快速重传等三种可靠传输协议。这三种方式在不同情况下保证了文件数据的可靠传输；接着为了进一步增加传输速率，实现了两种压缩算法，分别是多线程 lzma 算法压缩和 zstd 算法压缩。

【关键词】 UDP，**可靠性传输协议**，**文件压缩**

1　引 言

近年来，随着互联网的普及和人们对高速网络的需求日益增加，网络通信技术也发生了巨大的变化。UDP（用户数据报协议）是一种无连接的、无状态的、基于数据报的传输层协议，在许多网络应用中得到了广泛使用。然而，UDP 协议本身并不提供可靠性保证，因此在许多情况下，数据报可能会丢失或出错。为了解决这一问题，我们开发了一套基于 UDP 协议的网络程序，旨在尽快地通过不确定网络正确传输一组确定数据。

我们的研究在三台 Linux 主机的网络环境中进行，包括源（Source）节点、目标（Target）节点和监控（Monitor）节点。这三台主机均处于同一网段中，并且未在网络中部署防火墙。我们的程序服务端运行在源节点上，客户端运行在目标节点上，而监控工具软件运行在监控节点上，负责测试数据生成、正确性检查和实验运行时间的测量等工作。

为了提高传输可靠性，我们对基于 UDP 的数据报进行了封装，并实现了逐帧传输、回退 n 帧和快速重传三种可靠传输协议。这三种方法在不同情况下均能保证文件数据的可靠传输。

此外，为了进一步提高传输速率，我们还实现了两种压缩算法，分别是多线程 lzma 算法压缩和 zstd 算法压缩。这两种算法能够有效地减少数据的体积，从而提高传输速率。

在本文中，我们将详细介绍我们所开发的网络程序，并通过实验结果对其进行评估。我们的研究结果表明，通过对 UDP 协议的改进和压缩算法的应用，我们的程序能够在保证可靠性的同时，显著提高传输速率。

2　实验内容

本课程实验的需要基于套接字通信编写一套能尽快地通过不确定网络正确传输一组确定数据的网络程序。该程序具有如下特点：

- 作品程序支持将源节点上的指定数据文件读出，并传输到目标节点后存储为指定文件；
- 传输后生成的文件应该与原始文件数据上完全一致（可以通过 MD5 校验）；
- 数据传输发生在一个既定网络中的两个节点之间，数据将从其中的源节点传输到目标节点；
- 从源节点到目标节点的网络链路存在特定的网络传输特性，例如可能存在具有一定随机性的网络传输延迟与一定比例的网络丢包。
- 需要传输的数据会预先生成，并存储为源节点上的指定位置的一个文件，文件数据大小确定、并具有一定特征。
- 源节点、目标节点的处理器能力、内存大小、磁盘空间均有既定限制。

这套程序需要完成以下目标：

- 独立编程实现，采用的编程语言不限；不可抄袭；可以使用第三方库，但在使用前需要向教员说明并得到允许。
- 程序的网络传输代码必须基于 UDP 实现，不可在 TCP 之上实现。
- 作品程序能在源节点和目标节点上分别建立服务器端与客户端，并在指定的时刻开始运行。
- 数据传输时间应该尽量短，评测计时以客户端运行时间计算，即检测时假设客户端启动时即开始数据传输，完成数据存储后立刻退出。计

时从客户端程序开始运行时起，客户端程序退出时止。

2.1 进度安排

本课程实验计划按照课程的安排持续推进，将实验内容主要分为以下部分：

1. 需求分析：在该阶段，主要完成实验目标、实验内容和实验环境的熟悉工作，该部分预计工作时间为 1 周。
2. 概要设计：在该阶段，主要完成相关实验代码的熟悉以及实验环境运行原理工作，该部分预计工作时间为 2 周。
3. 功能实现：在该阶段，主要完成 udp 可靠性传输功能的代码实现，包括发送数据时的封帧和接受数据时的解包；逐帧传输、回退 n 帧和快速重传的机制；多线程 lamz 压缩算法以及 zstd 压缩算法的实现，该部分预计工作时间为 4 周。
4. 联调测试：在该阶段，主要进行代码的联调和测试，需要具体完成整体代码调试，实验数据分析工作，该部分预计工作时间为 1 周。

2.2 实验环境

本实验在 Windows 下进行开发，并利用虚拟机构建虚拟的局域网络进行联调与测试，主要的软硬件实验环境主要包括：

2.2.1 硬件环境

实验环境共包含三台连接在同一个网络上的 Linux 主机，分别为源（Source）节点、目标（Target）节点和监控（Monitor）节点。其网络拓扑如图1所示。

