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

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

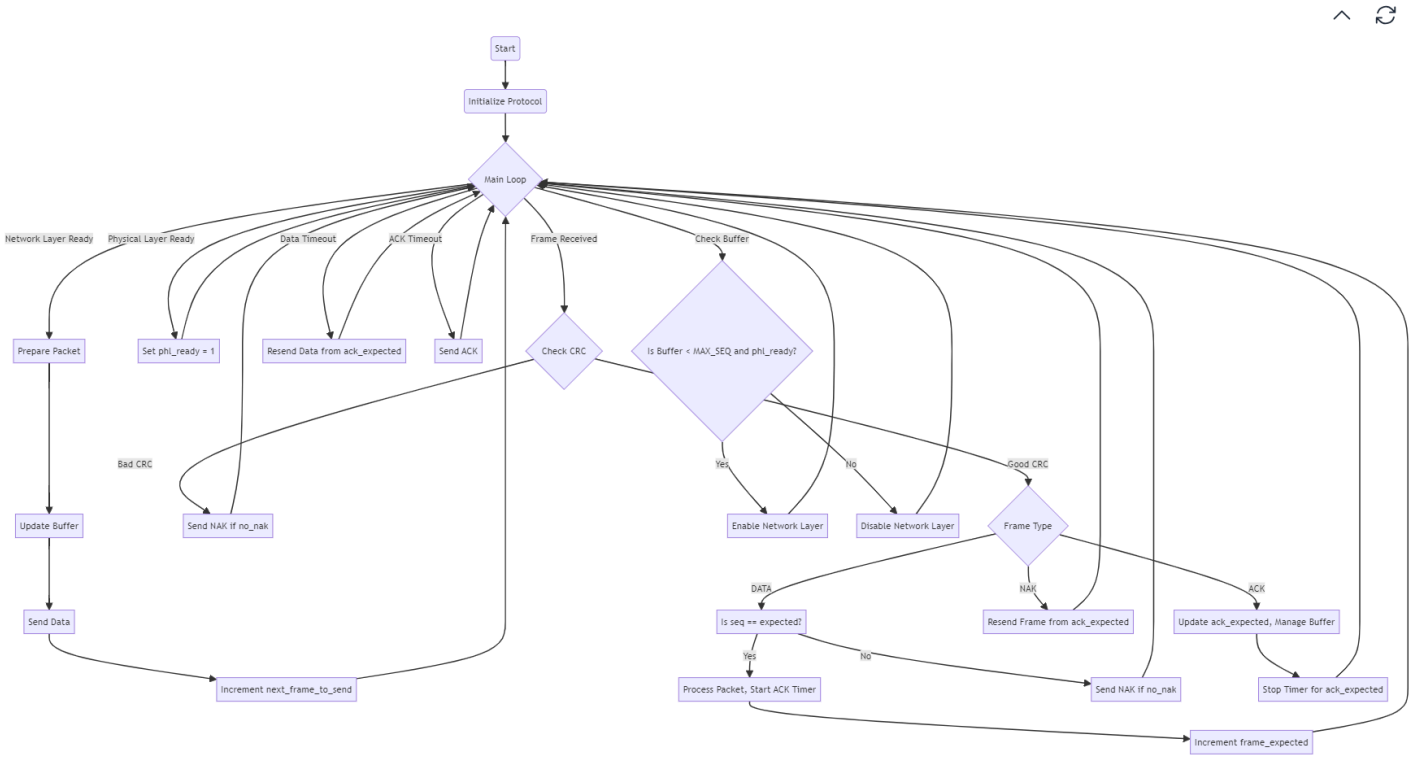

图 1　GBN 协议算法流程图

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

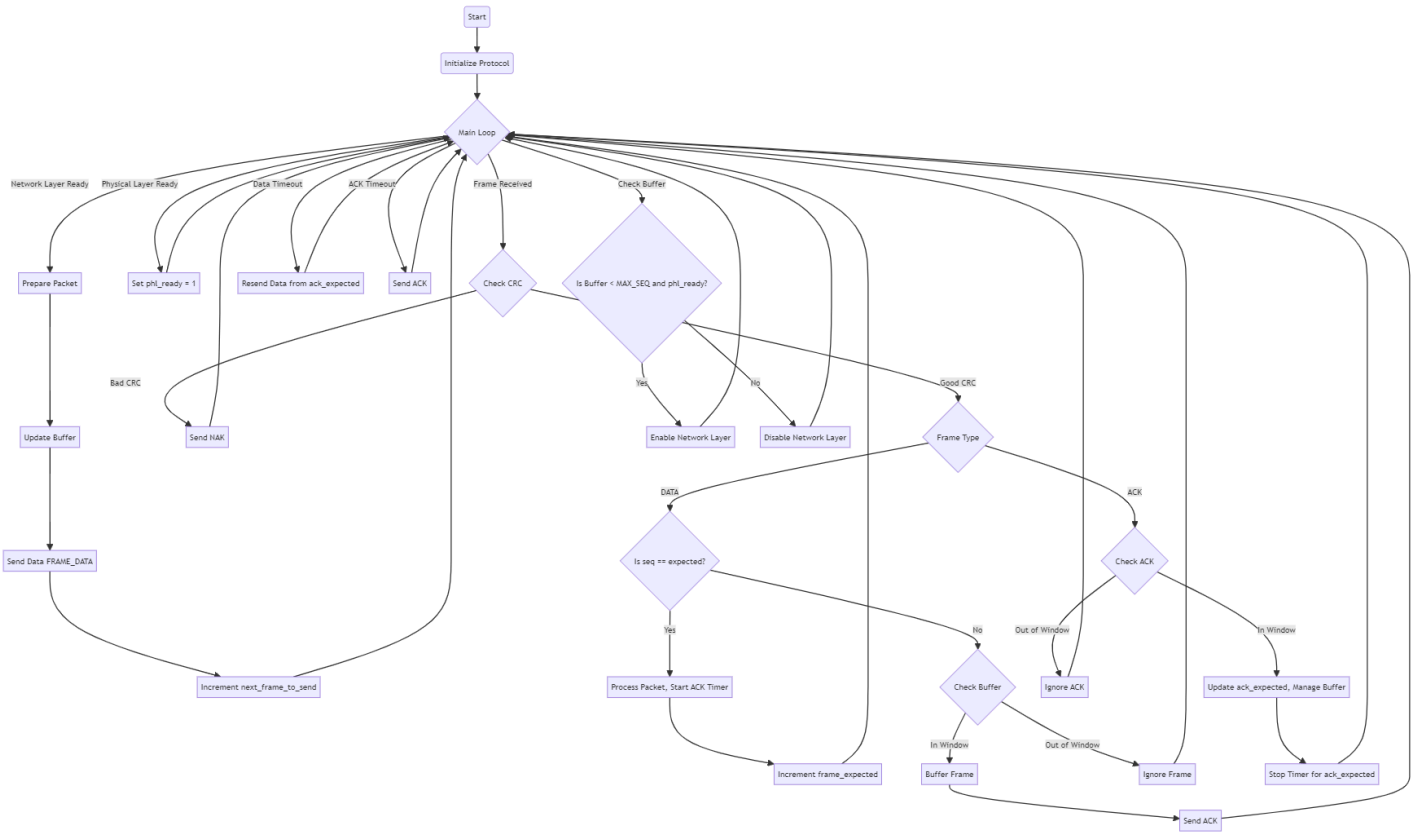

图 2　SR 协议算法流程图

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 $\frac{263}{1000bps\times 8=263ms}$。在无误码的情况下，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 $w = 5$ 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。

- **确认帧时限**：800ms

  我们为了实现捎带确认，至少需要等待一个数据包成帧的过程。即 263ms。为了保证数据帧传送的可靠性，我们选择了 800ms 作为确认帧的时限。

- **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 $2 \times (263ms + 270ms) + ACK\_TIMER = 1866ms$。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。

3.4 理论分析

在无误码的情况下，信道利用率 $\eta = \frac{w}{2+2a}$。本程序中，$w = 63$，$a = \frac{270ms}{263ms}$，所以 η 已经达到 100% 的效率。在有误码的情况下，以 $ber = 10^{-4}$ 为例，成功发出一帧的概率为 $P = (1 - 10^{-4})^{8\times 263} = 81.025\%$。所以 $\eta = P\eta = 81.025\%$。

