《计算机网络》Lab 1 实验报告

1900017812 高乐耘*

2023 年 3 月 15 日

I. 实验环境

- 编码环境：22.04.2 LTS (GNU/Linux 5.19.0-35-generic x86_64), GCC 11.3.0
- 调试环境：Windows 11 Professional 21H2 22000.1574, MinGW-W64 GCC 12.2.0
- 测试环境：Windows XP Professional SP2, NetRiver V 1.21, MinGW GCC 3.4.2

II. 实验准备：平台无关的字节序处理流程

由于编写、编译和测试程序跨越多个平台，各平台使用的字节序不一定相同。尽管我编写程序所使用的笔记本电脑和学校机房的台式电脑均为 x86_64 架构的 CPU，使用小端序，然而一个鲁棒的网络程序首先需要实现的，便是平台无关的字节序处理。

虽然 C++20 标准宣告可以使用 `std::endian` 枚举类型中的常量获取平台字节序，然而实际尝试发现即使是使用 g++ 11.3.0 并指定 -std=c++20，也无法使用这一特性。在 Linux 下，头文件 endian.h 中声明了在大/小端序和本机字节序之间转换不同大小的整数的一组函数，许多操作系统也提供了在网络和本机字节序之间转换不同大小的整数的网络编程接口（`hton` 和 `ntoh` 的 `l` 和 `s` 版本），然而它们均不是 C/C++ 标准的一部分，故在实际的跨平台应用中应当尽量避免使用。

下面为处理本机和网络之间字节序转换的一个非常简单但可靠的实现。

```
/*
 * 帮助函数：获取本机字节序
 *
 * 返回值
 *      0: 大端序，与网络一致
 *      1: 小端序，与网络不一致
 */
static UINT8 endian(void)
{
  UINT16 prob = 1;
  return *(UINT8 *)&prob;
}
```

`endian()` 函数实现了获取字节序的一个简单的方法：检验多字节整数 1 的最低地址处的字节是否为 1，如果为 1，则符合低地址低字节的原则，为小端序，否则为大端序。

*电子邮件地址: seeson@pku.edu.cn　手机号: 13759115414

```
/*
 * 帮助函数：改变字节序
 */
static UINT32 bswap32(UINT32 u32)
{
  UINT32 u0 = (u32 >> 24) & 0xff;
  UINT32 u1 = (u32 >> 16) & 0xff;
  UINT32 u2 = (u32 >>  8) & 0xff;
  UINT32 u3 = (u32 >>  0) & 0xff;
  return (u0 << 0) | (u1 << 8) | (u2 << 16) | (u3 << 24);
}
```

bswap32() 函数实现了将 32 位整数按字节颠倒顺序，即转换字节序的功能。在 GCC 中，它可以被 __builtin_bswap32() 代替并在大多数平台上被编译为一条汇编指令。

```
/*
 * 帮助函数：网络字节序 -> 主机字节序
 */
static UINT32 ntoh32(UINT32 u32)
{
  if(endian()) {  /* 应从大到小 */
    return bswap32(u32);
  }
  return u32;
}

/*
 * 帮助函数：主机字节序 -> 网络字节序
 */
static UINT32 hton32(UINT32 u32)
{
  return ntoh32(u32);
}
```

ntoh32() 和 hton32() 函数在网络和本机之间进行字节序转换。只有当本机为小端序时，才需要改变字节序。在结果上，这两个函数的功能是一样的，但它们的语义不同。

III. 本实验用到的数据结构

为了增强可移植性，下面我将本实验用到的数据结构改成了平台无关的版本。在实验指导书中，出现了 enum，unsigned int 等大小依赖平台和编译器的表述，经我在实验平台上求证后，替换成了对应的大小确定的类型。此外，实验指导书中的小写枚举成员也被我替换成了大写，以符合 C/C++ 程序一般的符号命名规则。这里帧类型没有再使用枚举类型，而是使用了 UINT32 类型，避免因 unsigned int 大小的不确定性造成枚举类型变量大小的不确定性。虽然 C++ 允许指定枚举的底层类型，但这里我使用了与 C 更好兼容的宏定义方法。

```
/*
 * 帧类型
 */
#define FRAME_KIND_DATA  0
#define FRAME_KIND_ACK   1
#define FRAME_KIND_NAK   2

/*
 * 帧头
 */
typedef struct frame_head {
  UINT32 kind;              /* 大端帧类型 */
```

```
  UINT32 seq;                /* 大端序列号 */
  UINT32 ack;                /* 大端确认号 */
  unsigned char data[100];   /* 二进制数据 */
} frame_head;

/*
 * 全帧
 */
typedef struct frame {
  frame_head head;  /* 帧头 */
  UINT32 size;      /* 大端数据大小 */
} frame;
```

