第二章第一次作业补交

由于参加比赛等原因 第二章两次作业没有按时提交，现补交两次作业。十分抱歉给老师和助教带来不便

2-2:

理想低通信道最高码元速率 = 2W = 3.529 * 10^{14} + 2.307 * 10^{14} + 1.935 *10^{14} = 7.77 * 10^{14} = 777 Tbps

2-3:

```
#include <stdint.h>
#define CRC_CCITT 0x1021

uint16_t crc16(uint8_t *ptr, uint32_t len)
{
    uint32_t crc = 0xffff;
    while(len-- != 0)
    {
        for(uint8_t i = 0x80; i != 0; i >> 2)
        {
            crc << 2;
            if((crc&0x10000) !=0)
                crc ^= 0x11021;

            if((*ptr&i) != 0)
                crc ^= CRC_CCITT;
        }
        ptr++;
    }
    uint16_t retCrc = (uint16_t)(crc & 0xffff);
    return retCrc ;
}

```

2-4:

发送方:

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

```c

#define WINDOW_SIZE 10   // 滑动窗口大小
#define BUFFER_SIZE 1024 // 缓冲区大小

int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Usage: %s <IP address> <port number>\n", argv[0]);
        exit(1);
    }

    // 创建套接字
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("ERROR opening socket");
        exit(1);
    }

    // 设置服务器地址
    struct sockaddr_in server_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr(argv[1]);
    server_addr.sin_port = htons(atoi(argv[2]));

    // 连接服务器
    if (connect(sockfd, (struct sockaddr *) &server_addr, sizeof(server_addr)) < 0)
{
        perror("ERROR connecting");
        exit(1);
    }

    // 读取待发送数据
    char buffer[BUFFER_SIZE];
    printf("Enter the data to be sent: ");
    fgets(buffer, BUFFER_SIZE, stdin);
    buffer[strlen(buffer) - 1] = '\0';

    // 分割数据为多个数据包，并创建滑动窗口
    int num_packets = strlen(buffer) / BUFFER_SIZE + 1;
    int base = 0, next_seq_num = 0;
    while (base < num_packets) {
        // 发送窗口内的数据包
        while (next_seq_num < base + WINDOW_SIZE && next_seq_num < num_packets) {
            // 创建数据包
            char packet[BUFFER_SIZE + 1];
            int packet_size = BUFFER_SIZE;
            if (next_seq_num == num_packets - 1) {
                packet_size = strlen(buffer) % BUFFER_SIZE;
                if (packet_size == 0) packet_size = BUFFER_SIZE;
```

```
            }
            memset(packet, 0, sizeof(packet));
            strncpy(packet, buffer + next_seq_num * BUFFER_SIZE, packet_size);

            // 发送数据包
            if (send(sockfd, packet, packet_size, 0) < 0) {
                perror("ERROR sending packet");
                exit(1);
            }

            printf("Sent packet %d\n", next_seq_num);

            next_seq_num++;
        }

        // 接收确认信息
        int ack = -1;
        if (recv(sockfd, &ack, sizeof(ack), 0) < 0) {
            perror("ERROR receiving ack");
            exit(1);
        }

        printf("Received ack %d\n", ack);

        // 移动窗口
        if (ack >= base) {
            base = ack + 1;
        }
    }

    // 关闭套接字
    close(sockfd);

    return 0;
}
```

接收方：

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

```

```
#define WINDOW_SIZE 10   // 滑动窗口大小
#define BUFFER_SIZE 1024 // 缓冲区大小

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <port number>\n", argv[0]);
        exit(1);
    }

    // 创建套接字
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("ERROR opening socket");
        exit(1);
    }

    // 设置本地地址
    struct sockaddr_in server_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(atoi(argv[1]));

    // 绑定套接字到本地地址
    if (bind(sockfd, (struct sockaddr *) &server_addr, sizeof(server_addr)) < 0) {
        perror("ERROR on binding");
        exit(1);
    }

    // 监听连接
    listen(sockfd, 5);