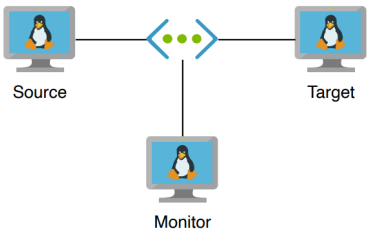


图 1 硬件网络拓扑结构

三台主机处于同一网段中，且未在网络中部署防火墙。监控节点可以通过 SSH 登录到源节点和目标节点上。作品程序的服务端运行在源节点上，客户端运行在目标节点上，而作为监控的工具软件运行在监控节点上，负责测试数据生成、正确性检查和实验运行时间的测量等工作。

2.2.2 软件环境

实验过程中用到的软件如下：

- 操作系统
  - Windows 11，开发笔记本操作系统
  - Ubuntu 20.04，测试服务器操作系统
  - Ubuntu 18.04，测试服务器操作系统
- 开发环境
  - VMware16，用于搭建虚拟测试网络
  - VSCode，集成开发环境
- 测试工具
  - tc 软件
  - python

3 基于 UDP 的可靠数据传输

本节介绍实验的具体过程，本实验需要利用 UDP 实现 server 与 client 的基本通信，保证传输后生成的文件与原始文件数据上完全一致。

3.1 实验要求

在本阶段，需要完成 Socket 套接字编程，使程序具体达成以下能力：

- 作品程序支持将源节点上的指定数据文件读出，并传输到目标节点后存储为指定文件；
- 传输后生成的文件应该与原始文件数据上完全一致（可以通过 MD5 校验）；
- 作品程序运行的时间应该尽量少，计时从客户端开始运行时起，到文件完全存储在目标节点时止；

3.2 实验内容

为了实现上述实验要求，拟分别采用 python 语言进行开发，本部分的实验工作具体可分为“可靠性传输实现”，“压缩算法实现”两个方面，下面分别进行详细说明。

3.2.1 可靠性传输实现

整个可靠性传输架构围绕 server 和 client 展开，并分别实现了逐帧传输、回退 n 帧和选择性重传三种可靠性传输协议。基于上述思路其结构如下图2所示。

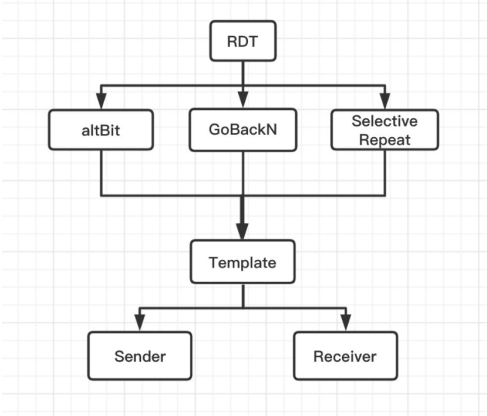


图 2 可靠性传输架构图

3.2.2　RDT 类实现

RDT 类主要实现的是数据发送时的封帧和数据接收时的解包，该方法的关键代码片段如表1所示。

表 1 数据发送时的封帧和数据接收时的解包

```
def make_pkt(self, seq_num, ack_num, pkg_len,
↪  payload: bytes) -> dict:
    pkt = self.init_pkt()
    pkt['SEQ_N'] = seq_num
    pkt['ACK_N'] = ack_num
    pkt['PKG_LEN'] = pkg_len
    pkt['DATA'][0:len(payload)] = payload
    # print(len(payload))
    pkt['DATA'] = bytes(pkt['DATA'])
    pkt['CHECK_SUM'] = self.calc_checksum(pkt)
    return pkt

def encoder_pkt(self, pkt: dict) -> bytes:
    head = struct.pack('!4i', pkt['SEQ_N'],
    ↪  pkt['ACK_N'], pkt['PKG_LEN'],
    ↪  pkt['CHECK_SUM'])
    raw_bytes = head + pkt['DATA']
    return raw_bytes

def decode_pkt(self, raw_bytes: bytes) -> dict:
    pkt = self.init_pkt()
    pkt['DATA'] = raw_bytes[-self.MSS:]
    head = raw_bytes[:-self.MSS]
    pkt['SEQ_N'], pkt['ACK_N'], pkt['PKG_LEN'],
    ↪  pkt['CHECK_SUM'] = struct.unpack('!4i',
    ↪  head)
    return pkt
```

