实 验 五　使用 TCP 的远程桌面

5.1　实验目的

TCP 网络程序包括客户端与服务端程序，通信双方实现数据传输的基础上还要根据数据内容互相协作，程序执行逻辑相比非网络程序调试更为复杂，工作线程还需与界面线程实现并发与同步控制。远程桌面项目在 TCP 通信流程基础上实现自定义数据包，是一种网络应用层协议实现。

5.2　TCP 通信模型

5.2.1　TCP 协议介绍

TCP(Transmission Control Protocol) 传输控制协议 TCP 是一种面向连接的、可靠的、基于字节流的运输层（Transport layer）通信协议，提供全双工通信。TCP 在传输性能方面的要求是以复杂的算法为代价的，TCP 把数据流分割成适当长度的报文段，TCP 将数据包给定序号，序号也保证了传送到接收端实体的包的按序接收，然后接收端实体对已成功收到的字节发回一个相应的确认 (ACK)；如果发送端实体在合理的往返时延 (RTT) 内未收到确认，那么对应的数据（假设丢失了）将会被重传以此保证不发生丢包现象。TCP 数据包格式比较复杂，它采用一个校验和函数来检验数据是否有错误，在发送和接收时都要计算校验和，网络包不易被伪造具有一定的安全性。TCP 协议通过三个报文段完成连接的建立，这个过程称为三次握手 (three-way handshake)，而终止一个连接要经过四次握手，这是由 TCP 的半关闭 (half-close) 特性决定的，有关 TCP 算法的原理介绍，读者可参考计算机网络教材。

Windows 的驱动程序实现 TCP 协议，系统提供给用户的是套接字接口函数，TCP 的三次握手或数据包编号确认等过程无需用户直接控制，用户按流程调用套接字接口函数实现数据的传输。TCP 通信是一种客户端/服务器非对等模式，在程序中服务端的流程主要包括下面几步：

1. 创建 Socket 对象用于监听；
2. 使用 bind 方法对 Socket 对象进行端口绑定；
3. 使用 listen 方法执行监听；
4. Accepte 方法接收一个连接请求，并创建一个新 Socket 对象用于数据传输；
5. 新创建的 Socket 对象执行数据读写 read/write 实现数据传输；
6. 通信完成后使用 close 方法关闭与客户端通信的 Socket 对象；
7. 监听的 socket 对象使用 close 方法关闭；

客户端流程较简单，步骤如下：

1. 创建用于通信的 Socket 对象；

2. 使用 Socket 对象 Connect 方法连接服务器端的监听端口；
3. 使用 Socket 对象读写 read/write 方法进行数据；
4. 通信完成后使用 Socket 对象的 close 方法关闭；

TCP 的通信模式中设计服务端与客户端一对多的模式，即允许多个客户端连接到同一个服务端，服务端同时与多个客户端通信。客户端线程创建和管理 Socket 对象流程比较直观，执行 Connect 命令连接服务端指定端口，成功后即可与服务器端进行数据读写操作。实际连接的客户端数目是不确定的，服务端通过监听的方法管理多个客户端的连接与断开。服务端创建一个 Socket 对象并在指定端口执行监听任务，监听状态的 Socket 对象用于接收客户端的连接请求，服务端线程收到客户端连接请求时系统通信模块会创建一个新的 Socket 对象，负责网络监听的 Socket 对象在通信过程中保持不变以便继续接收新的客户连接。

新创建的 Socket 对象分配到新的网络端口，它负责与客户端配对执行通信任务，服务端采用新建线程管理新 Socket 对象以完成数据通信功能，而 WinSock 中的 IOCP(完成端口) 方式采用线程池方式支持大量网络连接，能达到极高的网络通信效率。客户端与服务端的线程对象结合 TCP 通信流程模式可由图5-1进行描绘，图中描述的服务端新建线程管理新 Socket 对象，它实现的是连接数不多的 TCP 通信。