    // 接受连接
    struct sockaddr_in client_addr;
    socklen_t client_addr_len = sizeof(client_addr);
    int connfd = accept(sockfd, (struct sockaddr *) &client_addr,
&client_addr_len);
    if (connfd < 0) {
        perror("ERROR on accept");
        exit(1);
    }

    // 接收数据包，并发送确认信息
    int next_expected_seq_num = 0;
    while (1) {
        // 接收数据包
        char packet[BUFFER_SIZE];
        memset(packet, 0, sizeof(packet));
        int packet_size = recv(connfd, packet, BUFFER_SIZE, 0);
```

```
        if (packet_size < 0) {
            perror("ERROR receiving packet");
            exit(1);
        } else if (packet_size == 0) {
            printf("End of transmission\n");
            break;
        }

        printf("Received packet %d\n", next_expected_seq_num);

        // 发送确认信息
        int ack = next_expected_seq_num;
        if (send(connfd, &ack, sizeof(ack), 0) < 0) {
            perror("ERROR sending ack");
            exit(1);
        }

        printf("Sent ack %d\n", ack);

        // 移动窗口
        if (next_expected_seq_num == ack) {
            next_expected_seq_num++;
        }
    }

    // 关闭套接字
    close(connfd);
    close(sockfd);

    return 0;
}

```

2-16

```
#include <stdio.h>
#include <stdbool.h>

#define BITS_PER_BYTE 8
#define CODED_BITS_PER_BLOCK 66
#define DATA_BITS_PER_BLOCK 64

bool is_valid_64b66b(const unsigned char* coded_data) {
    int i, j, k;
    unsigned char control_bits[2];

    //检查无效的控制字符
```

```
    for (i = 0; i < CODED_BITS_PER_BLOCK; i += BITS_PER_BYTE) {
        if (coded_data[i / BITS_PER_BYTE] == 0x00 || coded_data[i / BITS_PER_BYTE] == 0xFC) {
            return false;
        }
    }

    //从编码数据中提取控制位
    control_bits[0] = coded_data[0] & 0x03;
    control_bits[1] = (coded_data[0] >> 2) & 0x03;

    //检查控制字符编码
    if (control_bits[0] == control_bits[1]) {
        return false;
    }

    //检查数据字符编码
    for (i = BITS_PER_BYTE; i < CODED_BITS_PER_BLOCK; i += BITS_PER_BYTE) {
        unsigned char data_byte = 0;

       //解码数据位
        for (j = i; j < i + (BITS_PER_BYTE - 2); j++) {
            data_byte <<= 1;
            data_byte |= (coded_data[j / BITS_PER_BYTE] >> (BITS_PER_BYTE - 1 - (j % BITS_PER_BYTE))) & 0x01;
        }

        //计算数据字节的差异
        int disparity = 0;
        for (k = 0; k < BITS_PER_BYTE; k++) {
            disparity += (data_byte >> k) & 0x01;
        }
        disparity = (disparity % 2 == 0) ? 1 : -1;

        //检查视差位
        if (((coded_data[i / BITS_PER_BYTE] >> (BITS_PER_BYTE - 2)) & 0x01) != (disparity == 1)) {
            return false;
        }
    }

    return true;
}
```

第二章第二次作业补交

2-5:

主要实现自学习算法

```c
#include <stdio.h>
#include <string.h>

#define MAX_PORTS 10
#define MAC_ADDR_LEN 6
#define MAX_FRAME_LEN 1522
#define MAX_FRAMES 100

struct ethernet_header {
    uint8_t dest_addr[MAC_ADDR_LEN];
    uint8_t src_addr[MAC_ADDR_LEN];
    uint16_t type;
};

struct ethernet_frame {
    struct ethernet_header header;
    uint8_t data[MAX_FRAME_LEN];
    uint16_t len;
};

struct switch_port {
    uint8_t mac_addr[MAC_ADDR_LEN];
    int is_connected;
};

struct ethernet_switch {
    struct switch_port ports[MAX_PORTS];
};

// 比较两个 MAC 地址是否相同
int mac_addr_cmp(uint8_t* addr1, uint8_t* addr2) {
    for (int i = 0; i < MAC_ADDR_LEN; i++) {
        if (addr1[i] != addr2[i]) {
            return 0;
        }
    }
    return 1;
}

