网络原理：滑动窗口协议实验报告

黄家晖　2014011330

2016 年 11 月 5 日

1　实验目标

1. 掌握理解课堂上讲述的 “滑动窗口” 技术；
2. 使用 C++ 编程语言实现协议栈**发送方**的核心部分；
3. 熟练掌握使用 NetRiver 系统的方法和技巧，并验证协议的正确性。

2　实验原理

本实验主要实现的协议是 1bit 滑动窗口协议和退后 N 帧协议，下面简单对这两种数据链路层协议作介绍。

2.1　1bit 滑动窗口协议

1bit 滑动窗口协议将原有滑动窗口的发送和接受窗口大小都固定为 1，退化为停 -等协议。其中，发送窗口所指内容代表*要发送的下一个帧号*，接收窗口所指内容代表*希望接受的帧的序号*。

发送方每发送一个分组，窗口右移一位，如果在发送前上一帧超时信号被触发，则需要重新发送上一帧，而接受窗口所要做的即是在收到期望的帧之后回复 ACK，非期望帧则丢弃。

2.2　退后 N 帧协议

退后 N 帧协议需要发送方维护的窗口大小为固定值，下界为*未得到确认的帧的最小序号*，上界为*要发送的下一个帧号*。另外在接收方回复 ACK 信号的时候，采用的是累积确认技术，即窗口下界直接加至不含该回复帧号为止。

当发送方的某一个帧超时的时候，需要从超时帧开始发送所有窗口内的帧，这体现了 “退后 N 帧” 含义。而接收方的窗口处理和 1bit 协议相同。

3　程序设计

在程序设计中，我注意到本次实验需要实现的发送方部分的两个协议除了发送窗口的大小不同之外，其余逻辑均相似。因此编写类 `sliding_window`，通过调用其不同的构造函数确定不同的窗口大小，体现了代码复用的思想。

实现方面，由于 NetRiver 实验系统的网络层不提供禁止功能，因此需要在数据链路层缓存所有网络层发来的分组，我使用 `deque`的数据结构来缓存这些帧，用两个整型无符号变量表示发送窗

口的上界和下界。整体逻辑的实现在函数 `event`中，系统会根据不同的信号对该函数进行调用。下面一一进行解释：

MSG_TYPE_SEND 该信号表示网络层有分组需要发送，程序会首先对帧进行缓存，如果当前的窗口大小没有超过最大值，则发送该分组，并开始计时。

MSG_TYPE_RECEIVE 该信号表示物理层接受到数据帧，由于只实现发送方，这个数据帧即为 ACK 帧，程序会根据上面所讲的策略改变窗口下界的大小，如果窗口大小不超过最大限制，则继续发送缓存中的帧。

MSG_TYPE_TIMEOUT 这个信号来自系统，表示某一个帧超时，此时需要重新发送该超时帧及其之后的所有窗口中的帧。由于系统存在些许问题，所以在实际实现中采用的是重新发送缓存中的帧，由于每次接收到 ACK 都会将已发送且确认接受的帧从缓存中删除，因此重新发送缓存中的帧和发送超时帧在本质上是一样的。

4 实验中遇到的问题

除了上面所提到的有关超时帧的问题之外，一个困扰了我很长时间的 BUG 是因为某些疏忽在程序中采用了非法变量，导致 `deque`结构 assert 崩溃，但是测试程序并没有相应提示，虽然看包的发送结果均正确，但是最终的评测始终无法通过。因此不能仅仅依赖系统的提示来修改 BUG，在自己编码的过程中也应该细心。