3.5 实验结果分析

3.5.1 性能测试结果

序号	说明	运行时间(秒)	GBN A 接收方信道利用率	GBN B 接收方信道利用率	SR A 接收方信道利用率	SR B 接收方信道利用率
1	无误码信道数据传输	1200	51.59%	96.97%	54.23%	96.97%
2	站点 A 分组层平缓方式发出数据, 站点 B 周期性交替“发送 100 秒, 慢发 100 秒”	1200	44.61%	82.62%	50.38%	92.21%
3	无误码信道, 站点 A 和站点 B 的分组层都洪水式产生分组	1200	96.97%	96.97%	96.96%	96.96%
4	站点 A/B 的分组层都洪水式产生分组	1200	84.87%	83.05%	93.36%	92.98%
5	站点 A/B 的分组层都洪水式产生分组，线路误码率设为10^{-4}	1200	24.09%	22.12%	61.05%	60.41%

从表格数据可以看出:

1. **站点 A 平缓发出数据，站点 B 周期性交替发送与慢发**：
   - 利用率相比洪水式发送有所下降。这是因为站点 B 发送速率的周期性变化使得难以进行捎带确认。
2. **无误码信道，站点 A 和 B 都洪水式产生分组**：
   - 接收方信道利用率非常高, 几乎达到了极限，显示在高数据负载下协议能有效处理传入数据。
3. **站点 A/B 的分组层都洪水式产生分组，线路误码率设为**10^{-4}：
   - 误码率的增加显著降低了信道利用率。这表明错误恢复机制在处理高误码环境时效率不高, 可能导致大量重传和错误处理开销。同时我们可以看到，在高误码率情况下，SR 协议的性能相对更好。

3.5.2 性能差距及原因

本协议距离示例数据仍有较大距离, 原因可能很多, 如部分算法性能较低、参数设置不合理、测试环境不同等。在实际应用中，需要根据具体情况进行调整和优化。

3.5.3 改进方案

1. **优化重传机制**：

- 实施更智能的重传策略，如基于接收方反馈的更精确错误信息进行选择性重传，而非简单的超时重传。

2. **动态调整窗口大小**：
   - 根据网络状况动态调整发送窗口大小，以适应不同的网络拥塞和误码率。
3. **改进误码处理**：
   - 引入更高效的编码技术，如前向纠错（FEC），以减少需要重传的数据量。
4. **监控和调整反馈策略**：
   - 实施更有效的反馈策略，优化 ACK 包的发送频率和内容，减少确认带来的额外负载。

3.6 存在的问题

根据测试结果，尽管程序没有明显的失败，但在高误码率场景下性能显著下降。这表明当前的错误处理机制可能不够健壮，需要进一步优化来提高恢复效率。此外，对于周期性变化的带宽处理策略可能还不够成熟，需要更细致的流量控制和调度算法来适应这种变化。

通过上述分析和建议的改进措施，可以进一步提升协议的效率和鲁棒性，使其更好地适应复杂多变的网络环境。

第四章 研究和探索的问题

4.1 CRC 校验能力

假设本次实验中所设计的协议用于建设一个通信系统。这种“在有误码的信道上实现无差错传输”的功能听起来很不错，但是后来该客户听说 CRC 校验理论上不可能 100% 检出所有错误。这的确是事实。你怎样说服他相信你的系统能够实现无差错传输? 如果传输一个分组途中出错却不能被接收端发现，算作一次分组层误码。该客户使用本次实验描述的信道，客户的通信系统每天的使用率 50%，即：每天只 有一半的时间在传输数据，那么，根据你对 CRC32 检错能力的理解，发生一次分组层误码事件，平均需要多少年? 从因特网或其他参考书查找相关材料，看看 CRC32 有没有充分考虑线路误码的概率模型，实际校验能力到底怎样。你的推算是过于保守了还是夸大了实际性能? 如果你给客户的回答不能让他满意，这种分组层误码率，你还有什么措施降低发生分组层误码事件的概率，这些措施需要什么代价?

CRC（循环冗余检验）是一种常用的错误检测技术，尤其是在数据通信领域。CRC32，特别是，是一种通过计算数据流的 32 位多项式表示来生成校验码的方法。理解和解释 CRC32 的能力和局限性对于说服客户信任通信系统的可靠性至关重要。