3.2.3　altBit 类实现

altBit 类主要实现的是数据单帧发送和接收时的可靠性传输实现，该方法的关键代码片段如表2所示。

表 2 数据单帧发送和接收时的可靠性传输实现

```
class altBit(RDT):
    def __init__(self) -> None:
        super(altBit, self).__init__()
        self.state = 0

    def send(self, content: bytes) -> None:
        pkt_num = self.get_pkt_num(content)
        self.inform(f"Total Pkt Num = {pkt_num}")
        offset = 0
        for i in range(pkt_num):
            self.inform(f"Sending Pkt Num = {i +
            ↪  1}")
            self.add_send_time(1)
            # 序列号，确认号，剩余包长，数据
            self.pkt_send =
            ↪  self.make_pkt(self.state, -1,
            ↪  pkt_num - i - 1,
            ↪  content[offset:offset +
            ↪  self.MSS])
            self.send_pkt(self.pkt_send)
            self.start_timer()
            pkt_recv = self.recv_pkt()
            self.stop_timer()
            if (self.check_pkt(pkt_recv) and
            ↪  self.is_ack(pkt_recv,
            ↪  self.state)):
                self.inform(f"Recv Right
                ↪  ACK{self.state}")
                self.add_success_time(1)
                self.change_state()
            else:
                self.inform(f"Recv Wrong ACK,
                ↪  resend Seq{self.state}")
                self.resend()
                self.start_timer()
            offset += self.MSS

    def recv(self) -> list:
        result = None
        while (result is None):
            pkt = self.recv_pkt()
            if (self.check_pkt(pkt) and
            ↪  self.is_seq(pkt, self.state)):
                self.send_ack(pkt['SEQ_N'])
                self.change_state()
                result = [pkt]
            else:
                self.send_ack(not self.state)
        return result
```

3.2.4 goBackN 类实现

Go-Back-N 协议是一种用于可靠数据传输的滑动窗口协议，可确保接收方接收到所有数据包。goBackN 类是 RDT 的子类，代表可靠数据传输。我们定义了几种用于实现 Go-Back-N 协议的方法，包括：

- __init__：该方法初始化 goBackN 对象并设置必要的实例变量，如发送缓冲区、窗口长度和序列号。
- is_ack：此方法接收一个数据包，如果该数据包是确认数据包，则返回 True，否则返回 False。
- change_state：此方法根据接收方已成功确认的数据包数量更新滑动窗口的状态。
- save_to_buffer：此方法将给定消息作为一系列数据包保存到发送缓冲区，每个数据包的最大大小为 MSS（最大段大小）。
- send_range：此方法从发送缓冲区发送一定范围的数据包。
- resend：该方法重新发送当前窗口中的所有数据包。
- send：此方法使用 Go-Back-N 协议向接收方发送消息。它首先将消息分成数据包并将它们保存到发送者缓冲区。然后它在当前窗口中发送数据包并启动计时器。它等待来自接收方的确认数据包，如果收到正确的确认数据包，它会停止计时器并移动窗口以发送下一批数据包。如果它收到一个错误的确认包或根本没有确认包，它会重新发送当前窗口中的所有包。
- recv：此方法使用 Go-Back-N 协议从发送方接收消息。它等待具有预期序列号的数据包，将确认数据包发送回发送方，并将数据包的有效负载存储在列表中。如果它收到一个序列号不正确的数据包，它会发送一个带有预期序列号的确认数据包。

3.2.5 selRepeat 类实现

selRepeat 是用于可靠数据传输 (RDT) 的选择性重复协议的实现。selRepeat 类派生自基类 RDT，它重载了多个方法以提供选择性重复功能。

- __init__ 方法初始化协议中使用的各种变量，例如窗口大小、用于存储发送和接收数据包的缓冲区以及用于跟踪数据包的序列号。
- is_ack 方法检查接收到的数据包是否是确认数据包。
- change_state 方法将窗口移动一定数量的步长。
- save_to_buffer 方法将要发送的数据包存储在发送缓冲区中。
- send_range 方法在发送缓冲区中发送一系列数据包。
- resend 方法重新发送数据包。
- mark_pkt 方法将数据包标记为在接收缓冲区中已接收。
- get_window_shift 方法根据接收缓冲区中的哪些数据包已标记为已接收来计算窗口应移动的步数。
- send 方法是实现选择性重复协议的主要方法。它首先初始化窗口并发送初始数据包集。然后它进入一个循环，等待接收数据包并相应地处理它们。如果接收到的数据包是有效的确认数据包，则移动窗口并在必要时发送更多数据包。如果接收到的数据包不是确认数据包，则将其存储在接收缓冲区中以供稍后处理。如果发生超时，则重新发送窗口左侧的数据包。循环继续，直到所有数据包都得到确认并且传输完成。
- recv 方法用于接收发送方发起的数据传输。它首先初始化窗口并发送初始确认。然后它进入一个循环，等待接收数据包并相应地处理它们。如果接收到的数据包在当前窗口中，则在接收缓冲区中将其标记为已接收并发送确认。如果接收到的数据包不在当前窗口中，则将其丢弃。如果接收到当前窗口之外的数据包，则移动窗口并为窗口中最左边的新数据包发送确认。循环继续，直到收到所有数据包并完成传输。

