数据链路层滑动窗口协议的设计与实现

谢牧航 2022211363　　班级 2022211301　　指导老师 高占春

目录

第一章　实验内容和实验环境描述 2
1.1　实验任务 2
1.2　实验内容 2
1.3　实验环境 2
第二章　软件设计 3
2.1　回退 N 步协议 3
2.1.1　数据结构和模块结构 3
2.1.2　算法流程图 4
2.2　选择重传协议 5
2.2.1　数据结构和模块结构 5
2.2.2　算法流程图 7
第三章　实验结果分析 8
3.1　无差错传输 8
3.1.1　回退 N 步协议 8
3.1.2　选择重传协议 8
3.2　健壮性 9
3.3　协议参数的选取 9

第一章 实验内容和实验环境描述

在这个实验中, 目标是实现和测试一个基于滑动窗口协议的可靠数据传输协议。这种类型的协议广泛用于计算机网络中, 以确保数据包能可靠地在发送方和接收方之间传输, 即使在不稳定的网络环境中也能正确处理数据包的丢失、错误和顺序问题。

1.1 实验任务

1. **协议实现**：编写代码来实现滑动窗口协议，包括数据帧、确认帧（ACK）和否认帧（NAK）的处理。
2. **错误处理**：实现 CRC 检验来检测数据在传输过程中的错误，并通过发送 NAK 请求重传。
3. **定时器管理**: 为数据帧设置超时定时器, 如果在指定时间内未收到 ACK, 则需要重传数据帧。
4. **窗口管理**：管理滑动窗口的大小，控制发送方的数据流，以防止接收方被数据淹没。

1.2 实验内容

本次实验选用的滑动窗口协议为回退 N 步和选择重传协议, 利用所学数据链路层原理, 自行设计一个滑动窗口协议, 在仿真环境下编程实现有噪音信道环境下两站点之间无差错双工通信。信道模型为 8000bps 全双工卫星信道，信道传播时延 270 毫秒，信道误码率为 10^{-5}，信道提供帧传输服务，网络层分组长度固定为 256 字节。

本次实验选用的滑动窗口协议为回退 N 步和选择重传协议，并且使用了 NAK 通知机制。

1.3 实验环境

- **开发工具**：使用 C 语言在 Linux（WSL）上进行开发。使用编辑器 VSCode 进行编写代码。
- **编译环境**：GCC 11.2.0; WSL Ubuntu 22.04 LTS.

第二章 软件设计

2.1 回退 N 步协议

2.1.1 数据结构和模块结构

2.1.1.1 结构体 FRAME

这个结构体是核心数据结构，用于存储单个帧的数据。

```
typedef struct {
    unsigned char kind;  // 帧的类型（数据帧、确认帧或否认帧）
    unsigned char ack;   // 确认号，指示发送方期望接收的下一个帧的序列号
    unsigned char seq;   // 序列号，当前帧的序列号
    unsigned char data[PKT_LEN];  // 存储实际传输的数据
    unsigned int crc;    // 循环冗余校验值，用于错误检测
} FRAME;
```

2.1.1.2 全局变量

- **phl_ready** (`static int phl_ready = 0;`)
  - 表示物理层是否准备好接收数据。当物理层准备就绪，可以开始发送数据帧。
- **no_nak** (`static int no_nak = 1;`)
  - 用于控制否认帧（NAK）的发送。为避免重复发送 NAK，这个标志指示是否已发送过 NAK。

2.1.1.3 主函数中的变量

- **ack_expected** (`unsigned char ack_expected = 0;`)
  - 指示发送方期待的下一个确认号，也即已发送但未被确认的最早帧的序列号。
- **next_frame_to_send** (`unsigned char next_frame_to_send = 0;`)
  - 指示下一个要发送的帧的序列号。
- **frame_expected** (`unsigned char frame_expected = 0;`)
  - 指示接收方期待的下一个帧的序列号。
- **buffer** (`unsigned char buffer[MAX_SEQ + 1][PKT_LEN];`)
  - 缓冲区数组，存储待发送的数据，按序列号索引。
- **nbuffered** (`unsigned int nbuffered = 0;`)
  - 当前缓冲区中已缓存但未确认的帧数量。
- **r** (`FRAME r;`)
  - 用于接收数据的帧结构体实例。
- **event, arg** (`int event, arg;`)
  - `event` 用于接收从 `wait_for_event()` 函数返回的事件类型。
  - `arg` 可用于传递事件特定的参数（如超时事件的帧序列号）。