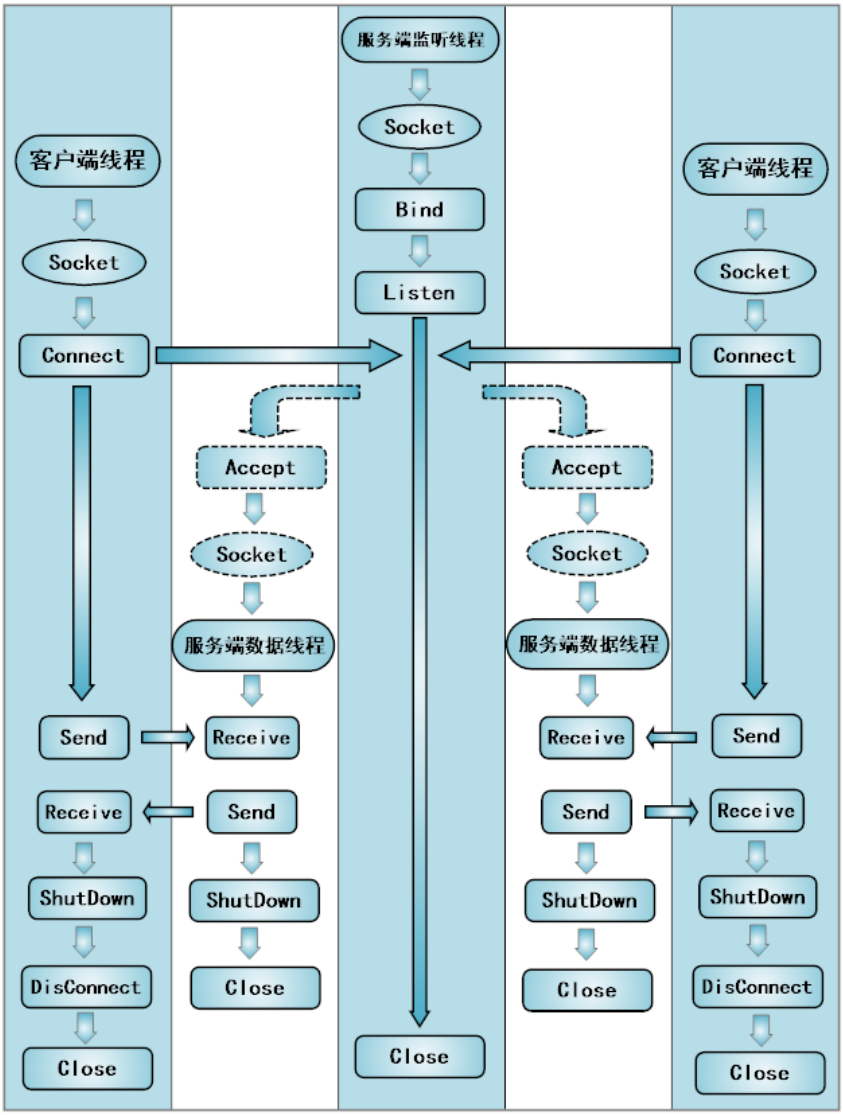

图 5-1 TCP 通信流程示意图

图 5-1表示有两个客户端连接同一个服务端，通信中的每个 socket 对象对应一个管理线程，包括监听线程一共有五个线程对象，每列代表一个线程对象的执行流程。图 5-1不仅表示了单个线程内操作步骤的时序情况，还表明了客户端与服务端关键步骤的时序情况。服务端 socket 对象必须先执行 bind 操作，然后是 listen 操作，客户端的 connect 操作必须在服务端的 listen 操作完成后执行，否则发生连接失败。服务端的 accept 操作是服务端编程的难点，它具有两个特殊的地方：第一它是一个被动操作，物理网络上具有真实的连接请求时它才完成一次执行，无连接到来时它处于阻塞状态；第二程序员需要控制 accept 方法的多次执行，以保证每个网络连接请求，在充许 accept 的过程中，执行 listen 操作的 socket 对象不能被终止，直到网络连接达到操作系统的最大连接数限制或被终止监听。

