网 络 协 议 分 析

课 程 实 验 报 告

姓名：　　叶剑飞

学号：　　1005020201

班级：　　10 网络二班

指导老师：　　文宏

湖南科技大学计算机科学与工程学院

2013 年 6 月 16 日

目录

1 Wireshark 实验

1.1 ping 的抓包

ping 是一种计算机网络管理工具，用于测试数据包能否通过 IP 协议到达特定主机，也用于测试消息传递的往返延时。ping 的协议为“网际控制报文协议”（ICMP）。向服务器端发包，并等待 ICMP 回复。

ICMP 数据包，作为 IP 数据包的正文来传送。IP 数据包分为 IPv4 和 IPv6 两种。IPv4 的首部是四位的 IP 版本号（version）、四位首部长度（length of header）、一个字节的服务类型（type of service），两个字节的总长度（total length）、两个字节的标识（identification）、一个字节的标志（flag）和三个字节的片偏移量（offset），接着是四个字节的 IPv4 源地址、四个字节的 IPv4 目地地址。IPv4 首部之后，是 IPv4 正文部分，即 ICMP 数据包。ICMP 数据包也分为 ICMP 首部和 ICMP 正文。ICMP 数据包的首部有一个字节的 ICMP 类型（ICMP type）、一个字节的类型代码（code of type）、两个字节的校验和（checksum）、两个字节的标识（identification）、两个字节的序号（sequence）。后面还有一些保留位，有时用来放时间戳（timestamp）等数据。

