厦門大學

信息学院软件工程系

《计算机网络》实验报告

题　　目　实验二　利用可见光传输帧的软件

班　　级　数字媒体技术 2022 级 1 班

姓　　名　魏清晨

学　　号　37220222203790

实验时间　2024 年 10 月 6 日

2024 年 10 月 6 日

填写说明

1、本文件为 Word 模板文件，建议使用 Microsoft Word 2021 打开，在可填写的区域中如实填写；

2、填表时勿改变字体字号，保持排版工整，打印为 PDF 文件提交；

3、文件总大小尽量控制在 1MB 以下，最大勿超过 5MB；

4、应将材料清单上传在代码托管平台上；

5、在实验课结束 14 天内，按原文件发送至课程 FTP 指定位置。

1 实验目的

通过完成实验，理解数据链路层传输的基本原理。掌握传输过程中的帧格式设计理念；熟悉传输中的帧与成帧、帧定界符等通信概念，熟悉多方通信中的时分、频分、波分或码分多路复用与解复用等概念，熟悉多方通信中的编址的概念。

2 实验环境

操作系统：Win11　编程语言：C++　　调试软件：CLion2023.2

3 实验结果

实验步骤：

编码接口调用：

```
PS E:\code\practice\Computer_Network\Experiment\cni-exp\E2_3790\bin> .\encode.exe encode ./ 512 video.mp4 15000
```

等待编码（如果有 warning，是 ZBar 库没能识别二维码）：

```
Active code page: 65001
每张二维码携带的数据量: 1560B
[##################################################] 100% 二维码编码完成
[##################################################] 100% 二维码绘制完成
PS E:\code\practice\Computer_Network\Experiment\cni-exp\E2_3790\bin>
```

解码接口调用（此处已用手机拍摄）：

```
PS E:\code\practice\Computer_Network\Experiment\cni-exp\E2_3790\bin> .\decode.exe decode video.mp4 output_folder ./
```

等待解码完成：

```
Active code page: 65001
[########################----------------------------] 48% 二维码解码中WARNING: C:/M/B/src/zbar-0.23.93/zbar/decoder/databar.c:1210: _zbar_decode_databar: Assertion "seg->finder >= 0" failed.
        i=14 f=-1(101) part=0
[##################################################] 100% 二维码解码完成
1.bin文件传输正确率: 84.49%
2.bin文件传输正确率: 98.41%
3.bin文件传输正确率: 94.11%
4.bin文件传输正确率: 83.81%
PS E:\code\practice\Computer_Network\Experiment\cni-exp\E2_3790\bin>
```

核心代码逻辑（主要是和上次实验不同的，这次实验的目的相关的）：

Encoder 部分（分时复用部分）：

```
        // 分时复用
        for (int t = 0, i = ch_per_qr - 1; ; t++, i += ch_per_qr)
        {
#ifndef DEBUG
            print_progress_bar( progress: current_data_size, total: total_size, info: "二维码编码中");
#endif
            // 是否所有的数据都处理完毕
            bool flag = false;
            for (auto& [file_path :const path , file_data :vector<unsigned char> ] : input_file_vectors)
            {
                // 生成载荷部分
                vector<uchar> qr_data = uchar_to_qrcode( &: input_file_vectors[file_path], end_idx: i, ch_per_qr);
                if (qr_data.empty()) continue;
                current_data_size += qr_data.size() - 10;

                flag = true;

                // 生成帧
                DataFrame data_frame = DataFrame( &: qr_data, source: (uint8_t)stoi( str: file_path.stem().string()), destination: 0);
                data_frame.generate_crc32();

                vector<uchar> serialized_frame = serialize( data: data_frame);
                // 为什么要用base64编码?
                // 因为zbar检测二维码是按照utf-8编码读数据的，所以如果最高位是1，就会变成c2/c3开头的宽字符，为了避免，我们使用base64，即四个6位bit表示3个char
                base64::Encoder b64_encoder = base64::Encoder();
                serialized_frame = b64_encoder.base64_encode( input: serialized_frame);

                QRcode qrCode = *QRcode_encodeData( size: (int)serialized_frame.size(), data: serialized_frame.data(), version: 0, level: QR_ECLEVEL_H);

                qr_arr.push_back(qrCode);

                // 测试识别二维码得到的帧数据是否一致
#ifdef DEBUG
                ...
#endif
            }

            if (!flag) break;
        }
```