网络编程者容易混淆网络协议的端口号与 socket 对象的端口号，客户端的 socket 对象在创建时具有自己的端口号，由机器自动分配，例如 5621，发起连接时指定的目标端口是服务端负责 listen 的端口，例如 80 端口。当服务端 accept 成功执行后，服务端创建的 socket 对象使用的是新的网络端口（由系统随机分配，例如 3382），在通信过程中，客户端的 5621 与服务端的 3382 形成 socket 通信对，匹配的这对 socket 对象完成网络通信过程。在计算机网络协议中，TCP 协议约定的通用端口都是指服务端的监听端口，而 TCP 协议中进行通信的每个 socket 对象端口号必定是唯一的。如果程序设计者认为多个客户端 socket 对象与服务端同一个端口号进行通信是不正确的的，也很难编写出有用的程序。

服务端成功执行一次 accept 操作后，新的 socket 对象准备与客户端进行实际数据传输，这时比较方便的是创建一个工作线程来使用和管理这个 socket 对象，工作线程则要依赖数据传输的实际情况执行复杂的逻辑任务，例如对数据完整性检验，加密与解密，甚至把数据送到游戏引擎等。工作线程的逻辑控制与数据传输有很大关系，例如它会根据传送的数据内容结束通信。服务端程序退出监听前要等待所有的 socket 对象都正常关闭，这样才能保证数据传输不会丢失或出错，而程序的主线程还要同步工作线程正常结束，这都需要应用 windows 的事件同步机制。

5.2.2 TCP 中的 Nagle 算法

网络通信协议 TCP 实现数据可靠有序传输，它在实现时考虑了一种小包问题，小包问题描述的是应用程序可能多次产生极小长度的数据，例如只有一个字节，而数据包由于经过两层或三层的封装后，网络协议数据包头将占用 40 字节，传输这一个有用字节需要使用额外的 40 字节的包头，这样传输的带宽利用率会很低，造成网络资源的浪费。TCP 协议实现中增加 Nagle 算法以图提高带宽利用率，算法思想是把要发送的数据先送到缓存中，直到攒到一定数量或经过一段时间间隔再将数据一并发送出去。Nagle 算法有效提高了网络带宽利用率，多数情况下，socket 对象在新创建时都开启了 Nagle 算法。

虽然能提高网络利用率，Nagle 算法却带来了一个小小的麻烦，假如通信一方发出了多个问题，而另一方对每个问题的回答又是很简短的，在使用 Nagle 算法情况下，多个简短的回答会被拼接在一起被一次发送，这样提问方虽然提出了多个问题，但只收到一个回答，这将造成提问者的困惑；再比如在游戏中有时只须传输一两字节的信息，Nagle 算法却把这么短的数据暂存在缓冲区而不是直接发送，游戏就失去了实时性而无法继续了。Nagle 算法在实际中将会造成数据发送的不确定性，比如究竟拼接后的数据为多长，缓冲数据要等待多长时间都是不确定的。TCP 这种以字节流形式无消息边界的协议给程序带来不确定性，就连 send 方法与 receive

方法都无法一一对应起来。在程序中通过事先设置 socket 的参数属性来关闭 Nagle 算法的执行，在 C# 语言中关闭 Nagle 算法可使用下面的方法：

sock.SetSocketOption(SocketOptionLevel.Socket, SocketOptionName.NoDelay, 1);

