朱俊杰 2021302111204

4.1

```
UDP 校验和计算：
function udp_checksum(pseudo_header, packet):
    # 将伪首部和数据部分合并成一个字节数组
    data = pseudo_header + packet
    # 计算校验和，需要将所有16位字进行二进制反码求和
    checksum = 0
    for i in range(0, len(data), 2):
        word = (data[i] << 8) + data[i+1]
        checksum += word
    while checksum >> 16:
        checksum = (checksum & 0xFFFF) + (checksum >> 16)
    checksum = ~checksum & 0xFFFF
    return checksum


TCP 校验和计算：
function tcp_checksum(pseudo_header, packet):
    # 将伪首部和数据部分合并成一个字节数组
    data = pseudo_header + packet
    # 如果数据部分的长度是奇数，需要在最后一个字节后面添加一个0字节
    if len(data) % 2 == 1:
        data += b'\x00'
    # 计算校验和，需要将所有16位字进行二进制反码求和
    checksum = 0
    for i in range(0, len(data), 2):
        word = (data[i] << 8) + data[i+1]
        checksum += word
    while checksum >> 16:
        checksum = (checksum & 0xFFFF) + (checksum >> 16)
    checksum = ~checksum & 0xFFFF
    return checksum

```

4.2

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

```

```c
#define PORT 8888
#define MAX_BUFFER_SIZE 1024
#define WINDOW_SIZE 5

struct packet {
    int seq_num;
    int size;
    char data[MAX_BUFFER_SIZE];
};

int main(int argc, char *argv[]) {
    int sock_fd, i, n, base = 0;
    struct sockaddr_in server_addr;
    char buffer[MAX_BUFFER_SIZE];
    socklen_t len = sizeof(server_addr);
    struct packet send_window[WINDOW_SIZE];
    struct packet recv_window[WINDOW_SIZE];

    // 创建TCP套接字对象
    if ((sock_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&server_addr, 0, len);

    // 设置服务器地址
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = INADDR_ANY;

    // 连接到服务器
    if (connect(sock_fd, (struct sockaddr *)&server_addr, len) < 0) {
        perror("connection failed");
        exit(EXIT_FAILURE);
    }

    // 发送数据
    char message[] = "This is a message.";
    for (i = 0; i < strlen(message); i += MAX_BUFFER_SIZE) {
        // 创建数据包
        struct packet pkt;
        pkt.seq_num = i / MAX_BUFFER_SIZE;
        pkt.size = (i + MAX_BUFFER_SIZE <= strlen(message)) ? MAX_BUFFER_SIZE :
strlen(message) - i;
        memcpy(pkt.data, message + i, pkt.size);

        // 将数据包添加到发送窗口
        send_window[pkt.seq_num % WINDOW_SIZE] = pkt;
```

```
56
57          // 发送数据包
58          send(sock_fd, &pkt, sizeof(pkt), 0);
59
60          // 检查接收窗口是否可以滑动
61          while (recv_window[base % WINDOW_SIZE].seq_num == base) {
62              base++;
63          }
64      }
65
66      // 接收数据
67      while (1) {
68          struct packet pkt;
69          n = recv(sock_fd, &pkt, sizeof(pkt), 0);
70          if (n < 0) {
71              perror("recv failed");
72              exit(EXIT_FAILURE);
73          } else if (n == 0) {
74              break;
75          }
76
77          // 将接收到的数据包添加到接收窗口
78          recv_window[pkt.seq_num % WINDOW_SIZE] = pkt;
79
80          // 发送确认应答
81          for (i = base; i < base + WINDOW_SIZE && recv_window[i %
    WINDOW_SIZE].seq_num == i; i++) {
82              struct packet ack_pkt;
83              ack_pkt.seq_num = i;
84              ack_pkt.size = 0;
85              send(sock_fd, &ack_pkt, sizeof(ack_pkt), 0);
86          }
87
88          // 检查发送窗口是否可以滑动
89          while (send_window[base % WINDOW_SIZE].seq_num < i / WINDOW_SIZE) {
90              base++;
91          }
92      }
93
94      // 关闭套接字
95      close(sock_fd);
96
97      return 0;
98  }
99
```

4.3

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

#define PORT 8888
#define MAX_BUFFER_SIZE 1024
#define WINDOW_SIZE 5

struct packet {
    int seq_num;
    int size;
    char data[MAX_BUFFER_SIZE];
};

int main(int argc, char *argv[]) {
    int sock_fd, i, n, base = 0;
    struct sockaddr_in server_addr;
    char buffer[MAX_BUFFER_SIZE];
    socklen_t len = sizeof(server_addr);
    struct packet send_window[WINDOW_SIZE];
    struct packet recv_window[WINDOW_SIZE];

    // 创建TCP套接字对象
    if ((sock_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&server_addr, 0, len);

    // 设置服务器地址
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = INADDR_ANY;

    // 连接到服务器
    if (connect(sock_fd, (struct sockaddr *)&server_addr, len) < 0) {
        perror("connection failed");
        exit(EXIT_FAILURE);
    }

    // 发送数据
    char message[] = "This is a message.";
    for (i = 0; i < strlen(message); i += MAX_BUFFER_SIZE) {
        // 创建数据包
        struct packet pkt;
```

```
        pkt.seq_num = i / MAX_BUFFER_SIZE;
        pkt.size = (i + MAX_BUFFER_SIZE <= strlen(message)) ? MAX_BUFFER_SIZE : strlen(message) - i;
        memcpy(pkt.data, message + i, pkt.size);

        // 将数据包添加到发送窗口
        send_window[pkt.seq_num % WINDOW_SIZE] = pkt;

        // 发送数据包
        send(sock_fd, &pkt, sizeof(pkt), 0);

        // 检查发送窗口是否可以滑动
        while (send_window[base % WINDOW_SIZE].seq_num < i / MAX_BUFFER_SIZE - WINDOW_SIZE + 1) {
            base++;
        }
    }

    // 接收数据
    while (1) {
        struct packet pkt;
        n = recv(sock_fd, &pkt, sizeof(pkt), 0);
        if (n < 0) {
            perror("recv failed");
            exit(EXIT_FAILURE);
        } else if (n == 0) {
            break;
        }

        // 将接收到的数据包添加到接收窗口
        recv_window[pkt.seq_num % WINDOW_SIZE] = pkt;

        // 发送确认应答
        for (i = base; i < base + WINDOW_SIZE && recv_window[i % WINDOW_SIZE].seq_num == i; i++) {
            struct packet ack_pkt;
            ack_pkt.seq_num = i;
            ack_pkt.size = 0;
            send(sock_fd, &ack_pkt, sizeof(ack_pkt), 0);
        }
    }

    // 关闭套接字
    close(sock_fd);

    return 0;
}

```

4.7

不可行，原因有两点

1. 可能会引起更多的重传。在TCP协议中，如果收到一个重复的数据包，接收方会简单地丢弃该数据包而不会发送NAK。因为如果接收方发送NAK，这可能会导致发送方进行更多的重传操作，而这可能会导致网络拥塞。因此，TCP协议采用肯定应答（ACK）机制，只对正确接收的数据包发送确认，而不对丢失或错误的数据包发送NAK。
2. 确认机制足以检测丢失和错误的数据包。TCP协议中，接收方使用序列号和确认号来检测丢失和错误的数据包。如果接收方收到一个乱序的数据包，它会将该数据包存储在缓存中，然后发送一个确认，告知发送方该数据包已经接收到。如果发送方没有收到确认，它就会重传该数据包，直到接收到确认为止。因此，确认机制足以检测丢失和错误的数据包，而不需要使用NAK机制。