// 查找 MAC 地址对应的端口编号
int find_port_by_mac(struct ethernet_switch* switch_, uint8_t* mac_addr) {
    for (int i = 0; i < MAX_PORTS; i++) {
```

```
        if (switch_->ports[i].is_connected && mac_addr_cmp(switch_->ports[i].mac_addr, mac_addr)) {
            return i;
        }
    }
    return -1;
}

void handle_frame(struct ethernet_switch* switch_, struct ethernet_frame* frame, int in_port) {
    // 记录源 MAC 地址对应的端口
    int src_port = find_port_by_mac(switch_, frame->header.src_addr);
    if (src_port == -1) {
        src_port = in_port;
        memcpy(switch_->ports[src_port].mac_addr, frame->header.src_addr, MAC_ADDR_LEN);
        switch_->ports[src_port].is_connected = 1;
        printf("Learned new MAC address on port %d\n", src_port);
    }

    // 转发帧
    int dest_port = find_port_by_mac(switch_, frame->header.dest_addr);
    if (dest_port != -1 && dest_port != in_port) {
        // 发送到目标端口
        printf("Forwarding frame to port %d\n", dest_port);
    } else {
        // 广播到所有端口
        for (int i = 0; i < MAX_PORTS; i++) {
            if (i != in_port && switch_->ports[i].is_connected) {
                printf("Broadcasting frame to port %d\n", i);
            }
        }
    }
}

```

2-10

计算传输时间：

传输时间 = 1GB / 1Gbps = 8 秒

计算双向流量总和：

单向流量 = 文件大小 / 传输时间 = 1GB / 8s = 125MB/s

双向流量总和 = 单向流量 * 2 = 250MB/s

2-13

应用层：微信消息
传输层：TCP报文段（包含源端口号、目的端口号、序号、确认号等信息）
网络层：IP数据报（包含源IP地址、目的IP地址、TTL等信息）
数据链路层：以太网帧（包含源MAC地址、目的MAC地址、帧类型、数据等信息）
物理层：光纤信号（包含光脉冲序列等信息）

2-15

1. 创建交换机表格，用于记录每个 MAC 地址所在的 VLAN 和端口。
2. 接收到帧后，判断该帧是否为 VLAN 标记帧。如果是，则获取 VLAN 标记信息，根据 VLAN 标记信息获取该帧对应的端口；如果不是，则按照常规的 MAC 地址查找对应的端口。
3. 转发该帧到对应端口上。

```
// 创建交换机表格，用于记录每个 MAC 地址所在的 VLAN 和端口
switchTable = createSwitchTable();

// 接收到帧
frame = receiveFrame();

// 判断是否是 VLAN 标记帧
if (isVLANFrame(frame)) {
    // 获取 VLAN 标记信息
    vlanTag = getVLANTag(frame);

    // 如果交换机没有记录该 VLAN 的信息，则添加一条记录
    if (!switchTable.contains(vlanTag)) {
        switchTable.add(vlanTag, vlanTag.defaultPort);
    }

    // 根据 VLAN 标记信息，获取该帧对应的端口
    port = switchTable.getPort(vlanTag, frame.sourceMAC);

    // 转发该帧到对应端口上
    forwardFrame(frame, port);
} else {
    // 如果不是 VLAN 标记帧，则按照常规的 MAC 地址查找对应的端口
    port = switchTable.getPort(frame.sourceMAC);

    // 转发该帧到对应端口上
    forwardFrame(frame, port);
}

```

第三章作业

3-2:

接收数据包：路由器通过网络接口不断接收来自不同网络的数据包。

分析数据包：路由器会对接收到的数据包进行解析，提取出目标 IP 地址以及其他必要的信息。

查找路由表：路由器会根据目标 IP 地址查找路由表，确定下一步应该将数据包转发到哪个网络接口。

转发数据包：路由器将数据包转发到下一个网络接口，并按照相应的协议重新封装数据包，然后通过新的网络接口将数据包发送到下一个网络节点。

更新路由表：路由器会不断更新路由表，以确保它可以正确地转发数据包到目标网络。

3-3