4.1.1CRC32 的错误检测能力

本次实验中 CRC32 校验和的计算直接调用了一个简单的库函数，8.8 节中的库函数 crc32() 是从 RFC1662 中复制并修改而来。在 PPP 相关协议文本 RFC1662.TXT（以另外单独一个文件提供）中含有计算 CRC-32 和 CRC-16 的源代码，浏览这些源代码。教材中给出了手工计算 CRC 校验和的方法，通过二 进制“模 2”除法求余数。这些源代码中却采用了以字节值查表并叠加的方案，看起来计算速度很快。你能分析出这些算法与我们课后习题中手工进行二进制“模 2”除法求余数的算法是等效的吗？算法中设置的查表数组 crc_table[256]数组是怎样构造出来的? 你能否写一段 C 语言程序，按照模 2 除法的规则， 用你的程序生成速查表 crc_table 的 256 个数字？或者，给出一个例子手工验证表中的某项是怎样得来的。在 x86 系列计算机上为某一帧计算 32 位 CRC32 校验和，比较一下：为相同帧计算 16 位 CRC16 校 验和，所花费的 CPU 时间会多一倍吗？本次实验课提供的 crc32 函数只有两个参数，分别为缓冲区首地 址和缓冲区长度，这似乎足够了，可是，RFC1662 给出的源代码样例中的函数 pppfcs32(fcs, cp, len) 需要三个参数，为什么要这样设计?

1. **错误检测率**：CRC32 能够检测任何单一错误、任何两个错误、任何奇数个错误、任何错位的双错误（即两个错误相隔 1-31 位）以及 32 位长度的错误模式。此外，对于长度小于等于 32 位的错误，CRC32 有很高的错误检测能力。

2. **误漏概率**：尽管 CRC32 非常有效，但它无法保证 100% 的错误检测率，尤其是对于更复杂的多错误模式和某些特定的错误分布，如错误数据的 CRC 校验和仍未 0 的情况。然而，未被检出的错误配置的概率非常低，通常在 2^{-32} 左右。

4.1.2 估算分组层误码事件的发生频率

假设：

- 数据传输的错误率为 p（比如 10^{-6}）
- 每天传输的数据量为 X 字节（假设每天传输 1TB，即 10^{12} 字节）
- 使用率为 50%，即每天传输数据量为 0.5TB

考虑到 CRC32 的检测能力，发生一次分组层误码事件的平均概率 q 可以估计为 2^{-32}。那么，每天未被检出的错误的期望次数 E 可以计算为：

$$E = X \times p \times q$$

如果每天传输 5×10^{11} 字节，且 $p = 10^{-6}$， 则 $E = 5 \times 10^{11} \times 10^{-6} \times 2^{-32} \approx 1.164 \times 10^{-9}$ 这意味着发生一次分组层误码的期望时间将非常长，可能需要数百万年。

4.1.3 提高系统可靠性的措施

如果客户对 CRC32 的保护水平仍有疑虑，可以采取以下措施进一步降低错误率：

1. **使用更强的错误检测代码**：例如 CRC64 或更复杂的纠错编码，如汉明码或里德-所罗门编码，这些能提供错误纠正功能。
2. **增加校验级别**：通过增加数据包中的冗余信息，比如多重 CRC 校验，或结合使用校验和和 CRC。
3. **改进物理层技术**：通过使用更高质量的传输介质或改善信号处理技术，比如使用更先进的调制解调器，可以直接减少误码率。
4. **引入重传机制**：在检测到错误时自动重传数据包，虽然会增加带宽和延迟，但可以显著提高数据传输的可靠性。

4.1.4 结论

虽然理论上 CRC32 并不保证完美无误，但在实际应用中，其检测错误的能力足够强大，能够满足大多数通信需求。通过适当的设计和额外的安全措施，可以进一步降低错误的风险，满足更严格的可靠性要求。

4.2 CRC 校验和的计算方法

4.2.1CRC 算法与模 2 除法等效性

CRC 校验是基于生成多项式和模 2 除法的。在本实验中，这个多项式是 $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$。这个多项式决定了在数据帧后附加的 CRC 校验序列。

生成 CRC 时，数据帧（视为一个大的二进制数）被左移 32 位（对于 CRC32），然后用上述多项式的二进制表示进行模 2 除法。最终的余数就是 CRC 值。在硬件中，这通常通过移位寄存器和一些 XOR 门实现，而在软件中可以通过查表方法实现，这提高了计算的速度。

4.2.2 查表法的生成

查表法是一种优化技术，用于加速CRC的计算。每个可能的字节值都预先计算出一个CRC，并存储在一个表中（通常称为CRC表）。这样，在计算CRC时，只需简单地查找每个字节的CRC并进行组合，而不是每次都重新计算。

表的生成如下：对于256个可能的字节值，使用CRC的生成多项式计算每个值经过CRC计算后的结果。这个过程只需进行一次，然后在任何需要 CRC 校验的实现中复用这个表。