关闭 Nagle 算法虽然牺牲了网络的传输效率，但能够获得较好的交互性和实时性，在有些场合中是必要的。

5.3 远程桌面程序实现

5.3.1 任务分析与自定义数据包

本实验项目实现一个较简单的远程桌面功能，有两个窗体应用程序，客户端项目命名为 rd_c，服务端项目命名为 rd_s；客户端运行 TCP 的客户端流程，主动向服务端连接，将屏幕内容复制为图片后发送到服务端，服务端运行 TCP 的服务端流程，负责启动监听，接收连接请求，接收客户端发来的图片数据，显示图片模拟简单的远程桌面功能，程序任务示意可参考图5-2。通信双方使用.NET 平台提供的 Socket 类 Send 和 Receive 方法用于发送与接收字节序列，MemoryStream 内存流的存取速度优于文件流，用它暂存图片数据。项目中的客户端与服务端程序都采用窗体项目，主窗体负责与用户直接交互，工作线程实现通过网络发送数据和接收数据，工作线程与窗体线程采用事件机制并结合消息机制互相通信。

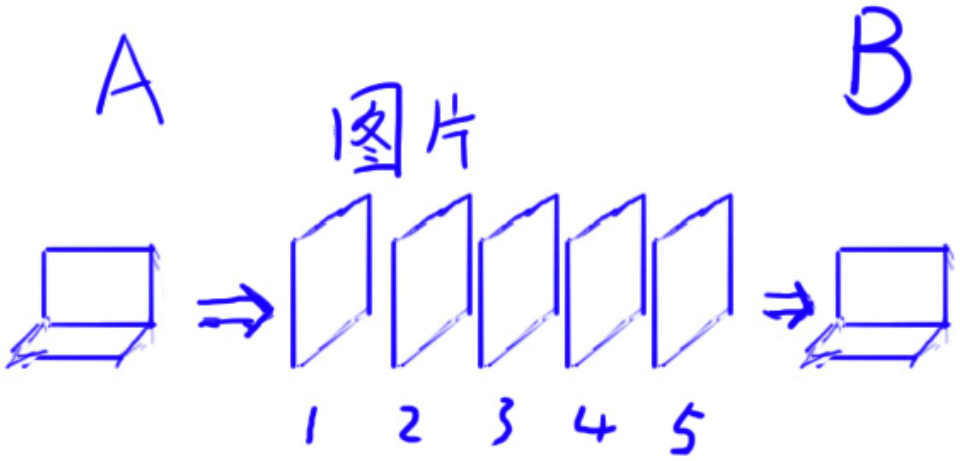

图 5-2 远程桌面程序任务示意图

考虑程序要解决的问题：

1. 屏幕自动截屏与数据循环发送；
2. 数据接收与图片复原；
3. 通信双方传输图片如何保证发送和接收速度一致；
4. 约定单个图片发送的开始与结束；
5. 终止的双方通信；

程序仅仅能够收发数据显然是不够的，还要对数据有意义说明才能解决面临的问题，根据在本项目的程序需要，设计表5-1表示的的自定义网络数据包，在要发送的数据前添加格式头信息。自定义数据包可解决程序多个问题，如规定图片传输的开始与结束，通过发送者等待接收

者的包确认实现双方网络读取速率一致，还能实现终止双方通信的控制命令。

表 5-1 自定义数据包

格式头数值	数据意义	操作响应
10	新的一屏图像开始	内存区准备接收新数据
20	屏幕图像数据	向内存区写入数据
30	一屏图像数据结束	完成图像接收，准备显示
50	客户端终止连接	断开与客户端连接