本实验中我们只关心 `frame_head` 结构体中的 `kind`，`seq` 和 `ack` 字段，对于其它部分，作到逐字节拷贝即可。需要注意的时，上述结构体中的所有 32 位整形变量均为网络字节序，即大端序。此外，我使用了链表实现等待发送或确认的帧队列，有关数据结构定义如下：

```
/*
 * 帧队列链表节点
 */
typedef struct flist_node {
  frame *pframe;             /* 帧内容的拷贝 */
  int flen;                  /* 帧长度 */
  struct flist_node *prev;   /* 上一节点 */
  struct flist_node *next;   /* 下一节点 */
} flist_node;

/*
 * 帧队列链表
 */
typedef struct flist {
  flist_node *head;  /* 头节点 */
  flist_node *tail;  /* 尾节点 */
  int size;          /* 节点数 */
} flist;
```

IV. 协议下层接口的包装

本实验实现的链路层协议是基于这个测试系统提供的接口：

```
/*
 * 系统函数：发送帧
 *
 * pData: 指向要发送的帧的内容的指针
 * len:   要发送的帧的长度
 */
extern void SendFRAMEPacket(unsigned char *pData, unsigned int len);
```

为了方便调用、调试和插入日志，现对其进行以下包装：

```
/*
 * 包装函数：发送帧
 */
void send_frame(frame *pframe, int flen)
{
  printf("*** %s: seq=%d flen=%d\n", __func__, ntoh32(pframe->head.seq), flen);
  SendFRAMEPacket((unsigned char *)pframe, (unsigned int)flen);
}
```

统一的强制类型转换减少了转换次数，使得代码更加简洁清晰，也降低了犯错概率。这一包装函数方便我们插入统一的调试代码和以日志的形式获取更多系统状态信息。

V. 停等协议的实现

```
/*
 * 停等协议测试函数
 *
 * pBuffer:      指向系统要发送或接收到的帧内容的指针,
 *               或者指向超时消息中超时帧的序列号内容的指针
 * bufferSize:   pBuffer 表示内容的长度
 * messageType: 传入的消息类型, 可以为以下几种情况
 *      MSG_TYPE_TIMEOUT  某个帧超时
 *      MSG_TYPE_SEND     系统要发送一个帧
 *      MSG_TYPE_RECEIVE  系统接收到一个帧的 ACK
 */
int stud_slide_window_stop_and_wait(char *pBuffer, int bufferSize,
    UINT8 messageType)
{
  /* 发送队列链表 */
  static flist sndlst = flist_empty;

  /* 帧内容和长度 */
  frame *pframe;
  int flen;

  /* 超时帧的序列号 */
  UINT32 seq;

  switch(messageType) {

  case MSG_TYPE_TIMEOUT:  /* 某个帧超时 */

    /* 获取超时帧的序列号 */
    seq = *(UINT32 *)pBuffer;

    printf("*** %s: MSG_TYPE_TIMEOUT: seq=%d\n", __func__, seq);

    /* 该序列号必定与发送队头相等 */
    assert(sndlst.head != NULL
        && ntoh32(sndlst.head->pframe->head.seq) == seq);

    /* 获取帧内容和长度 */
    pframe = sndlst.head->pframe;
    flen = sndlst.head->flen;

    /* 重发该帧 */
    send_frame(pframe, flen);

    break;

  case MSG_TYPE_SEND:  /* 系统要发送一个帧 */

    /* 获取帧内容和长度 */
    pframe = (frame *)pBuffer;
    flen = bufferSize;

    printf("*** %s: MSG_TYPE_SEND: seq=%d\n",
        __func__, ntoh32(pframe->head.seq));

    /* 将当前帧压入队列链表 */
    flist_push(&sndlst, pframe, flen);

    /* 如果队列链表中没有等待 ACK 的帧, 发送当前帧 */
    if(sndlst.size == 1) {
      send_frame(pframe, flen);
```

```
    }

    break;

  case MSG_TYPE_RECEIVE:  /* 系统接收到一个帧的 ACK */

    /* 获取帧内容和长度 */
    pframe = (frame *)pBuffer;
    flen = bufferSize;

    printf("*** %s: MSG_TYPE_RECEIVE: ack=%d\n",
        __func__, ntoh32(pframe->head.ack));

    /* 检查 ACK 是否与 SEQ 对应 */
    if(sndlst.size == 0) break;
    if(pframe->head.ack != sndlst.head->pframe->head.seq) {
      break;
    }

    /* 如果对应，则移除发送队列链表头 */
    flist_pop(&sndlst);

    /* 没有其它排队任务时才退回 */
    if(sndlst.size == 0) break;

    /* 获取帧内容和长度 */
    pframe = sndlst.head->pframe;
    flen = sndlst.head->flen;

    /* 发送该帧 */
    send_frame(pframe, flen);

    break;

  }

  flist_print(&sndlst);
  return 0;  /* XXX */
}
```