下面给出 Wireshark 捕获到的 ping 命令收发的数据包。

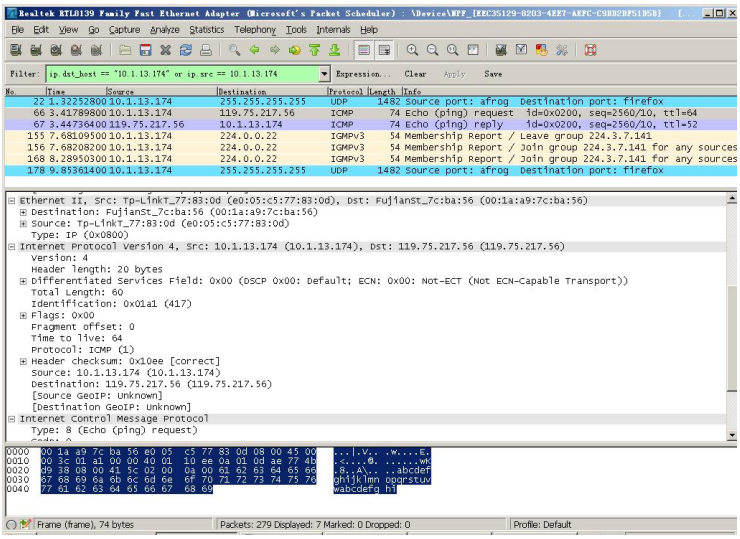

图 1: ping 命令发出的 ICMP 请求包

图1为 ping 命令发出的 ICMP 请求包。从中我们看到，ICMP 包的首部，“类型”中的数据为 8，表示这个包为回送请求。

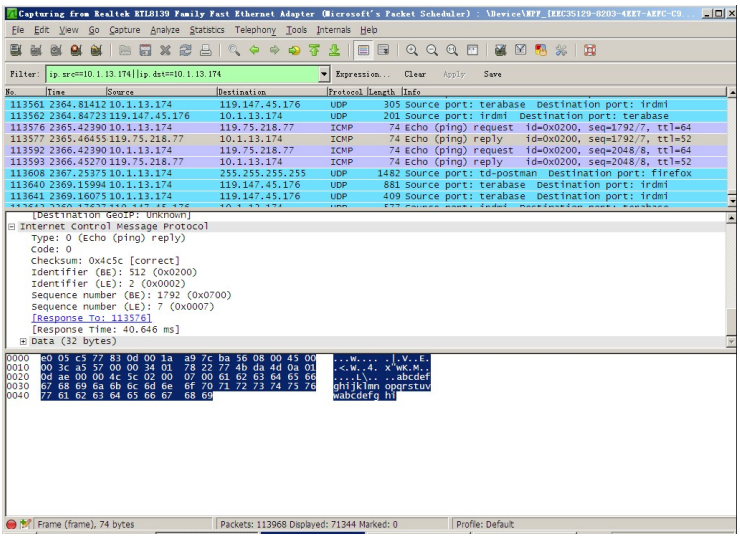

图 2: ping 命令收到的 ICMP 回应包

图2为 ping 命令接收到的 ICMP 回应包。从中我们可以看到，服务器端回馈的 ICMP 包，“类型”填的是 0，表示这个包为回送应答。

1.2 UDP 协议通讯的抓包

UDP 协议，即“用户数据报协议”（User Datagram Protocol），是因特网协议簇的核心成员之一。使用 UDP，计算机之间可以在先前无需预先通告或建立连接的情况下，用 IP 协议发送数据报。

UDP 使用的仅仅是协议机制的一些简单传输模型。它没有握手会话，所以它是不可靠的。UDP 首部字段由 4 个部分组成，其中两个是可选的。各 16 位的来源端口和目的端口用来标记发送和接受的应用进程。因为 UDP 不需要应答，所以来源端口是可选的，如果来源端口不用，那么置为零。在目的端口后面是长度固定的以字节为单位的长度域，用来指定 UDP 数据报包括数据部分的长度，长度最小值为 8 个字节。首部剩下的 16 位是用来对首部和数据部分一起做校验和（checksum）的，这部分是可选的，但在实际应用中一般都使用这一功能。

以下两张图片是 QQ 聊天时的的一些抓包。

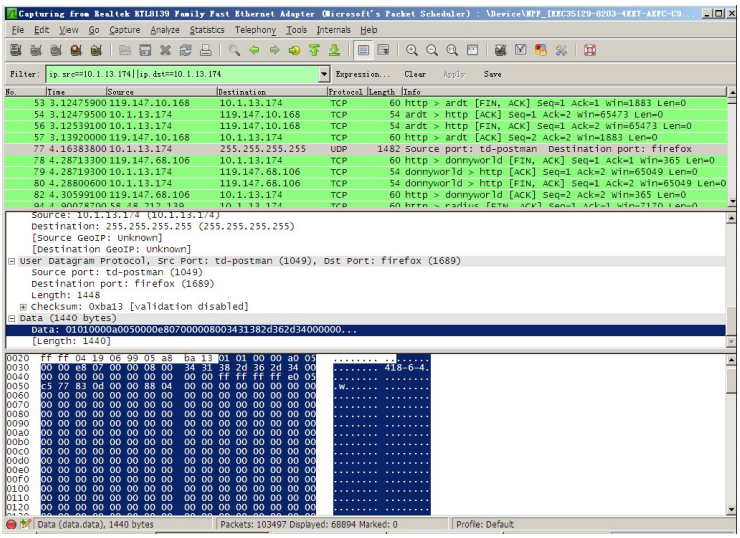

图 3：UDP 协议广播

图3是 QQ 发出广播消息。源端口号是 1049，目的端口号是 1689。可以看出，发送的数据中有本地计算机的计算机名。后面是大量的空字符。然后后面又有一一点信息。不过传送的都是腾讯公司自定格式的一些数据，我们也不知道这些数据具体是什么意思。