4.2.3C 语言实现示例

下面是一个示例 C 代码，展示如何生成一个 CRC32 的查找表：

```c
#include <stdio.h>
#define POLY 0xEDB88320  // CRC-32 的多项式

int main() {
    unsigned long crc_table[256];
    unsigned long crc;
    for (int i = 0; i < 256; i++) {
        crc = i;
        for (int j = 8; j > 0; j--) {
            if (crc & 1)
                crc = (crc >> 1) ^ POLY;
            else
                crc >>= 1;
        }
        crc_table[i] = crc;
    }

    // 打印生成的 CRC 表
    printf("unsigned long crc_table[256] = {\n");
    for (int i = 0; i < 256; i++) {
        printf("0x%08lX, ", crc_table[i]);
        if ((i + 1) % 4 == 0) printf("\n");
    }
    printf("};\n");
    return 0;
}
```

4.2.4CRC32 与 CRC16 性能比较

在理论上，CRC32 比 CRC16 的计算更耗时，因为它涉及更长的位模式。然而，如果使用查表法，这两者的计算时间差异可能并不显著，因为主要耗时在于表查找操作，而表查找的时间复杂度为 O(1)。实际的性能差异会依赖于具体实现和运行时的硬件。在实测中，两者的性能差异并没达到二倍的程度。

4.2.5RFC1662 中 CRC 的设计

RFC1662 中提供了 CRC 算法的详细说明，并且提供了软件实现的示例。三个参数的设计（如 `pppfcs32(fcs, cp, len)`）允许函数在已有的 FCS 值基础上继续计算，这使得可以逐块处理数据，而不必一次处理整个数据帧。这在处理大量数据或者分块接收数据时非常有用。

4.2.6 结论

总之，CRC32 提供了一个非常强的错误检测能力，而查表法的使用在实际应用中可以有效地加速 CRC 的计算，使其成为实时数据通信中可行的错误检测方案。对于绝大多数应用场景，CRC32 已经足够可靠，可以满足错误检测的需求。如果客户需要更高的保证，可以考虑使用更高位数的 CRC，如 CRC64，或者结合其他错误校正方法。

4.3 程序设计方面的问题

8.10 节提出的协议软件的跟踪功能有什么意义？你的程序实现这样的功能了吗？程序库中获取时间坐标的函数 get_ms()不是 C 语言标准库中的函数，你能自己实现一个这样的函数吗？ 在“C 语言程序设计”课程中学习过 printf 函数，printf 风格的函数特点是参数数目不确定，类似的标准库函数还有 fprintf，sprintf，scanf 等。为了便于调试程序，程序库中提供了日志文件和屏幕窗 口同步输出的 lprintf 函数(参见 8.4)，这些函数都不是标准的 C 语言库函数，调用风格与 printf 类似。可以参考给出的源代码，看看 printf 风格的函数是怎样实现的。 8.10 节中给出了两个函数 start_timer()和 start_ack_timer()，它们都是定时器函数，两个函数启动定时器的时机不同，而且在定时器到时之前重新调用函数对原残留时间的处理方式也不同，为什么要这样设计？

4.3.1 协议软件的跟踪功能的意义

跟踪功能在协议软件中非常重要，主要是因为它允许开发者和测试者监视和记录协议操作的详细情况，包括数据传输、错误处理、定时器事件等关键操作。这种能力对于调试和优化协议实现至关重要。它可以帮助确定协议的哪些部分在特定条件下表现不佳，哪些部分可能存在编程错误，或者在协议处理特定数据模式时的性能瓶颈。

4.3.2 实现跟踪功能的考虑

在协议开发中实现跟踪功能，通常包括以下几个方面：- **日志记录**：记录每个重要事件的发生时间和相关数据，这有助于事后分析。- **错误跟踪**：特别标记错误事件和异常条件，以便快速定位问题。 - **性能度量**：记录关键操作的执行时间和其他性能指标，以评估协议的效率。

4.3.3get_ms() 函数的实现

在 C 语言标准库中没有直接提供获取毫秒级时间戳的函数。但可以通过以下几种方法实现：

在 Linux 平台上，可以使用 gettimeofday()函数来实现，代码示例如下：

```
#include <sys/time.h>
#include <stdio.h>

unsigned int get_ms() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec * 1000 + tv.tv_usec / 1000;
}
```