3.2.6 压缩算法实现

我们总共尝试使用了两种压缩算法对原始生成的文件数据进行压缩，分别是多线程 lzma 算法和 zstd 算法。多线程 lzma 算法使用文件对象的 read 方法以大小为 chunk_size 的块读取输入文件。然后它为每个块创建一个新线程以使用 compress_chunk 函数压缩块。compress_chunk 函数使用 lzma 模块压缩数据并将压缩数据附加到 compressed_data_buffer。一旦所有线程都完成执行，压

缩数据将写入输出文件。该方法的关键代码片段如表3所示。

表 3 多线程 lzma 算法

```
def compress_file1(input_file, output_file,
↪  chunk_size=100*1024*1024):
    def compress_chunk(data, buffer):
        compressed_data = lzma.compress(data)
        buffer.append(compressed_data)

    # Read the input file in chunks
    with open(input_file, 'rb') as input_file:
        chunks = []
        while True:
            chunk = input_file.read(chunk_size)
            if not chunk:
                break
            chunks.append(chunk)

    # Create a buffer to hold the compressed data
    compressed_data_buffer = []

    # Compress the chunks in parallel
    threads = []
    for chunk in chunks:
        thread =
        ↪  threading.Thread(target=compress_chunk,
        ↪  args=(chunk, compressed_data_buffer))
        thread.start()
        threads.append(thread)

    # Wait for all threads to finish
    for thread in threads:
        thread.join()

    # Write the compressed data to the output
    ↪  file
    with open(output_file, 'wb') as output_file:
        for chunk in compressed_data_buffer:
            output_file.write(chunk)
```

zstd 算法使用 zstd 模块中的 ZstdCompressor 类来压缩输入文件并将压缩数据写入输出文件 ZstdCompressor 类的级别参数决定了压缩级别，级别越高，压缩效果越好，但压缩时间越长。该方法的关键代码片段如表4所示。

3.3　实验结果

通过上述实验，成功在系统中实现了 server 与 client 之间的无误通信。具体测试结果如下图所示。我们首先使用 zstd 算法 + 选择性重传算法（三种传输算法中表现效果最好）在延迟为 100ms, 丢包丢设置为 1% 的条件下进行实验。

表 4 zstd 算法

```
def compress_file2(input_file, output_file):
    with open(input_file, 'rb') as f_in,
    ↪  open(output_file, 'wb') as f_out:
    # 创建压缩器
        compressor =
        ↪  zstd.ZstdCompressor(level=22)
    # 将文件压缩到另一个文件
        compressor.copy_stream(f_in, f_out)
```