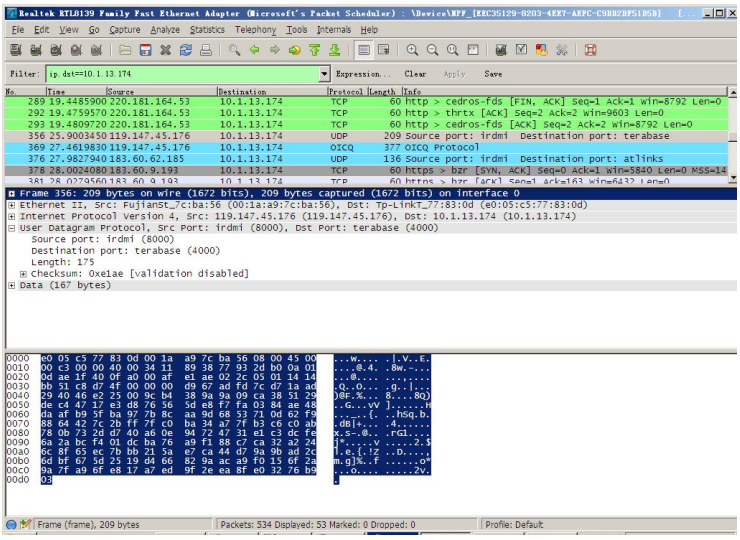

图 4：UDP 协议单播

图4是服务器发给 QQ 的 UDP 报文。源 IP 地址是 119.147.45.176（腾讯公司服务器机之一，其域名为 sz5.tencent.com），源端口号是 8000，目的端口号是 4000。传送 UDP 的正文是一些腾讯公司自己规定的一些格式的数据。

1.3 SSL/TLS 协议通讯的抓包

SSL 协议，即“安全套接字层”（Secure Sockets Layer）是一种安全协议。该协议是网景公司（Netscape）在推出 Netscape 浏览器首版的同时提出的，目的是为网络通信提供安全及数据完整性。该协议在传输层对网络连接进行加密。后来 ISO 将其规范为 TLS 协议，即“传输层安全”（Transport Layer Security）。

SSL/TLS 协议采用了公钥加密技术，以保证两个应用程序间通信的保密性，客户与服务器应用之间的通信不被攻击者窃听。该协议目前已成为互联网上保密通讯的工业标准。现行 Web 浏览器亦普遍将应用层的 HTTP 协议和传输层的 SSL/TLS 协议相结合，即 HTTPS 协议，从而实现安全的 Web 通信。