每一个周期把所有的数据都生成一个帧出来，直到没有任何一个数据传来帧，剩余部分（上一个实验）不需要改变，使用 base64 原因见注释

Decode 部分（检验和编址部分）：

```
// 解码得帧数据
vector<uchar> current_frame_data_string;
decode( input_image: mat, &: current_frame_data_string);

// 没收到数据
if (current_frame_data_string.empty()) continue;

base64::Decoder b64_decoder = base64::Decoder();
current_frame_data_string = b64_decoder.base64_decode( input: current_frame_data_string);
DataFrame current_frame_data = DataFrame( &: current_frame_data_string);

// crc检验有误
if (!current_frame_data.verify_crc32()) continue;
// 长度和数据的大小中有一个不一样
if (current_frame_data.length != current_frame_data.data.size()) continue;

QrData current_qr_data = QrData( &: current_frame_data.data);
```

主要增加了 crc 检验以及 base64 的解码，获得帧数据后直接获取载荷并反序列化作为二维码数据的结构体数据

```
// 数据接收开始
if (!data_start.contains( x: current_frame_data.source))
{
    if (current_qr_data.start)
    {
        data_start.insert( x: current_frame_data.source);
    } else
    {
        continue;
    }
}

// 二维码数据序号一样，重复
if (!previous_data[current_frame_data.source].data.empty() &&
    previous_data[current_frame_data.source].index == current_qr_data.index) continue;

// 有中间二维码没识别出来
if (!previous_data[current_frame_data.source].data.empty() &&
    previous_data[current_frame_data.source].index + 1 < current_qr_data.index)
{
    forup (k, 1, current_qr_data.index - previous_data[current_frame_data.source].index - 1)
    {
        QrData recovery_qrcode = QrData();
        recovery_qrcode.index = previous_data[current_frame_data.source].index + k;
        recovery_qrcode.data = vector<uchar>( n: previous_data[current_frame_data.source].len,  value: 'a');
        append_data( output_file_path: output_info_directory + std::format( fmt: "{:d}.bin",  &: (int)current_frame_data.source),  &: recovery_qrcode.data);
    }
}

encoded_data = current_qr_data;

append_data( output_file_path: output_info_directory + std::format( fmt: "/{:d}.bin",  &: (int)current_frame_data.source),  &: encoded_data.data);

previous_img = &mat;
previous_data[current_frame_data.source] = current_qr_data;

// 数据接收结束
if (current_qr_data.end) break;
```

另外数据的写和校正是通过帧中的源地址 source 进行分组区别的（见所有高亮部分）

4 实验代码

本次实验的代码已上传于以下代码仓库：[CNI-Exp: 厦门大学计算机网络课程实验项目集 (gitee.com)]

5 课后思考题

1、 在实验中的，帧格式是什么？ 各个字段的作用是什么？长度范围是多少？

帧首定界符	目的地址	源地址	长度	载荷	CRC
1 字节	1 字节	1 字节	2 字节	0~512 字节	4 字节

帧首定界符用于判定帧的开始，目的地址用于确定帧的接收方，源地址用于确定帧的发送方，长度用于表示数据长度，载荷就是帧所携带的数据，crc 是校验码

2、 在实验中的，如何将数据成帧？其中，帧首定界符是什么？

这里是将数据写入结构体，再将结构体序列化为字符串，但其实等价于在数据前后添加帧首定界符、目的地址等字符。

帧首定界符是 0x3F

3、 在实验中的，采用的是何种多路复用算法？ 简要说明其基本原理。

时分多路复用。

时分多路复用就是每一段时间内由一个发送端传送数据，下一段时间由下一个发送端传送数据

4、 在实验中，编址方案是什么？ 为什么这么设计？

这里的话不涉及网络传输，所以没有传统上的编址，不过这里的话我是通过源地址的值来区分不同来源的数据，以此来写入不同的文件。

关于为什么这么设计，一是简单方便，实验实现区分足够，易于解码而且比较短也不容易出错

5、 **你的编码数据载荷大小范围是什么？ 设定该上限值的依据是什么？**

这里的话是 0~0.5KB

依据主要有两点，一是 libqrencode 库对字节编码二维码最多编 1.4KB，第二点是对于一些数据量太大的二维码可能识别不出来

6 实验总结

本次实验是在上次实验的基础上完成，改动的地方也不多，但也遇到了不少问题。

首先谈谈收获，主要是亲手实现了帧格式，模拟了时分多路复用，学习了帧首定界符出现在数据中的集中解决方案。

另外，还有遇到的一些问题，主要有一个 bug：由于解码的数据识别格式是 utf-8，导致首位为 1 的数据变成宽字符，后来使用本被放弃的 base64 编码解决。