停等协议要求发送方一次只能发送一帧，而后必须等待收到相应 ACK 消息之后，才能开始发送下一帧。对上层在一帧未确认之前发出的其它发送帧请求，必须将其挂起等待。函数 `stud_slide_window_stop_and_wait()` 通过调用 `flist_push()` 函数将挂起等待的帧压入发送队列链表 `sndlst` 的尾部，等待成功处理 ACK 消息后再按从头到尾的顺序一一取出进行发送。函数 `flist_push()` 带有深拷贝语义，即函数会分配适当的空间以完整拷贝挂起等待发送的帧 `*pframe`，并记录需要作为参数传给下层接口的帧大小 `flen`。被确认收到的帧被 `flist_pop()` 调用从帧队列中删除，其占有的内存资源同时被释放。

`stud_slide_window_stop_and_wait()` 函数中使用了静态变量 `sndlst` 作为发送队列，这一实现确保了每次函数调用可以看到它的历史，从而可以实现对过往发送失败的帧进行重传等功能。这在实际应用中不一定是一个好的办法。直观上不加保护地使用静态变量将引入竞争条件，在异步网络应用中可能产生未定义行为。然而，局部加锁也并非可靠的解决方案，如果这一函数是受事件驱动而被异步调用的话，如不对临界区内的中断进行限制，将不得不面临潜在的死锁问题。故在主程序控制流中实现消息队列和事件顺序处理的同步机制更为合理，如此单个的事件处理例程如 `stud_slide_window_stop_and_wait()` 函数便无需考虑静态资源访问的同步互斥问题了。

`stud_slide_window_stop_and_wait()` 函数中有多个以 `printf()` 函数实现的日志输出功能，用来监测系统的状态，同时也确保对字节序的处理准确无误。值得一提的是，实验指导书中并未明确指出处理 `MSG_TYPE_TIMEOUT` 事件时传入的 SEQ 号为本机字节序，此时 `printf()` 函数打印出的日志便给调试工作帮了大忙。实验指导书也并未明确函数退回值的意义，我只好在末尾退回 0，并在注释中打上 XXX 标记。退回语句前的 `flist_print()` 调用将发送队列链表中的所有帧的 SEQ 号打印出来，作为更丰富的调试信息的一部分。