5.3.2 客户端实现

远程桌面的客户端有一个工作线程，线程代码由 thread_capture_send 定义，它将定时捕获屏幕图片，并将图片发送出去。Graphics 类封装一个 GDI+ 绘图图面，Graphics 类的方法 CopyFromScreen 能够获取屏幕位图数据，执行颜色数据从屏幕到 Graphics 的绘图图面的位块传输。先创建一个 Bitmap 位图对象，用于保存位图数据，基于 Bitmap 位图对象初始化 Graphics 对象。下面是具体实现。

```
static void thread_capture_send()
{
  //连接服务器
  IPEndPoint remoteEP = new IPEndPoint(IPAddress.Parse("192.168.1.100"), Int32.Parse("8133"));
  // Create a TCP/IP socket.
  Socket client_sock = new Socket(AddressFamily.InterNetwork,
    SocketType.Stream, ProtocolType.Tcp);
  client_sock.SetSocketOption(SocketOptionLevel.Socket, SocketOptionName.NoDelay, 1);
  try
  {
    client_sock.Blocking = true;
    client_sock.Connect(remoteEP);
    try
    {
      //定时屏幕保存，
      int s_wid = Screen.PrimaryScreen.Bounds.Width;
      int s_height = Screen.PrimaryScreen.Bounds.Height;
      Bitmap b_1 = new Bitmap(s_wid, s_height);
      Graphics g_ = Graphics.FromImage(b_1);
      long each_screen_data_len;
      byte[] SendDataBuffer = new byte[1024];//数据发送缓存
      byte[] ReadDataBuffer = new byte[1024];//数据接收缓存
      byte[] b_data_len;
      int tran_count = 0;
      int ms_data_read_count;
```

```
int numberOfByteRead;
int replay_code;
//拷屏部分应用控制来循环处理
do
{
  g_.CopyFromScreen(0, 0, 0, 0, new Size(s_wid, s_height));
  //为了响应迅速, 使用 MemoryStream 内存为数据存区
  //设置流为最开始的位置
  ms_cap_pic.Seek(0, SeekOrigin.Begin);
  b_1.Save(ms_cap_pic, System.Drawing.Imaging.ImageFormat.Jpeg);
  //数据量大小得到 --通过流的位置来决定要发送数据的大小
  each_screen_data_len = ms_cap_pic.Position;
  //所有的数据全部转化成字节后，拼装在一起发送出去。
  //1 将数组值清空
  Array.Clear(SendDataBuffer, 0, 1024);
  //将数据命令数转化为字节数组
  b_data_len = BitConverter.GetBytes((int)10);//10 代表开始发送一屏
  Array.Copy(b_data_len, 0, SendDataBuffer, 0, 4);
  client_sock.Send(SendDataBuffer, 1024, SocketFlags.None);
  //等待服务器响应代码
  Array.Clear(ReadDataBuffer, 0, 1024);
  numberOfByteRead = client_sock.Receive(ReadDataBuffer, 1024, SocketFlags.None);
  replay_code = BitConverter.ToInt32(ReadDataBuffer, 0);
  tran_count = 0;
  ms_cap_pic.Seek(0, SeekOrigin.Begin);
  do
  {
    b_data_len = BitConverter.GetBytes((int)20);//20 代表屏幕图像数据
    Array.Copy(b_data_len, 0, SendDataBuffer, 0, 4);
    //read from memory to the buffer
    ms_data_read_count = ms_cap_pic.Read(SendDataBuffer, 4, 1020);
    //write to network
    client_sock.Send(SendDataBuffer, ms_data_read_count + 4, SocketFlags.None);
    //等待服务器响应代码
    Array.Clear(ReadDataBuffer, 0, 1024);
    numberOfByteRead = client_sock.Receive(ReadDataBuffer, 1024, SocketFlags.None);
    replay_code = BitConverter.ToInt32(ReadDataBuffer, 0);
    tran_count += ms_data_read_count;
  } while (tran_count < each_screen_data_len);
  b_data_len = BitConverter.GetBytes((int)30);//30 代表图像数据发送完毕
```

```
            Array.Copy(b_data_len, 0, SendDataBuffer, 0, 4);
            client_sock.Send(SendDataBuffer, ms_data_read_count, SocketFlags.None);
            //等待服务器响应代码
            Array.Clear(ReadDataBuffer, 0, 1024);
            numberOfByteRead = client_sock.Receive(ReadDataBuffer, 1024, SocketFlags.None);
            replay_code = BitConverter.ToInt32(ReadDataBuffer, 0);
            //间隔一定时间
            Thread.Sleep(cap_period);
        } while (!terminate_capture.WaitOne(1));
        //1 将数组值清空
        Array.Clear(SendDataBuffer, 0, 1024);
        //发送命令数据 50
        b_data_len = BitConverter.GetBytes((int)50);//50 代表客户端退出
        Array.Copy(b_data_len, 0, SendDataBuffer, 0, 4);
        client_sock.Send(SendDataBuffer, ms_data_read_count, SocketFlags.None);
        //等待服务器响应代码
        Array.Clear(ReadDataBuffer, 0, 1024);
        numberOfByteRead = client_sock.Receive(ReadDataBuffer, 1024, SocketFlags.None);
        replay_code = BitConverter.ToInt32(ReadDataBuffer, 0);
        client_sock.Shutdown(SocketShutdown.Both);
        //拷屏部分应用控制来循环处理
      }
      catch (SocketException se3)
      {
        MessageBox.Show(" 客户端异常" + se3.Message);
      }
    }
    catch (SocketException se1)
    {
      MessageBox.Show("SocketException 客户端连接不到服务器呢" + se1.Message);
    }
    catch (Exception se2)
    {
      MessageBox.Show(" 客户端异常" + se2.Message);
    }
    //线程代码 –end
}
```