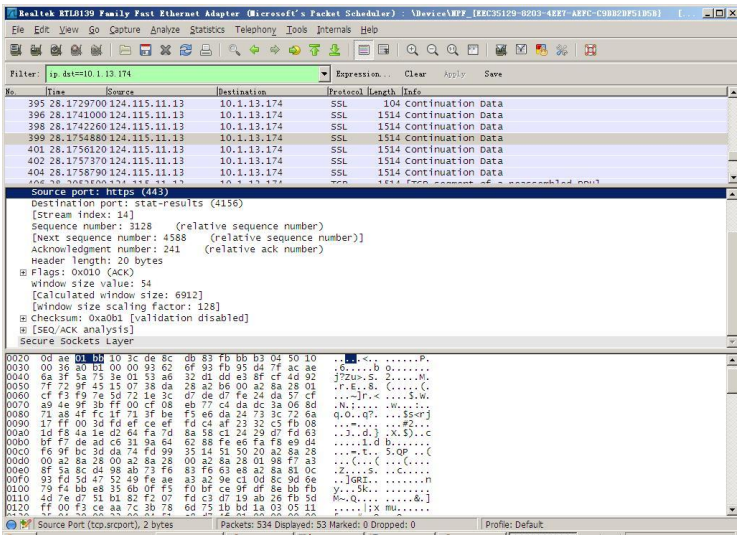

图 5: SSL 协议

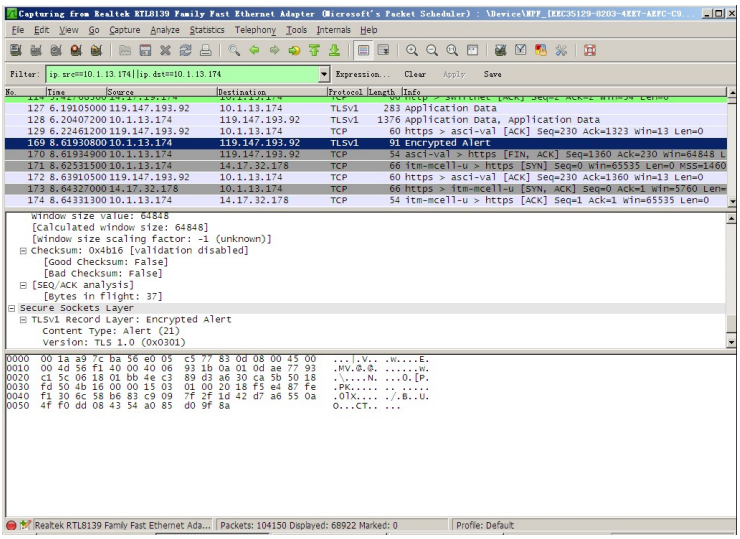

图 6: TLS 协议

图5显示的是 SSL 协议。图6显示的是 TLS 协议。它们都自底而上有着常规的物理层数据、数据链路层数据，网络层也有 IP 协议，传输层同样有着常规的 TCP 协议的数据。在 TCP 协议之上，用的是 SSL/TLS 协议。数据被加密过，无法解读其应用层协议数据。

图5是访问谷歌 https 主页 https://www.google.com.hk/ 时，从谷歌服务器机传来的加密数据。服务器端口号是 443，即 HTTPS 协议的默认端口号。客户端浏览器使用 4145 端口接收。由于传输层用的是 SSL 协议，所以 Wireshark 无法解读其应用层的 HTTP 协议的数据。

1.4 OICQ 协议的抓包

OICQ 协议，是腾讯公司自创的一种应用层协议，运行于 UDP 协议之上。用于传送一些 QQ 的控制性信息，例如好友列表、在线状态等。从 Wireshark 对该协议的解析中，我们看到该协议中指明了接收该信息的 QQ 号，和控制性信息的具体内容。

从图7中可以看出，在 UDP 首部中，源端口号为 4000，目的端口号为 8000。在 OICQ 报文中，标志位为 0x01，版本号为 0x2c05，发送的信息类型是签名操作。