最后让我们给出程序编译和链接 NetRiver 主程序和运行时库后与评测服务器通信的完整日志结果，其中我用 `printf()` 输出的日志均以三个星号（***）开头，以方便辨认：

```
begin test!, testItem = 0  testcase = 0
accept len = 32 packet
accept len = 946 packet
frame seq ==========================1
*** stud_slide_window_stop_and_wait: MSG_TYPE_SEND: seq=1
*** send_frame: seq=1 flen=29
send a message to main ui, len = 39  type = 2  subtype = 1
*** flist_print: 1
frame seq ==========================2
*** stud_slide_window_stop_and_wait: MSG_TYPE_SEND: seq=2
*** flist_print: 1 2
frame seq ==========================3
*** stud_slide_window_stop_and_wait: MSG_TYPE_SEND: seq=3
*** flist_print: 1 2 3
frame seq ==========================4
*** stud_slide_window_stop_and_wait: MSG_TYPE_SEND: seq=4
*** flist_print: 1 2 3 4
frame seq ==========================5
*** stud_slide_window_stop_and_wait: MSG_TYPE_SEND: seq=5
*** flist_print: 1 2 3 4 5
accept len = 24 packet
send a message to main ui, len = 22  type = 2  subtype = 0
receive a frame
*** stud_slide_window_stop_and_wait: MSG_TYPE_RECEIVE: ack=1
*** send_frame: seq=2 flen=29
send a message to main ui, len = 39  type = 2  subtype = 1
*** flist_print: 2 3 4 5
accept len = 24 packet
send a message to main ui, len = 22  type = 2  subtype = 0
receive a frame
*** stud_slide_window_stop_and_wait: MSG_TYPE_RECEIVE: ack=2
*** send_frame: seq=3 flen=29
send a message to main ui, len = 39  type = 2  subtype = 1
*** flist_print: 3 4 5
*** stud_slide_window_stop_and_wait: MSG_TYPE_TIMEOUT: seq=3
*** send_frame: seq=3 flen=29
send a message to main ui, len = 39  type = 2  subtype = 1
*** flist_print: 3 4 5
accept len = 24 packet
send a message to main ui, len = 22  type = 2  subtype = 0
receive a frame
*** stud_slide_window_stop_and_wait: MSG_TYPE_RECEIVE: ack=3
*** send_frame: seq=4 flen=29
send a message to main ui, len = 39  type = 2  subtype = 1
*** flist_print: 4 5
accept len = 24 packet
send a message to main ui, len = 22  type = 2  subtype = 0
receive a frame
*** stud_slide_window_stop_and_wait: MSG_TYPE_RECEIVE: ack=4
*** send_frame: seq=5 flen=29
send a message to main ui, len = 39  type = 2  subtype = 1
```

```
*** flist_print: 5
accept len = 24 packet
send a message to main ui, len = 22  type = 2  subtype = 0
receive a frame
*** stud_slide_window_stop_and_wait: MSG_TYPE_RECEIVE: ack=5
*** flist_print: (empty)
accept len = 6 packet
result = 0
send a message to main ui, len = 6  type = 1  subtype = 7
Test over!
```

上面的输出结果中，有以下几个要点：

- “停等”的正确实现：5 个发送帧请求连续到达，发送队列大小一度增长至 5，然而在 ack=1 的 ACK 消息到来之前，只有 seq=1 的帧被发送了；当这个 ACK 到来之后 seq=2 的帧随即被发出，“停”的时机和“等”的期限均合理适当
- 超时重传的正确实现：seq=3 的帧超时重传，它正好对应发送队列链表的首个节点，重传之后链表头没有弹出
- 资源管理的正确实现：链表加深拷贝机制正确实现了缓存队列，发送的所有帧被确认收到后链表回空，程序正确释放了申请的内存资源

最后不得不说，ACK 确认相同序列号的数据帧是大坑，且测试平台的编译环境默认不在 `assert()` 宏断言失败时退出，这造成了一定的调试困难，但在引入 `printf()` 日志之后得以顺利解决。

VI. 回退 N 帧协议的实现

与停等协议不同的是，回退 N 帧的发送窗口大小大于 1，特别地，实验规定发送窗口的大小如下：

```
#define WINDOW_SIZE_BACK_N_FRAME  4
```

为了实现窗口大小的限制，相比停等协议，需要在代码中增加如下两个静态变量：

```
/* 当前已发送且等待 ACK 的帧数 */
static int nwaiting = 0;

/* 对应第一个被挂起等待发送的帧 */
static flist_node *first_pending = NULL;
```

两个变量所提供的信息是相同的，但由于实现上链表不支持随机访问，故从 `nwaiting` 和 `sndlst.head` 推知 `first_pending` 或相反过程都比较费时，所以同时使用了这两个变量。