5 实验结果

1bit 滑动窗口协议的实验结果如图所示：

编号	时间	源地址	目的地址	协议	数据包描述	实验描述
1	Sat Nov 05 18:15:51.828 2016			FRAME	Frame DATA	1比特滑动窗口协议
2	Sat Nov 05 18:15:51.890 2016			FRAME	Frame ACK	1比特滑动窗口协议
3	Sat Nov 05 18:15:51.890 2016			FRAME	Frame DATA	1比特滑动窗口协议
4	Sat Nov 05 18:15:51.890 2016			FRAME	Frame ACK	1比特滑动窗口协议
5	Sat Nov 05 18:15:51.890 2016			FRAME	Frame DATA	1比特滑动窗口协议
6	Sat Nov 05 18:15:51.906 2016			FRAME	Frame DATA	1比特滑动窗口协议
7	Sat Nov 05 18:15:51.921 2016			FRAME	Frame ACK	1比特滑动窗口协议
8	Sat Nov 05 18:15:51.937 2016			FRAME	Frame DATA	1比特滑动窗口协议
9	Sat Nov 05 18:15:51.937 2016			FRAME	Frame ACK	1比特滑动窗口协议
10	Sat Nov 05 18:15:51.937 2016			FRAME	Frame DATA	1比特滑动窗口协议
11	Sat Nov 05 18:15:51.953 2016			FRAME	Frame ACK	1比特滑动窗口协议

退后 N 帧重传协议的实验结果如下图所示：

编号	时间	源地址	目的地址	协议	数据包描述	实验描述
12	Sat Nov 05 18:15:52.000 2016			FRAME	Frame DATA	回退N帧协议
13	Sat Nov 05 18:15:52.015 2016			FRAME	Frame DATA	回退N帧协议
14	Sat Nov 05 18:15:52.031 2016			FRAME	Frame DATA	回退N帧协议
15	Sat Nov 05 18:15:52.046 2016			FRAME	Frame DATA	回退N帧协议
16	Sat Nov 05 18:15:55.687 2016			FRAME	Frame ACK	回退N帧协议
17	Sat Nov 05 18:15:55.687 2016			FRAME	Frame DATA	回退N帧协议
18	Sat Nov 05 18:15:55.703 2016			FRAME	Frame DATA	回退N帧协议
19	Sat Nov 05 18:15:55.703 2016			FRAME	Frame DATA	回退N帧协议
20	Sat Nov 05 18:15:55.718 2016			FRAME	Frame DATA	回退N帧协议
21	Sat Nov 05 18:15:55.718 2016			FRAME	Frame DATA	回退N帧协议
22	Sat Nov 05 18:15:55.718 2016			FRAME	Frame DATA	回退N帧协议
23	Sat Nov 05 18:15:55.734 2016			FRAME	Frame DATA	回退N帧协议
24	Sat Nov 05 18:15:55.734 2016			FRAME	Frame DATA	回退N帧协议
25	Sat Nov 05 18:15:55.734 2016			FRAME	Frame DATA	回退N帧协议
26	Sat Nov 05 18:15:55.734 2016			FRAME	Frame DATA	回退N帧协议
27	Sat Nov 05 18:15:55.750 2016			FRAME	Frame ACK	回退N帧协议

可见程序正确实现了发送方的功能。

6 思考题

1 1bit 滑动窗口协议为 “停 -等” 协议，单个分组发送之后，必须受到对方的回馈才能继续发送下一个分组。因此，当信道的延迟比较大而带宽充裕的时候，整个信道的利用率非常低。退后 N 帧协议则采取每次发送多个分组的策略，相比 1bit 滑动窗口协议来说能够更加充分地利用带宽。

2 试想一个窗口大小为 m 的退后 N 帧协议。发送方依次发送分组 0 到 $m-1$，而由于网络原因，接收方没有接收到第 1 个包，此时发送方必须退后 $m-1$ 帧重新发送，实际发送效率只有 $\frac{1}{m}$。更一般的来讲，发送方连续发送的分组中只要有一个分组未被正确接受，发送方就必须再次发送该分组及其之后的所有分组，造成了有用信息的浪费。

一个更好的解决方法是采用 “选择性重传协议”，在这种方法中，接收方在链路层使用缓存机制存储固定个数的分组，这样就能使发送方在漏包的时候仅仅重发错误的分组，再由接收方将分组重新排序返回网络层。这样虽然增加了接收方的压力，但是能够大大减小因为重传而造成的带宽浪费。