2.1.1.4 模块之间的调用关系和功能

1. **发送和接收处理（`send_data`, `put_frame` 函数）**
   - `send_data` 根据帧类型组装帧，并通过 `put_frame` 发送。
   - `put_frame` 计算 CRC，并调用底层的 `send_frame` 将帧发送到物理层。
2. **超时管理（`start_timer`, `stop_timer`）**
   - 根据帧序列号启动和停止计时器，用于管理数据帧的超时重传。
3. **确认管理（`stop_ack_timer`, `start_ack_timer`）**
   - 控制确认帧的发送时机，避免过于频繁的 ACK 发送。
4. **网络层管理（`enable_network_layer`, `disable_network_layer`）**
   - 根据当前网络状态和缓冲区情况控制网络层的数据发送能力，以防止发送方溢出。
5. **事件驱动处理（主循环）**
   - 主循环等待事件，根据事件类型（如网络层就绪、数据接收、超时）调用相应的处理函数。

2.1.2 算法流程图

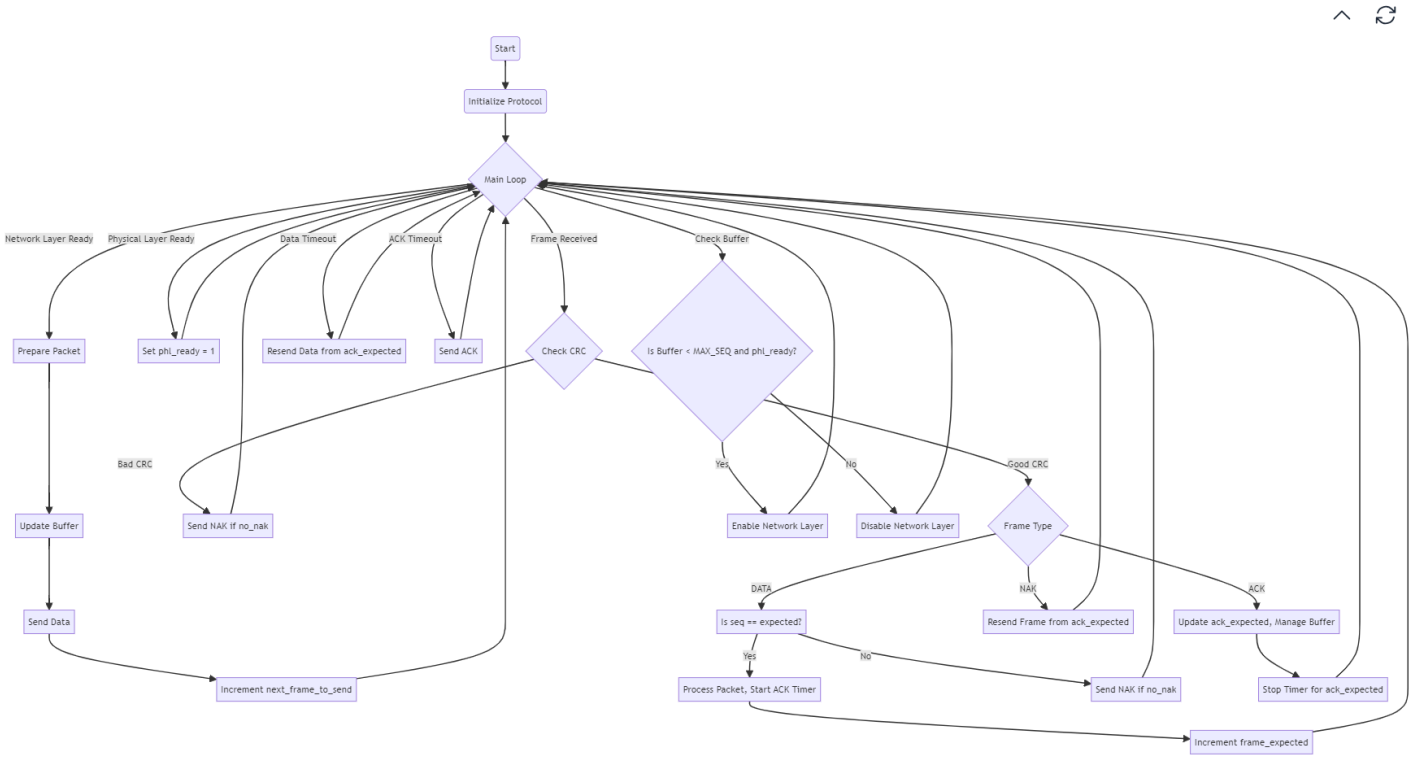

图 1　GBN 协议算法流程图

2.2 选择重传协议

2.2.1 数据结构和模块结构

2.2.1.1 数据结构

2.2.1.1.1 结构体 FRAME

这个结构体用于存储单个数据帧的详细信息。

```
typedef struct {
    unsigned char kind;  // 帧类型（数据、确认、否认）
    unsigned char ack;   // 确认号
    unsigned char seq;   // 序列号
    unsigned char data[PKT_LEN];  // 数据载荷
    unsigned int crc;    // CRC 校验码
} FRAME;
```

2.2.1.2 全局变量

- **phl_ready** (`static int phl_ready = 0;`)
  - 指示物理层是否准备好发送数据。此变量为 1 时，可以开始发送数据帧。
- **no_nak** (`static int no_nak = 1;`)
  - 防止对同一错误帧多次发送否认（NAK）。此变量为 1 时，表示可以发送 NAK。

2.2.1.3 主函数中的变量

- **ack_expected** (`unsigned char ack_expected = 0;`)
  - 表示发送方期待的下一个确认帧的序列号。
- **next_frame_to_send** (`unsigned char next_frame_to_send = 0;`)
  - 表示发送方下一个要发送的数据帧的序列号。
- **frame_expected** (`unsigned char frame_expected = 0;`)
  - 表示接收方期待接收的下一个数据帧的序列号。
- **buffer** (`unsigned char buffer[MAX_SEQ + 1][PKT_LEN];`)