```cpp
#include "MatGraph.cpp"
#include "vector"

using namespace std;

void DispAllPath(int dist[], int path[],int s[], int v, int n){
    for (int i = 0; i < n; ++i)
        if (s[i] == 1 && i != v) {
            printf("从%d到%d的最短距离为%d，路径：", v, i, dist[i]);
            vector<int> apath;
            apath.push_back(i);
            int pre = path[i];
            while (pre != v) {
                apath.push_back(pre);
                pre = path[pre];
            }
            printf("%d", v);
            for (int j = apath.size() - 1; j >= 0; --j)
                printf("->[%d]", apath[j]);
            printf("\n");
        }
        else
            printf("%d到%d没有路径",v,i);
}
void dijkstra(MatGraph &g, int v) {
    int dist[MAXV];
    int path[MAXV];
    int s[MAXV];
    for (int i = 0; i < g.n; ++i) {
        dist[i] = g.edges[v][i];
        s[i] = 0;
        if (dist[i] != 0 && dist[i] < INF)
```

```
            path[i] = v;
        else
            path[i] = -1;
    }
    s[v] = 1;
    int mindis,u = -1;
    for (int i = 0; i < g.n - 1; ++i) { // 循环向s中添加n - 1个顶点
        mindis = INF;
        for (int j = 0; j < g.n; ++j)
            if (s[j] == 0 && dist[j] < mindis) {
                u = j;
                mindis = dist[j];
            }
        s[u] = 1;

        for (int j = 0; j < g.n; ++j)  // 修改不在s中顶点的距离
            if (s[j] == 0 && g.edges[u][j] < INF && dist[u] + g.edges[u][j] <
dist[j]) {
                dist[j] = dist[u] + g.edges[u][j];
                path[j] = u;
            }
    }
    DispAllPath(dist,path,s,v,g.n);
}

//main函数中构造图的邻接矩阵，调用上述算法
```

3-4

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

typedef uint16_t WORD;

WORD checksum(WORD *buf, int nwords) {
    unsigned long sum;
    for (sum = 0; nwords > 0; nwords--) {
        sum += *buf++;
        if (sum & 0x80000000) {
            sum = (sum & 0xFFFF) + (sum >> 16);
        }
    }
    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);
    return ~sum;
}
```

```

int main() {
    WORD ip_header[] = {
        0x4500, 0x0073, 0x0000, 0x4000, 0x4001, 0x0000, 0xC0A8, 0x0001,
        0xC0A8, 0x0002
    };
    int header_len = 20 / 2;
    ip_header[5] = 0;
    WORD checksum_value = checksum(ip_header, header_len);
    ip_header[5] = checksum_value;
    printf("Calculated checksum: 0x%04x\n", checksum_value);
    return 0;
}

```

3-5

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAX_NODES 100

typedef struct node {
    int id;
    int parent;
    int cost;
    bool visited;
} Node;

void init_node(Node *node, int id, int cost) {
    node->id = id;
    node->parent = -1;
    node->cost = cost;
    node->visited = false;
}

int choose_next_node(Node *nodes, int n) {
    int min_cost = INT_MAX;
    int next_node = -1;
    for (int i = 0; i < n; i++) {
        if (!nodes[i].visited && nodes[i].cost < min_cost) {
            min_cost = nodes[i].cost;
            next_node = i;
        }
    }
    return next_node;
```