7 附录：源代码

```
1  #include "sysinclude.h"
2  #include <cstring>
3  #include <cstdio>
4  #include <cstdlib>
5  #include <deque>
6
7  extern void SendFRAMEPacket(unsigned char* pData, unsigned int len);
8
9  #define WINDOW_SIZE_STOP_WAIT 1
10 #define WINDOW_SIZE_BACK_N_FRAME 4
11
12 typedef unsigned char uchar;
13
14 enum frame_kind { data, ack, nak };
15
16 struct frame_head {
17   frame_kind kind;
18   unsigned int seq;
19   unsigned int ack;
20   uchar data[100];
21 };
22
23 struct frame {
24   frame_head head;
25   unsigned int size;
```

```
26 };
27
28 struct buffer_element {
29   frame* payload;
30   unsigned int size;
31   buffer_element() { payload = 0; size = 0; }
32   ~buffer_element() { if (payload) delete payload; }
33 };
34
35 class sliding_window {
36   const int win_size_max;
37   std::deque<buffer_element*> buffer;
38
39   unsigned int min_ack_expected;
40   unsigned int next_frame;
41 public:
42   sliding_window(int wsize) : win_size_max(wsize) {
43     min_ack_expected = 1;
44     next_frame = 1;
45   }
46   ~sliding_window() {
47     for (std::deque<buffer_element*>::iterator it = buffer.begin();
48         it != buffer.end(); ++ it) {
49       delete (*it);
50     }
51     buffer.clear();
52   }
53   int event(char *pBuffer, int bufferSize, UINT8 messageType) {
54     switch (messageType) {
55       case MSG_TYPE_SEND: {
56         // Buffer the incoming frame
57         buffer_element* buf = new buffer_element();
58         buf->payload = make_frame_copy((frame*)pBuffer, bufferSize);
59         buf->size = bufferSize;
60         buffer.push_back(buf);
61
62         if ((next_frame - min_ack_expected) < win_size_max) {
63           next_frame = ntohl(buf->payload->head.seq) + 1;
64           SendFRAMEPacket((uchar*)pBuffer, bufferSize);
65         }
66
67         break;
68       }
69       case MSG_TYPE_RECEIVE: {  // only ack.
70         unsigned int ack = ntohl(((frame*)pBuffer)->head.ack);
71         while (ack < next_frame && ack >= min_ack_expected) {
72           buffer_element* del_buf = buffer.front();
```

```
73            buffer.pop_front();
74            delete del_buf;
75            ++ min_ack_expected;
76          }
77          while (buffer.size() > 0 && (next_frame - min_ack_expected) < win_size_max) {
78            buffer_element* pending = buffer.at(next_frame - min_ack_expected);
79            next_frame = ntohl(pending->payload->head.seq) + 1;
80            SendFRAMEPacket((uchar*)pending->payload, pending->size);
81          }
82          break;
83        }
84        case MSG_TYPE_TIMEOUT: {
85          unsigned int seq = *(unsigned int*) pBuffer;
86          unsigned int resend_size = 0;
87          for (std::deque<buffer_element*>::iterator it = buffer.begin();
88              it != buffer.end(); ++ it) {
89            SendFRAMEPacket((uchar*)(*it)->payload, (*it)->size);
90            ++ resend_size;
91            next_frame = ntohl((*it)->payload->head.seq) + 1;
92            if (resend_size == win_size_max) break;
93          }
94          break;
95        }
96      }
97      return 0;
98    }
99    frame* make_frame_copy(frame* src, int size) {
100     frame* ret_val = new frame();
101     std::memcpy(ret_val, src, (unsigned int)size);
102     return ret_val;
103   }
104 }
105 stop_and_wait_window(WINDOW_SIZE_STOP_WAIT),
106 back_n_frame_window(WINDOW_SIZE_BACK_N_FRAME);
107
108 /*
109 * 停等协议测试函数
110 */
111 int stud_slide_window_stop_and_wait(char *pBuffer, int bufferSize, UINT8 messageType)
112 {
113   stop_and_wait_window.event(pBuffer, bufferSize, messageType);
114   return 0;
115 }
116
117 /*
118 * 回退帧测试函数n
119 */
```

```
120 int stud_slide_window_back_n_frame(char *pBuffer, int bufferSize, UINT8 messageType)
121 {
122   back_n_frame_window.event(pBuffer, bufferSize, messageType);
123   return 0;
124 }
125
126 /*
127 * 选择性重传测试函数
128 */
129 int stud_slide_window_choice_frame_resend(char *pBuffer, int bufferSize, UINT8 messageType)
130 {
131   return 0;
132 }
```