  - 存储待发送数据的缓冲区。
- **nbuffered** (`unsigned int nbuffered = 0;`)
  - 缓存中的帧数量，即已发送但尚未确认的帧的数量。
- **r** (`FRAME r;`)
  - 接收数据的帧结构体实例。
- **event, arg** (`int event, arg;`)
  - 用于接收事件类型和相应参数的变量，由 `wait_for_event()`函数提供。

2.2.1.4 模块之间的调用关系和功能

1. **主控制逻辑 (main function)**

- 主函数设置初始状态并进入一个无限循环，持续等待和响应不同的网络事件。
- 控制流转移给不同的处理函数, 根据网络、物理层事件或数据超时事件来处理发送和接收数据的逻辑。

2. **协议初始化 (`protocol_init`)**
   - 初始化网络协议和环境设置。
   - 被主函数调用来准备所有必要的环境配置。
3. **数据包处理 (`send_data`)**
   - 根据事件和当前状态处理并发送数据包。
   - 构建帧（包括数据帧、确认帧和否认帧），并调用 `put_frame` 来发送。
   - 由主控制逻辑在适当的事件（如网络层就绪）时调用。
4. **帧发送 (`put_frame`)**
   - 计算帧的 CRC，并将帧发送到物理层。
   - 被 `send_data` 函数调用，用于实际的帧发送操作。
5. **事件等待 (`wait_for_event`)**
   - 等待并返回下一个发生的事件, 可能是网络层就绪、物理层就绪、帧接收、数据超时或确认超时。
   - 主循环中调用，用于决定下一步的操作。
6. **数据接收和处理**
   - 接收帧并根据帧的类型和内容处理帧。
   - 主要包括验证 CRC，处理接收到的数据帧、确认帧和否认帧。
   - 在主循环中，根据接收事件调用，由 `recv_frame` 函数支持。
7. **定时器控制 (`start_timer`, `stop_timer`, `start_ack_timer`, `stop_ack_timer`)**
   - 管理数据帧和确认帧的发送超时。
   - 在数据帧发送后或接收帧后调用，用于确保超时后可以重新发送帧或发送确认帧。
8. **网络层控制 (`enable_network_layer`, `disable_network_layer`)**
   - 根据当前的缓冲状态和物理层就绪状态，启用或禁用网络层的数据发送。
   - 用于控制数据流，防止发送方的数据缓冲区溢出。
9. **日志和调试 (`dbg_frame`, `dbg_event`, `dbg_warning`)**
   - 输出调试信息，帮助跟踪程序的执行和数据处理的状态。
   - 在各个关键的数据处理点调用。
10. **初始化**：设置协议、日志输出和禁用网络层。
11. **主循环**：持续等待事件，并根据事件类型处理相应逻辑。
    - **网络层就绪**：准备数据包发送，并更新缓冲区。