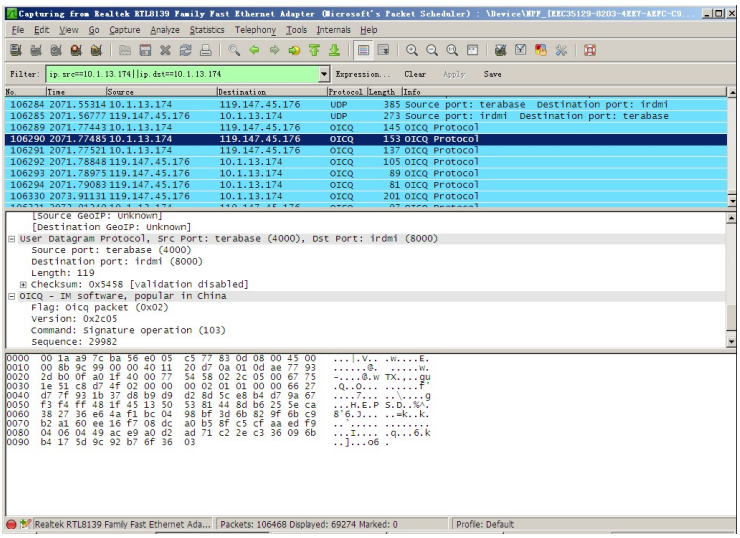

图 7: OICQ 协议

2　网络协议代码分析

2.1　选择重传协议

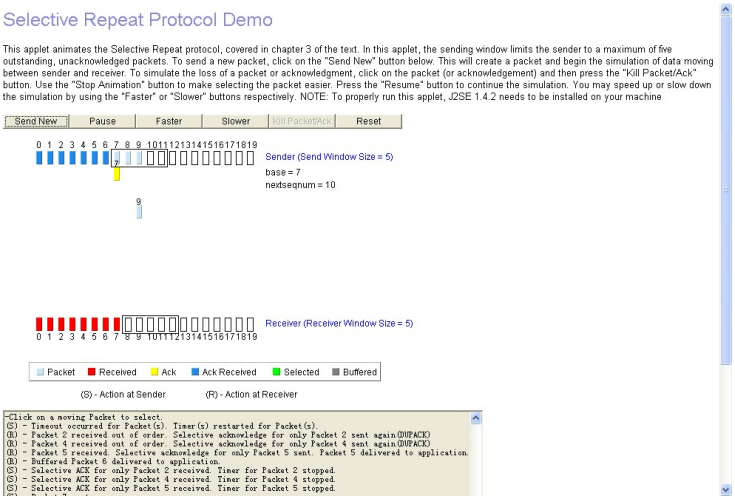

图 8: 选择重传协议演示

如图8所示，该选择重传协议用的发送方协议是：发送方发送每一个帧，都等待它的 ACK 帧，凡是收到 ACK 帧的，就做上已收到 ACK 帧的标记。从第一个窗口滑动到第一个没有 ACK 帧标记的位置。如果没有 ACK 标记的帧超时并没收到 ACK 帧的重发。接收方协议是：接收方每收到的一个帧，都回发一个 ACK 的帧。如果未收到过此帧，则保存此帧并后标记已收到此帧。窗口滑动到从第未标记为已收帧的位置。

```
// user pressed the send new button check if we can send a new Packet
if (cmd == "rdt" && nextseqsum < base + window_len) {
    // create our new Packet in the sender array
    sender[nextseqsum] = new SelectiveRepeatPacket(true, pack_height + ADVANCE_PACKET,nextseqsum);
    // tell user the Packet was successfully created and sent
    output.append("(S) - Packet " + nextseqsum + " sent\n");
    // simulate our per Packet timers
    output.append("(S) - Timer started for Packet " + nextseqsum + "\n");
    if (base == nextseqsum) // i.e. the window is empty and new data is
        // comming in
        {
            // start the timer thread for timeout processing
            if (timerThread == null)
                timerThread = new Thread(this);
            timerSleep = true;
            timerThread.start();
        }

    repaint();
    nextseqsum++;
    if (nextseqsum == base + window_len)
        send.setEnabled(false);
    start();
}
```

图 9: 单击 Send New 按钮后，将运行的代码

```
private void retransmitOutstandingPackets() {
    int retransmitPacket = 0;
    // after the timerThread wakes up process the Packets in sender
    // array from the base of our window (the leftmost edge)
    for (int n = base; n < base + window_len; n++)
        if (sender[n] != null)
            if (!sender[n].acknowledged && !sender[n].buffered) {
                sender[n].on_way = true;
                sender[n].Packet_ack = true;
                sender[n].Packet_pos = pack_height + 5;
                retransmitPacket++;
            } else if (!sender[n].acknowledged && sender[n].buffered) {
                sender[n].on_way = true;
                sender[n].Packet_ack = true;
                sender[n].Packet_pos = pack_height + 5;
                retransmitPacket++;
            }
    timerSleep = true;
    if (gbnThread == null) {
        gbnThread = new Thread(this);
        gbnThread.start();
    }
    if (retransmitPacket == 0) {
        timerThread = null;
    } else
        output.append("(S) - Timeout occurred for Packet(s). Timer(s) restarted for Packet(s). \n");

}
```

图 10: 超时重传代码

```
void deliverBuffer(int PacketNumber) {
    int j = 0;

    // Find our first buffered Packet in our array
    while (j < PacketNumber) {
        // error - all Packets up to PacketNumber should be created
        // if not something has gone horribly wrong
        if (sender[j] == null)
            return;
        // if Packet is ackd everythings fine keep looping
        else if (sender[j].acknowledged) {
            sender[j].buffered = false;
            j++;
            // else it must be buffered or in transmission stop here
            // this is our first possible buffered Packet
        } else
            break;
    }
    // above loop stops on last acked packet + 1
    // adjust count to make sure we start check at appropriate count
    // test > 0 to prevent index out of bounds
    if (j > 0)
        j--;
    for (int k = j; k < total_Packet; k++) {
        // prevent indexing out of bounds
        if (sender[k] == null)
            break;
        // if packet is buffered deliver to application and advance window
        else if (sender[k].buffered) {
            sender[k].buffered = false;
            // sender[k].acknowledged = true;
            output.append("(R) - Buffered Packet " + k + " delivered to application.\n");

            // if this packet is ack'd already advance
        } else if (sender[k].acknowledged) {
            sender[k].acknowledged = true;
            sender[k].buffered = false;
        } else if (!sender[k].Packet_ack) {
            sender[k].buffered = false;
            // if Packet is buffered deliver to application
            // and increment receiver window
        } else
            break;
    }
```