- 不同压缩系数下传输 0.01G 文件时的实验结果，如图3所示。

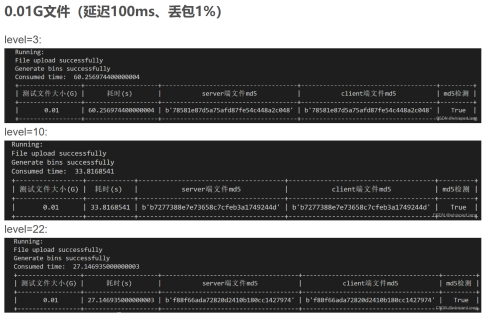


图 3 zstd 传输 0.01G 文件结果

- 不同压缩系数下传输 0.1G 文件时的实验结果，如图4所示。

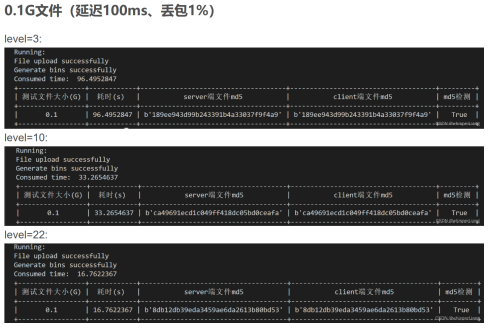


图 4 zstd 传输 0.1G 文件结果

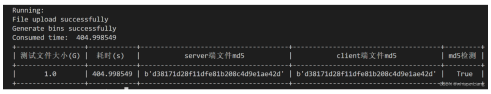
- 不同压缩系数下传输 1G 文件时的实验结果，如图5所示。

接着使用 lzma 算法 + 选择性重传算法在延迟为 100ms, 丢包丢设置为 1% 的条件下进行实验。

- 不同压缩系数下传输 0.01G 文件时的实验结果，如图6所示。
- 不同压缩系数下传输 0.1G 文件时的实验结果，如图7所示。
- 不同压缩系数下传输 1G 文件时的实验结果，如图8所示。可以看到这时 md5 检测值为 False，

1G文件（延迟100ms、丢包1%）

level=3:



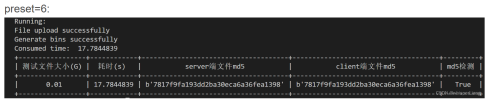
level=10:

level=22:

图 5 zstd 传输 1G 文件结果

0.01G文件（延迟100ms、丢包1%）

preset=6:



preset=9:

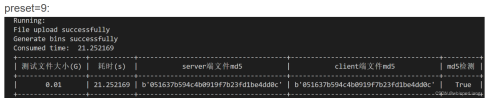


图 6 lzma 传输 0.01G 文件结果

这是由于当 lzma 压缩大文件时，文件受损，且压缩时间长，系统直接杀死进程。

最后我们为了确定快速重传算法的最佳窗口数量进行了相关实验，实验结果如下表5所示。由

表 5 最佳窗口数量确定

size 大小	传输文件规模 (G)	传输时间（s）	文件完整性
10	1	32.34	TRUE
50	1	18.54	TRUE
80	1	18.66	TRUE
100	1	22.38	TRUE
200	1	226.98	TRUE

实验数据可得，最佳窗口数量应该选择为 50。

4 实验结果与分析

测试结果显示，通过上面实现可以看出 zstd 压缩算法比多线程 lzma 算法效果较好，且在不同的情况下都能准确地传输文件数据，其中压缩等级 level=22 优于其他等级。当多线程 lzma 算法压缩大文件时会出现系统杀死进程的情况，后续应该学习使用一些工具，例如互斥锁或信号量，来解决线上述问题。最后我们在压缩算法确定为 zstd 算法，可靠传输协议为选择性重传协议时通过相关实验确定了选择性重传算法的最佳窗口数量。

0.1G文件（延迟100ms、丢包1%）

preset=6:

preset=9:

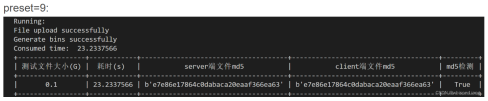
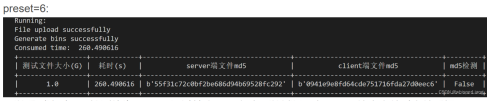


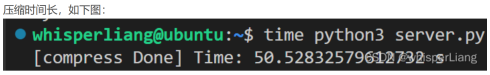
图 7 lzma 传输 0.1G 文件结果

1G文件（延迟100ms、丢包1%）

preset=6:



可以看到这时md5检测值为False，通过结合下面实验可以判断，当lzma压缩大文件时文件受损，且压缩时间长，如下图：



preset=9:

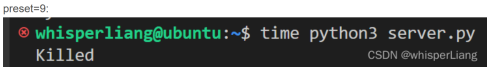


图 8 lzma 传输 1G 文件结果

5 结 论

在这次实验中，我通过对 UDP 协议的封装实现了逐帧传输、回退 n 帧和快速重传三种可靠传输协议，并且通过多线程 LZMA 压缩和 ZSTD 压缩算法来提高传输速率。

通过这次实验，我收获颇丰。首先，我更加熟悉了 UDP 协议的工作原理和特点，并且学会了如何在 UDP 协议的基础上实现可靠传输。其次，我学会了两种压缩算法，并且能够在实际应用中使用它们来提高传输速率。

在实验过程中，我也发现了一些问题。首先，UDP 协议在传输过程中可能会丢包，导致数据传输不完整或出错。其次，在实现多线程压缩时，我们需要考虑线程同步问题，以避免出现冲突。

基于以上问题，我建议在今后的实验中，可以考虑使用更加可靠的传输协议，例如 TCP 协议，来避免数据传输不完整的问题。此外，在实现多线程压缩时，可以使用一些工具，例如互斥锁或信号量，来解决线程同步问题。