- **物理层就绪**：设置物理层准备就绪标志。
- **接收帧**：接收帧并根据 CRC 检验处理帧。
  - 若 CRC 校验失败且未发送 NAK，则发送 NAK。
  - 若接收到数据帧，并且序列号为期待的序列号，处理数据并向上层传递，更新期待的序列号，启动 ACK 定时器。
  - 若接收到意外的数据帧，并且 `no_nak` 为真，发送 NAK。
  - 对于确认帧，更新期望的确认序列号，并处理缓冲区。
  - 对于否认帧，若确认号加一等于期望的确认号，重发相应帧。
- **数据超时**：重发从 `ack_expected` 开始的所有帧。
- **确认超时**：发送当前期望的确认帧。

12. **网络层控制**：根据缓冲区的状态和物理层的就绪状态启用或禁用网络层。

2.2.2 算法流程图

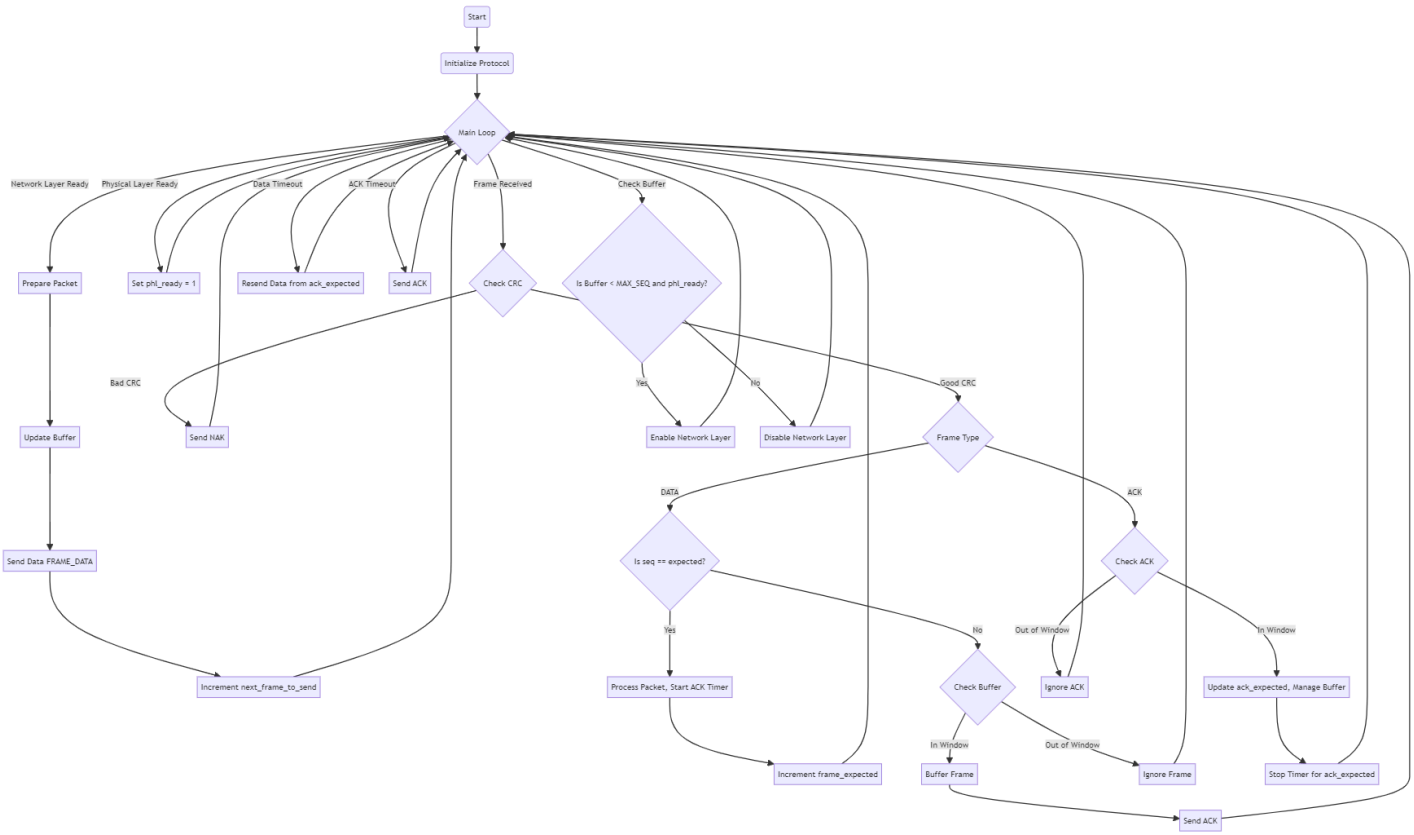

图 2　SR 协议算法流程图

第三章 实验结果分析

3.1 无差错传输

3.1.1 回退 N 步协议

在有误码信道环境下，无差错传输是关键挑战之一，涉及到数据的正确接收，即便在存在干扰和误码的情况下。回退 N 步（Go-Back-N, GBN）和选择重传（Selective Repeat, SR）协议都采用了不同的机制来确保数据的正确性和完整性。现在，我们来具体分析你提供的两段代码，以判断这些协议是否实现了有误码信道环境中无差错传输功能。

在提供的 GBN 协议代码中，关键的错误控制机制包括：

1. **CRC 校验**：代码中使用 `crc32()`函数来生成和验证每个帧的 CRC 校验码，这是检测传输过程中数据帧是否出现错误的重要手段。如果接收到的帧的 CRC 校验失败，该帧及其后的所有帧将被重新请求和重传。

```
if (len < 5 || crc32((unsigned char *)&r, len) != 0) {
    dbg_event("**** Receiver Error, Bad CRC Checksum\n");
    // 重传机制触发
}
```

2. **超时重传**：设置了定时器，用于每个数据帧设置超时时间。如果在指定时间内未收到确认，发送方将重传该帧及其后的所有帧。

```
case DATA_TIMEOUT:
    dbg_event("---- DATA %d timeout\n", arg);
    // 重传从 ack_expected 开始的所有帧
```

3.1.2 选择重传协议

在 SR 协议的代码中，为了处理有误码信道环境下的传输，实现了以下机制：