图 11: 发送报文的代码

2.2 IP 分段

输入的三个参数分别是数据报长度（包括 IP 首部 20 字节）、最大传输单元（MTU）、数据报 ID。最大传输单元和数据报大小必须大于 30，并且所有输入的值都必须小于 $2^{16} - 1$。输入完毕后，按下“计算”（Calculate）键，程序就会对其进行分片。例如下面的图 12，数据报大小 70 字节，最大传输单元为 30 字节，然后数据就被分成了三片，第一个是 20 字节的信息和 20 字节的数据，第二个帧是 20 字节的信息和 20 字节的数据，第三帧是 10 字节的信息和 30 字节的数据报。这样一个包就被拆成了三帧来发送。

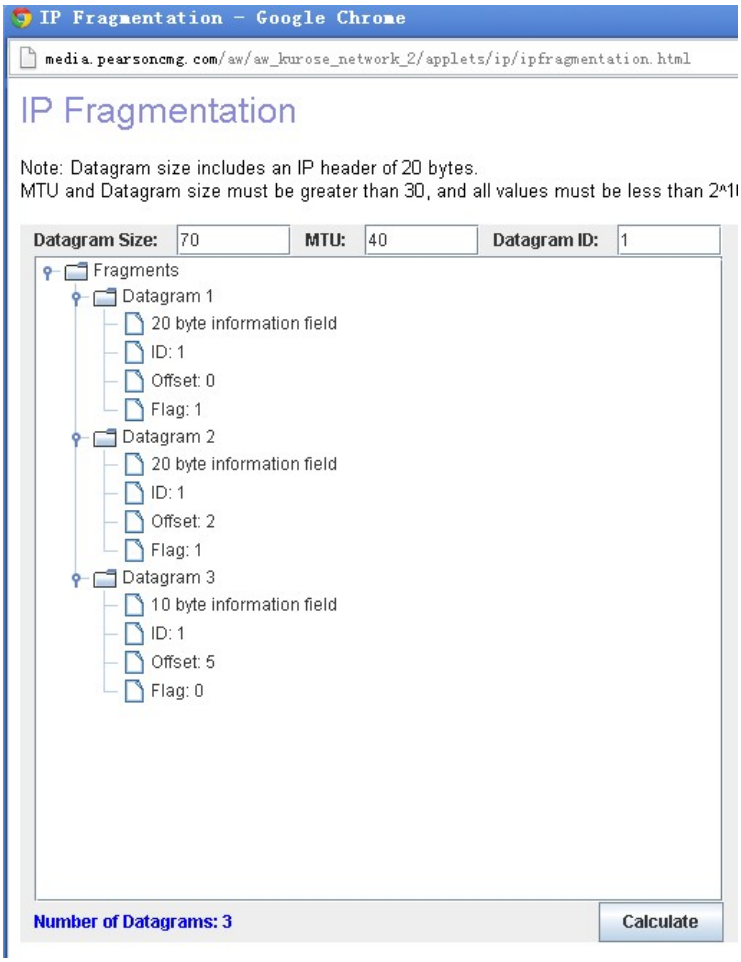

图 12: IP 分段演示

2.3 回退 N（Go-Back-N）协议

回退 N（Go-Back-N）协议，是数据链路层的网络协议之一。接收方每次收到一帧后，都只发送未收到的第一帧的前一帧的 ACK 包，并把接收窗口移到这一帧的位置。发送方每发一帧，都要计时。超时还未收到 ACK 包的，即重发。每次收到 ACK 包后，都将发送窗口移到 ACK 包确认帧的下一帧。