在 Windows 平台上，可以使用 GetTickCount()函数：

```
#include <windows.h>

unsigned int get_ms() {
    return GetTickCount();
}
```

4.3.4 定时器函数设计的考虑

start_timer() 和 start_ack_timer() 的不同设计反映了它们在协议中的不同用途：start_timer() 被设计为在物理层发送队列清空之后开始计时。这确保了计时从所有相关数据发送完成后开始，从而更准确地反映超时前等待 ACK 的时间。start_ack_timer() 则立即开始计时，需要快速响应，不依赖于数据发送的完成。

这两种设计方法都考虑到了协议操作中的实际需求，即如何在发送数据和确认数据中找到合适的平衡点。

4.3.5 结论

通过上述方法，可以确保协议软件在各种情况下的正确性和鲁棒性，同时也为开发者提供了及时反馈，以便快速迭代和改进协议。

4.4 软件测试方面的问题

验证所完成的程序能否在各种情况下都能够正确工作，是软件测试环节的主要目的。表 3 中列出了七种测试方案，设计这么多种测试方案的目的是什么？分析每种测试方案，每种方案主要是为了瞄准你的协议软件中可能出问题的哪些环节？或者说，你的协议软件存在什么问题时，测试会失败？你觉得还存在有哪些问题是这些测试尚未覆盖的？这些测试方案和验证协议正确性的手段由指导教师给出，如果是由你自己独立完成整个协议的设计和测试，你会采用哪些手段来验证你的程序能正确工作？针对本次实验的具体问题，你能不能提出一种更高效的软件测试方案？本次实验所提供的程序库还有哪些不足，怎样才能对协议开发提

供更方便的支撑, 这关系到在整体软件开发过程中的不同模块间的功能划分问题, 给出你的建议。

在软件测试阶段, 尤其是针对通信协议软件的测试, 设计多种测试方案的目的是为了确保协议在不同的网络条件和应用场景下都能够正确、有效地工作。这些测试方案通常设计来模拟现实世界中可能遇到的各种极端和常规条件, 以便揭示潜在的缺陷和问题。下面对表 3 中列出的测试方案进行分析，看看它们各自针对协议软件可能出现的哪些问题:

4.4.1 测试方案分析

1. **无误码信道数据传输**：
   - **目的**：验证在理想条件下，协议是否能够无误地传输数据。
   - **可能发现的问题**：基本功能失效，如数据不完整、顺序错误等。
2. **站点 A 分组层平缓方式发出数据**：
   - **目的**：检测协议在正常速率下的数据处理能力和流量控制机制。
   - **可能发现的问题**：发送方的缓冲管理、流量控制不当导致数据丢失或延迟。
3. **站点 B 周期性交替“发送 100 秒，慢发 100 秒”**：
   - **目的**：测试协议在变化的网络带宽下的适应性和鲁棒性。
   - **可能发现的问题**：动态网络条件下流量控制和拥塞控制机制不足。
4. **无误码信道且站点 A 和站点 B 的分组层都洪水式产生分组**：
   - **目的**：检验在高负载下协议的性能和稳定性。
   - **可能发现的问题**：在极端负载下可能出现的性能瓶颈、处理延迟和缓冲区溢出。
5. **站点 A/B 的分组层都洪水式产生分组**：
   - **目的**：模拟高负载同时伴随着轻微误码的网络环境，测试误码恢复机制。
   - **可能发现的问题**：错误恢复策略不充分，如自动重传请求（ARQ）处理不当。
6. **站点 A/B 的分组层都洪水式产生分组，线路误码率设为**10^{-4}：
   - **目的**：在更高误码率下测试协议的错误检测和纠正能力。
   - **可能发现的问题**：高误码环境下的数据完整性问题，CRC 或校验和算法可能不够强大。

4.4.2 未覆盖的问题

- **网络分段和重组**：测试方案可能没有考虑数据包分段和重组的效率和正确性。
- **多种网络拓扑**：协议在不同的网络拓扑结构中的表现可能未被充分测试，如多跳网络环境。
- **安全性测试**：数据传输的加密和认证机制未被提及。

4.4.3 独立完成协议设计和测试的方法

如果由我独立完成整个协议的设计和测试，我会采用以下手段:

- **单元测试**：针对协议的每个独立模块进行详尽的测试。
- **集成测试**：测试模块间的接口和交互。

- **模拟测试**：使用网络模拟器模拟不同的网络条件。
- **性能测试**：评估在不同负载和网络条件下的协议性能。
- **安全性测试**：确保数据传输的安全性和数据完整性。