5.3.3 异步网络事件

使用套接字 (Socket) 时，一些操作的执行依赖网络状态，例如服务端的 accept 操作，没有连接请求时，它处于阻塞状态，当连接请求到达时返回新的 socket 对象；receive 方法的执行也比较类似，没有数据时它处于阻塞状态，有数据到达，函数返回接收到的字节数组。网络事件的发生时间是不确定的，套接字对这些操作提供了同步方式和异步方式，当线程调用的是同步方式时，例如使用 Socket.Receive 方法，则 Receive 方法将一直处于阻止状态，直到数据出现。同步方法容易理解和编写程序，异步方法有时更适应用程序需要。异步方式要先定义异步回调的执行函数，本项目中定义的是 AcceptCallback 函数，服务端线程调用 listen 方法后，使用 Socket.BeginAccept 方法，它将 AcceptCallback 函数绑定到 accept 网络事件，线程没有阻塞，它是异步方式，只要服务端处于监听状态任何时候有连接请求，这个处理函数就会自动执行。Socket.BeginAccept 方法比 Socket.Accept 方法灵活的地方是只需调用一次，当有客户连接 AcceptCallback 函数会自动重复执行，同步 accept 方法则要由程序员构造循环结构了。

网络通信中的 Socket 方法还有 Connect，Disconnect，Receive，ReceiveFrom，Send，SendTo 等都有对应的异步方法。

5.3.4 服务端实现

远程桌面程序服务端要略显复杂，它包括一个 listen 线程用于监听网络连接，当有客户端连接请求时将异步执行 Accept 方法，在 AcceptCallback 方法中得到新创建的 socket 对象与客户通信，这时启动接收数据的线程 thread_recv_display，线程 thread_capture_send 与 thread_recv_display 互相配合循环进行图片的发送和接收。监听线程 thread_listen 采用异步的 accept 方法，代码如下：

```
static void thread_listen()
{
    IPAddress[] host_ip = Dns.GetHostAddresses(Dns.GetHostName());
    S_Listen_sock = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
    LingerOption _lingerOption = new LingerOption(true, 3);
    S_Listen_sock.SetSocketOption(SocketOptionLevel.Socket, SocketOptionName.Linger, _lingerOption);
    S_Listen_sock.Blocking = false;//设定其为异步
    IPEndPoint host_end = new IPEndPoint(IPAddress.Parse("192.168.1.100"), Int32.Parse("8133"));
    User_Terminate_listen.Reset();
    S_Listen_sock.Bind(host_end);//开始绑定
    S_Listen_sock.Listen(3);//开始监听
    Accep_Object Ac_state = new Accep_Object();
    S_Listen_sock.BeginAccept(
        new AsyncCallback(AcceptCallback),
        Ac_state);
    SendMessage(main_wnd_handle, BEGIN_LISTEN, 100, 200);
    User_Terminate_listen.WaitOne();
```

```
    //关闭所有的子 socket，结束监听
    //不应该马上调用关闭，因为这会清空 S_Listen_sock 对象
    //将会迟些时候关闭
    SendMessage(main_wnd_handle, END_LISTEN, 100, 200);
  }
```