实验手册中没有提到，但课程补充说明了，本协议中接收方发回的 ACK 消息语义为累计确认语义。这可以导出一条非常重要的推论：所有的超时帧必定落在发送窗口内。此外，课程补充要求，处理超时事件只需重发发送窗口中的所有帧，这样可以确保重发了所有可能丢失或因错误被接收方拒收的所有已发送帧。

与停等协议类似的是，回退 N 帧协议也实现了窗口满时发送任务的挂起和发送任务确认完成后提交被挂起任务的逻辑，只不过它们需要读取和修改 `nwaiting` 和 `first_pending` 将窗口右侧的任务移入窗口。本实验中回退 N 帧协议还实现了累计确认逻辑，如果确认号对

应发送队列中的某帧则将确认号对应帧及其以前的帧全部弹出，并更新 `nwaiting`，将发送窗口的左沿右移。

本部分完整的代码实现和测试输出参见：

`https://github.com/lyazj/netriver-labs/blob/main/lab1/lab1.cpp#L337-L477`
`https://github.com/lyazj/netriver-labs/blob/main/lab1/lab1-test-2.txt`

由于篇幅所限在此不再一一列出。从上述链接中的输出结果，可以归纳总结出以下要点：

- 有界发送窗口的正确实现：测试系统连续提交了 6 个发送帧请求，但由于滑动窗口的大小为 4，仅有前 4 个发送任务被立即提交执行，后 2 个则挂起在队列尾部；系统随后收到前 2 帧确认收到的消息，随即发送了后 2 帧，挂起的任务被正确重启提交
- 累计确认语义的正确实现：测试系统收到了 ack=2 的 ACK 消息，它被正确理解为确认收到 seq=1 和 seq=2 的帧，滑动窗口的左沿被正确更新；对之后的 ack=6 的 ACK 消息处理也是如此
- 回退 N 帧的正确实现：在本实验中，回退 N 帧的行为变更为重发滑动窗口中的所有帧，在处理 seq=3 和 seq=4 的 TIMEOUT 事件时，程序均正确地重新发送了滑动窗口内的全部的 3–6 帧

VII. 选择性重传协议的实现

本部分协议与回退 N 帧协议使用了相同大小的有界发送窗口，相同的发送队列溢出滑动窗口右沿时的挂起-重启机制，还有相同的累计确认语义。与之前最大的不同在于，程序在处理 RECEIVE 事件时需要判断收到的消息是 ACK 还是 NAK，这通过检查 `frame_head` 结构体中的 `kind` 字段实现，同时此处也不可忽略必要的字节序转换。与上面的回退 N 帧协议不同的是，NAK 到来时程序并非重发滑动窗口中的所有帧，而是只重发错误帧。本部分的代码相比上一部分没有特别大的改动，具体实现及测试输出见下面两个链接：

`https://github.com/lyazj/netriver-labs/blob/main/lab1/lab1.cpp#L479-L599`
`https://github.com/lyazj/netriver-labs/blob/main/lab1/lab1-test-3.txt`

让我们还是以归纳测试输出中新的要点作结：

- ACK 和 NAK 消息的正确区分：区分的正确性由通过评测机的严格测试保证，而从我 `printf()` 出的日志看，评测机确实产生了 ACK 和 NAK 两类消息，故而程序可以正确区分 ACK 和 NAK 消息并进行不同的处理
- 对 NAK 消息的正确处理：评测机在发出 ack=2 的 ACK 消息后，发出了 ack=3 的 NAK 消息，从输出中看到程序重新发送了 seq=3 的帧，且发送队列没有改变

NAK 消息对应的序列号由 ack 给出，这一细节也是指导手册中所缺少的。一方面，这促进了实验者的不断尝试、发现问题和解决问题，但另一方面，如此种种也体现出了这份文档完整性的些许不足。

VIII. 完整代码实现和完整测试结果

完整代码实现：

https://github.com/lyazj/netriver-labs/blob/main/lab1/lab1.cpp

完整测试输出：

https://github.com/lyazj/netriver-labs/blob/main/lab1/lab1-test.txt

完整测试结论：

https://github.com/lyazj/netriver-labs/raw/main/lab1/lab1-test.png

---

参考文献

[1] Kurose, J. F., Ross, K. W. (2021). Computer Networking: A Top-Down Approach. Boston, MA: Pearson. ISBN: 978-0-13-592861-5.