4.4.4 更高效的测试方案建议

针对本次实验，可以设计一个自动化的测试框架，该框架能够自动调整网络条件参数，并收集关键性能指标，如吞吐量、延迟和数据完整性率。通过这种方式，可以快速地在多种预定义场景下评估协议的表现，大大提高测试的效率和覆盖率。

4.4.5 程序库的不足和改进建议

程序库在支持复杂的网络模拟、错误注入、日志记录和性能分析方面可能还有不足。改进的方法包括：

- **增强调试和分析工具**：集成更先进的调试和性能分析工具。使用类似 WireShark 的网络分析工具，可以更好地监控和分析数据包的传输。

4.4.6 结论

通过这些方法，可以为协议开发提供更全面的支持，从而在整个软件开发过程中实现不同模块间的有效功能划分和协作。

4.5 对等协议实体之间的流量控制

> 在教科书中多次提及“流量控制”问题，这个问题的确很重要。在本次实验所设计的程序中，多多少少也考虑了上下层软件实体之间的数据流量控制问题。你认为你所设计的滑动窗口协议软件有没有解决两个站点的数据链路层对等实体之间的流量控制问题？如果是已经解决了，那么是怎样解决的？如果尚未解决，那么还需要对协议进行哪方面的改进？

在数据通信中，流量控制是确保发送方不会因为发送数据过快而淹没接收方的关键机制。在数据链路层，流量控制常常通过机制如滑动窗口协议来实现。这种协议不仅解决了数据的可靠传输问题，也提供了基本的流量控制功能。

4.5.1 滑动窗口协议如何实现流量控制

滑动窗口协议通过以下几种方式实现流量控制：

1. **窗口大小限制**：发送方的窗口大小限制了它可以发送而不需要确认的数据帧的数量。这个窗口大小通常是根据接收方的缓冲能力来设置的，从而确保接收方不会因为接收到的数据过多而处理不过来。
2. **确认和超时机制**：接收方通过发送确认（ACK）告知发送方哪些帧已经成功接收。如果发送方在设定的超时时间内没有收到确认，它会重传那些未被确认的帧。这样确保了数据的可靠传输同时也控制了数据的流速。

3. **接收窗口的通告**：接收方可以通过控制字段（如 TCP 头中的窗口大小）来动态调整发送方的窗口大小，这是一种更为动态的流量控制方法，允许接收方根据当前的处理能力来控制发送方的发送速度。

4.5.2 滑动窗口协议的流量控制效果

滑动窗口协议在以下方面有效解决了流量控制问题：

- **适应性**：窗口大小可以动态调整，适应网络条件的变化和接收方的处理能力。
- **弹性**：通过超时重传机制，协议能够应对网络中的丢包和错误，自动进行数据的重传。
- **公平性**：在多个通信实体共享网络资源时，滑动窗口协议通过控制每个实体的窗口大小帮助实现公平的带宽分配。

4.5.3 需要改进的方面

尽管滑动窗口协议为流量控制提供了基础，但在某些高级应用或极端网络条件下，可能需要进一步的改进：

1. **更智能的窗口调整策略**：现有的滑动窗口协议预设好了固定的窗口大小。更智能的算法，如基于预测模型的流量控制，可能会根据实时的网络状态动态调整窗口大小。
2. **网络拥塞的考虑**：在网络拥塞时，仅仅依靠接收方的能力来调整窗口大小可能不够。集成拥塞控制机制，如 TCP 中的拥塞避免算法，可以进一步改善性能。
3. **更细粒度的控制**：对于需要高度可靠性的应用，可能需要在滑动窗口协议中实现更细粒度的控制机制，例如按优先级处理不同类型的数据帧。

4.5.4 结论

综上所述，滑动窗口协议为两个站点的数据链路层对等实体之间的流量控制问题提供了基本的解决方案。然而，根据具体应用和网络环境的需求，可能还需要对协议进行进一步的优化和改进。

4.6 与标准协议的对比

如果现实中有两个相距 5000 公里的站点要利用你所设计的协议通过卫星信道进行通信，还有哪些问题需要解决？实验协议离实用还有哪些差距？你觉得还需要增加哪方面的功能？从因特网或其他资料查阅 LAPB 相关的协议介绍和对该协议的评论，用成熟的 CCITT 链路层协议标准对比你所实现的实验性协议，实验性协议的设计还遗漏了哪些重要问题？