1. **CRC 校验**：同样采用 CRC 校验来确保数据帧的正确性。接收方在 CRC 校验失败时发送 NAK，请求发送方重新发送特定的帧，而不是所有后续帧。

```
if (len < 5 || crc32((unsigned char *)&r, len) != 0) {
    dbg_event("**** Receiver Error, Bad CRC Checksum\n");
    if (no_nak)
        send_data(FRAME_NAK, 0, frame_expected, buffer);
}
```

2. **独立帧重传**：与 GBN 协议不同，SR 协议允许接收方独立确认每个帧，并且只请求重传出现错误的帧。

```
if (r.kind == FRAME_NAK && (r.ack + 1) % (MAX_SEQ + 1) == ack_expected) {
    dbg_frame("Recv NAK  %d\n",  (r.ack + 1) % (MAX_SEQ + 1));
    // 重传特定的帧
}
```

两个协议都实现了有误码信道环境中无差错传输的功能。GBN 协议通过 CRC 校验和超时重传机制来确保数据完整性，适用于错误较少的环境。而 SR 协议在处理更频繁错误和复杂网络环境中显示出更高的效率，通过 CRC 校验和选择性重传机制来减少不必要的重传，从而提高传输效率。

这些特性使得两种协议都能有效地在有误码的信道环境中传输数据，尽管它们在错误恢复策略和效率方面有所不同。

3.2 健壮性

经实际测试，两种协议代码在各种参数的环境下都能稳定运行超过 20 分钟以上。

3.3 协议参数的选取

在实验中，我选择了以下参数:

- **最大帧序号**：63

我们可以知道最大帧长度为 $3 + 256 + 4 = 263$ 字节。传送帧的时间为 $\frac{263}{1000bps \times 8 = 263ms}$。根据捎带确认 $\frac{w}{2+2a}$，其中 w 为窗口大小，a 为 $\frac{270ms}{263ms}$。计算得到 $w = 4$ 的时候效率已经达到 100%。由于程序中所给计时器的范围为 0 ~ 63，我们尽量选择了最大的序号范围。

- **重传数据帧时限**：2000ms

不考虑 CPU 处理时间，数据帧传送并被捎带确认的时间为 $2 \times (263ms + 270ms) = 1066ms$。为了给 CPU 处理时间留出余量，我选择了 2000ms 作为重传数据帧的时限。

- **确认帧时限**：1200ms

ACK 帧的大小为 6 字节，所以发送时间为 6ms。