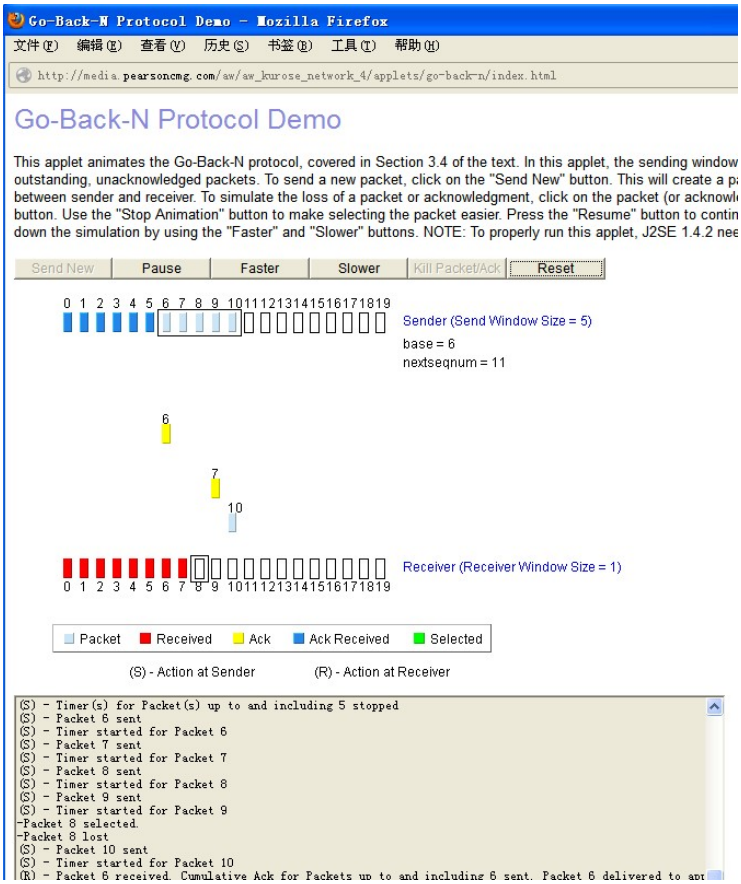

图 13: 回退 N（Go-Back-N）协议演示

图14为超时重传的代码。定位代码在15的位置。

```
private void retransmitOutstandingPackets() {
   // after the timerThread wakes up process the packets in sender
   // array from the base of our window (the leftmost edge)

   // up to the last packet send
   for (int n = (base == 0) ? 0 : base - 1; n < base + window_len; n++) {
      if (sender[n] != null)
         if (!sender[n].acknowledged) {
            sender[n].on_way = true;
            sender[n].packet_ack = true;
            sender[n].packet_pos = pack_height + ADVANCE_PACKET;
            sender[n].ackFor = n;
         }
      timerSleep = true;

      if (gbnThread == null) {
      gbnThread = new Thread(this);
      gbnThread.start();
      }

   }// end for
   //test for border case -- needs cleanup

   for(int i = base; i < total_packet; i++)
       if(sender[i].acknowledged == false)
       {
               output.append("(S) - Timeout occurred for Packet " + (i) + ". \r
               break;
       }
   output.append("(S) - All outstanding Packet(s) from " + base + " to " + (nex
}
```

图 14: 回退 N 协议超时代码

```
public void simGoBackN(int i) {
   // set all packets in the sender array up to index i (our ack that just
   // arrived) to acknowledged per go back n specs.
   for (int n = 0; n <= i; n++) {
      sender[n].acknowledged = true;
   }
   // if our packet was selected clear the selection bit.
   if (i == selected) {
      selected = DESELECTED;
      kill.setEnabled(false);
   }

   timerThread = null; // resetting timer thread
   // increment our base value to reflect the new ack we just received
   if (i + window_len < total_packet)
      base = i + 1;

   // if we have room in our window allow the user to send a new packet
   if (nextseq < base + window_len)
      send.setEnabled(true);

   if (base != nextseq) {
      // need to test to make sure our lastKnownSucPacket + 1 does not
      // throw an index out of bounds exception
      if (lastKnownSucPacket < (total_packet - 1))
         if (sender[lastKnownSucPacket + 1] != null && i != lastKnownSucPacl
            output.append("(S) -Timer still running for Packet " + (lastKnou
         else if (sender[lastKnownSucPacket + 1] != null)
            output.append("(S) -Timer still running for Packet " + (lastKnou
      timerThread = new Thread(this);
      timerSleep = true;
      timerThread.start();
   } else
      // set out of order to false in order to control the last known
      // packet received
      sender[i].out_of_order = false;
}
```

图 15: 回退 N 协议定位代码