通过卫星信道进行通信，尤其是涉及到长达 5000 公里的距离时，将会面临多种挑战和限制。这些问题包括但不限于信号延迟、信号衰减、更高的误码率以及可能的信道拥堵。针对这些问题，我们可以从以下几个方面来探讨所设计的协议与成熟的链路层协议（如 LAPB）的差距和需要改进的地方。

4.6.1 面临的主要挑战

1. **高延迟**：卫星通信的信号往返延迟通常在 240 毫秒到 480 毫秒之间，这远高于地面网络。高延迟会影响协议的效率，尤其是在等待确认（ACK）和超时重传方面。
2. **信号衰减和误码率**：卫星信号在传输过程中可能会遭受衰减和干扰，从而增加误码率。这要求链路层协议必须具有较强的错误检测和纠正能力。
3. **带宽限制**：尽管现代卫星通信技术已大幅提高带宽，但相比地面光纤网络，其带宽仍然较低，且成本较高。

4.6.2 协议设计考虑

针对上述挑战，我们的实验性协议需要在以下几个方面进行优化或增加新功能：

1. **优化超时重传机制**：基于固定超时时间的重传机制可能不适用于卫星通信，需要根据实际的往返时间（RTT）动态调整超时计时器。
2. **引入更复杂的错误控制**：增强错误控制机制，如引入前向纠错（FEC）技术，可以在数据到达接收端之前预先纠正错误，减少需要重传的数据量。
3. **窗口大小的动态调整**：考虑到高延迟的影响，动态调整窗口大小以适应变化的网络条件是提高卫星链路效率的关键。
4. **多级流量控制**：实施更细致的流量控制策略，以适应卫星通信中的带宽限制和变化。

4.6.3 对比 LAPB 协议

LAPB（链路访问程序平衡）是一种成熟的链路层协议，广泛用于 X.25 网络。它提供了：

- **帧同步**：确保数据帧的正确边界。
- **透明性**：数据中的控制字符不会被解释为命令。
- **错误检测**：使用 FCS（帧检查序列）检测错误。
- **流量控制**：通过窗口机制实现。
- **有序传输**：保证数据帧的顺序。

4.6.4 实验性协议的不足

与 LAPB 相比，实验性协议可能在以下方面存在不足：

- **健壮的错误处理**：可能没有足够强大的机制来处理高误码率环境。
- **高效的流量控制**：对高延迟环境的适应性可能不足。
- **协议操作的成熟度和稳定性**：LAPB 作为一个标准，已经在全球范围内得到了广泛的实施和验证。

4.6.5 结论

综上所述，虽然实验性协议提供了基础的链路层控制，但要达到 LAPB 那样成熟、稳定且能有效应对卫星通信特有挑战的水平，还需要进一步的研究和开发。这包括对协议参数的动态调整、错误控制的增强以及流量控制策略的优化。

第五章 实验总结和心得体会

这次实验是我第一次设计和实现一个完整的通信协议软件, 虽然遇到了许多困难和挑战, 但通过不懈的努力, 最终完成了实验目标。在这个过程中, 我学到了很多关于网络通信、协议设计和软件开发的知识, 也提升了自己的编程技能和解决问题的能力。

5.1.1.1 实际上机调试时间

预计的完成时间为 4 6 个小时, 这和我的完成时间基本一致, 这包括编程、调试和多次测试来验证程序的正确性和性能。

5.1.1.2 协议方面问题

1. **协议死锁**: 在特定条件下, 发送和接收过程陷入死锁状态。这是由于确认消息处理不当导致的（序号处理错误)。对协议状态机进行了重构, 解决了死锁问题。

5.1.1.3 综合提升

- **C 语言技能**: 对 C 语言的掌握更加熟练, 特别是在指针操作、内存管理和模块化编程方面。
- **协议软件开发**: 对网络协议的理解更深入, 能够设计和实现复杂的通信协议。
- **理论知识**: 通过实践加深了对数据链路层滑动窗口协议的理解。
- **软件工程**: 学习了软件开发的整个生命周期, 包括需求分析、设计、编程、测试和维护等。

总体来说, 这次实验不仅提升了我的编程技能和协议设计能力, 还增强了我解决复杂问题和使用工具的能力。尽管遇到了许多挑战, 但通过这些解决问题的过程, 我获得了宝贵的经验和自信。