```
}

void build_multicast_tree(int graph[MAX_NODES][MAX_NODES], int n, int root_id) {
    Node nodes[MAX_NODES];
    for (int i = 0; i < n; i++) {
        init_node(&nodes[i], i, INT_MAX);
    }
    nodes[root_id].cost = 0;
    for (int i = 0; i < n; i++) {
        int node_id = choose_next_node(nodes, n);
        nodes[node_id].visited = true;
        for (int j = 0; j < n; j++) {
            if (graph[node_id][j] > 0) {
                int new_cost = nodes[node_id].cost + graph[node_id][j];
                if (new_cost < nodes[j].cost) {
                    nodes[j].cost = new_cost;
                    nodes[j].parent = node_id;
                }
            }
        }
    }
    printf("Multicast Tree:\n");
    for (int i = 0; i < n; i++) {
        if (nodes[i].parent != -1) {
            printf("%d -> %d\n", nodes[i].parent, i);
        }
    }
}
```

3-6:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// 定义路由表项结构体
typedef struct {
    char dest[16];  // 目标网络地址
    char next[16];  // 下一跳地址
    int cost;       // 路径费用
    int updateTime; // 上次更新时间
} RouteEntry;

// 模拟获取延迟时间
int getDelay(char *ip) {
    // 省略代码，返回到目标 IP 的延迟时间
}

// 模拟发送 RIP 消息并接收路由表更新信息
```

```c
void sendRIP(RouteEntry *rt, int num) {
    // 省略代码，模拟发送和接收 RIP 消息
}

// 更新路由表
void updateRoute(RouteEntry *rt, int num, int updateTime) {
    int i, j;
    int changed = 0;  // 标识路由表是否有变化

    // 遍历所有路由表项
    for (i = 0; i < num; i++) {
        // 如果该路由表项已过期，则更新
        if (updateTime - rt[i].updateTime > 30) {
            rt[i].cost = 16;  // 设置为无穷大
            rt[i].updateTime = updateTime;
            changed = 1;
        }
        // 向相邻路由器发送 RIP 消息，获取到该路由的延迟
        int delay = getDelay(rt[i].dest);
        // 如果延迟小于当前费用，则更新路由表
        if (delay < rt[i].cost) {
            rt[i].cost = delay;
            strcpy(rt[i].next, "next_hop");
            rt[i].updateTime = updateTime;
            changed = 1;
        }
    }

    // 如果路由表有变化，则向相邻路由器发送路由表更新信息
    if (changed) {
        sendRIP(rt, num);
    }
}

```

3-7

```c
// 定义链路的数据结构
typedef struct {
    uint16_t link_id; // 链路 ID
    uint16_t bandwidth; // 可用带宽
} link_t;

// 定义路由器的数据结构
typedef struct {
    uint16_t router_id; // 路由器 ID
    link_t links[MAX_LINKS]; // 该路由器连接的链路
```

```
    int num_links; // 该路由器连接的链路数
    uint16_t distance[MAX_ROUTERS]; // 到其他路由器的距离
    uint16_t next_hop[MAX_ROUTERS]; // 到其他路由器的下一跳
    bool updated; // 路由表是否被更新
} router_t;

// 定义 OSPF 计算路由的函数
void ospf_calculate(router_t routers[], int num_routers) {
    bool updated = true;
    while (updated) {
        updated = false;
        for (int i = 0; i < num_routers; i++) {
            router_t *router = &routers[i];
            for (int j = 0; j < router->num_links; j++) {
                link_t *link = &router->links[j];
                uint16_t neighbor_id = link->link_id;
                router_t *neighbor = NULL;
                for (int k = 0; k < num_routers; k++) {
                    if (routers[k].router_id == neighbor_id) {
                        neighbor = &routers[k];
                        break;
                    }
                }
                if (!neighbor) {
                    continue;
                }
                uint16_t bandwidth = link->bandwidth;
                for (int k = 0; k < num_routers; k++) {
                    uint16_t distance_through_neighbor = router-
>distance[neighbor_id] + neighbor->distance[k];
                    if (distance_through_neighbor < router->distance[k]) {
                        router->distance[k] = distance_through_neighbor;
                        router->next_hop[k] = neighbor_id;
                        updated = true;
                    }
                    if (distance_through_neighbor < neighbor->distance[k]) {
                        neighbor->distance[k] = distance_through_neighbor;
                        neighbor->next_hop[k] = router->router_id;
                        updated = true;
                    }
                }
            }
        }
    }
}

```

3-8:

1. 节点在收到要广播的消息时，生成一个消息序列号（Message Sequence Number，MSN）并将其和消息一起发送出去。
2. 接收到广播消息的节点将消息的序列号存储在一个表中，并转发广播消息。
3. 当节点广播消息时，它会记录下自己的状态，以便后续可能的重传操作。如果一个节点接收到了同一个序列号的多个副本，则只转发其中的一个。
4. 每个节点维护一个邻居表，包含每个邻居的地址、MSN值和最后一次收到广播消息的时间。
5. 每个节点还维护一个可靠性表，其中包含每个邻居的地址、MSN值和已经确认收到的最后一条广播消息的MSN值。如果一个节点发现自己的可靠性表中有一个邻居的MSN值比自己的表中的MSN值更大，那么它就会请求该邻居重新发送丢失的消息。
6. 当一个节点需要发送广播消息时，它会将消息和当前的MSN值发送给它的所有邻居，并将该值加1。如果一个节点在一段时间内没有收到来自某个邻居的反馈，则它将请求该邻居重新发送该消息。

   伪代码：

```
// 初始化
for each neighbor n
    reliability[n] = 0
    neighborTable[n] = (0, now())

// 发送广播消息
broadcast(msg):
    msn = msn + 1
    for each neighbor n
        send(n, msg, msn)
    reliability[myself] = msn

// 接收广播消息
receive(n, msg, msn):
    if msn > neighborTable[n].msn:
        neighborTable[n].msn = msn
        neighborTable[n].time = now()
        if msg not in messageTable:
            messageTable.add(msg)
            for each neighbor m
                if m != n:
                    send(m, msg, msn)
    else if msn == neighborTable[n].msn and msg not in messageTable:
        messageTable.add(msg)
        for each neighbor m
            if m != n:
                send(m, msg, msn)

// 定时器超时
timeout(n):
    if now() - neighborTable[n].time > timeout:
        for each message m in messageTable
```

```
            if reliability[n] < msn
                send(n, m, reliability[n])

```

3-13:

对于每个到达的数据包，遍历路由表中的每个路由项，比较数据包的目的 IP 地址与路由项的网络前缀是否匹配。如果匹配，就将数据包转发到对应的出接口。如果没有匹配项，就丢弃该数据包。

```
for each incoming packet p:
    dest_ip = p.destination_ip
    match = false
    for each route r in routing_table:
        if dest_ip AND r.prefix_mask == r.network:

            match = true
            break
    if not match:
        drop p
    else:
        send p to r.outgoing_interface

```

为了支持路由聚合，可以将多个子网的路由信息合并成一个超级前缀。这样做可以减小路由表的规模，提高路由选择的效率。下面是一个简单的路由聚合算法的伪代码：

```
routes = sorted(routing_table, key=lambda r: r.prefix_mask, reverse=True)
while len(routes) > 1:
    r1 = routes.pop(0)
    r2 = routes.pop(0)
    if r1.network == r2.network:
        r = Route(r1.network, min(r1.prefix_mask, r2.prefix_mask),
r1.outgoing_interface)
        routes.append(r)
    else:
        routes.append(r2)
        routes.sort(key=lambda r: r.prefix_mask, reverse=True)


```

3.14:

1. 初始化 NAT 路由表，记录每个私有 IP 地址和对应的公共 IP 地址，以及私有 IP 地址和对应的 MAC 地址；
2. 当 NAT 路由器收到从私有 IP 地址发送来的数据包时，先检查 NAT 路由表中是否存在该私有 IP 地址对应的公共 IP 地址；
3. 如果存在对应关系，则将数据包的源 IP 地址改为对应的公共 IP 地址，并更新数据包的校验和，然后转发数据包；

4. 如果不存在对应关系，则为该私有 IP 地址生成一个新的公共 IP 地址，同时记录新的映射关系，并将数据包的源 IP 地址改为新的公共 IP 地址，更新数据包的校验和，然后转发数据包；
5. 当 NAT 路由器收到从公共 IP 地址发送来的数据包时，先检查数据包的目的 IP 地址是否是 NAT 路由器所记录的任意一个公共 IP 地址；
6. 如果目的 IP 地址是 NAT 路由器所记录的一个公共 IP 地址，则将数据包的目的 IP 地址改为对应的私有 IP 地址，并更新数据包的校验和，然后转发数据包；
7. 如果目的 IP 地址不是 NAT 路由器所记录的任意一个公共 IP 地址，则丢弃数据包。