下面是回调函数 AcceptCallback 定义：

```
  //负责接收连接的回调函数
  public static void AcceptCallback(IAsyncResult ar)
  {
    //有新的客户端连接
    if (S_Listen_sock == null)
    { //原因比较复杂，暂不作解释
      S_client_sock = S_Listen_sock.EndAccept(ar);
      MessageBox.Show(" 新客户已经开始连接服务器");
    }
    else
    {
      //MessageBox.Show(" 新客户已经连接到服务器 or 服务端停止了监听。");
      S_client_sock = S_Listen_sock.EndAccept(ar);
      S_client_sock.Blocking = true;
      //每次新的 Client 到来则启动一个新的线程，利用新的 Socket 与客户交互
      ThreadStart clientWorkStart = new ThreadStart(thread_recv_display);
      Thread clientThread = new Thread(clientWorkStart);
      clientThread.IsBackground = true;
      clientThread.Start();
    }
  }
```

工作线程 thread_recv_display 的实现代码如下：

```
  static void thread_recv_display()
  {
  //线程流程利用 client_socket 接收图像信息
    try
    {
      S_client_sock.SetSocketOption(SocketOptionLevel.Socket, SocketOptionName.NoDelay,
1);
      byte[] ReceiveDataBuffer = new byte[1024];
      byte[] SendDataBuffer = new byte[1024];
      byte[] b_data_reply;
      int numberOfBytesRead;
      long data_cmd;
```

```
        bool bl_not_end_recv = true;
        do
        {
          //1. 获取网络数据
          numberOfBytesRead = S_client_sock.Receive(ReceiveDataBuffer, 1024, SocketFlags.None);
          //2. 获取格式命令头信息
          data_cmd = BitConverter.ToInt32(ReceiveDataBuffer, 0);
          b_data_reply = BitConverter.GetBytes((int)data_cmd + 1);
          //data_cmd+1 代表对 data_cmd 的响应
          Array.Clear(SendDataBuffer, 0, 1024);
          Array.Copy(b_data_reply, SendDataBuffer, 4);
          S_client_sock.Send(SendDataBuffer, 1024, SocketFlags.None);
          switch (data_cmd)
          {
            case 10: //新一屏数据到来，准备图像内存区
              ms_cap_pic.Seek(0, SeekOrigin.Begin);
              break;
            case 20: //从网络接收到的数据写入指定内存区
              //write to memory stream
              ms_cap_pic.Write(ReceiveDataBuffer, 4, numberOfBytesRead - 4);
              break;
            case 30: //屏幕数据发送完毕，复原出屏幕位图向主窗体发送消息，更新屏幕显示图
像
              //发送消息，更新屏幕图像
              bp_full = (Bitmap)Bitmap.FromStream(ms_cap_pic);
              SendMessage(main_wnd_handle, UPDATE_SCREEN, 100, 100);
              break;
            case 40: //
              break;
            case 50: //50 客户端退出, 通知主窗体复原
              SendMessage(main_wnd_handle, WINDOW_RESTORE, 100, 100);
              bl_not_end_recv = false;
              break;
          }
        } while (bl_not_end_recv);
      }
      catch (SocketException Se1)
      {
        MessageBox.Show("SocketException:" + Se1.Message);
      }
```

```
    catch (Exception Se2)
    {
      MessageBox.Show(Se2.Message);
    }
  }//thread_recv_display
```

工作线程 thread_recv_display 接收客户端发来的图片数据，接收过程由自定义协议包控制，工作线程在每次成功接收图片时向主窗体发送 UPDATE_SCREEN 消息，窗体则收到消息后进行图片的重绘，线程收到客户端发来结束消息后向主窗体发送 WINDOW_RESTORE 消息，主窗体将由全屏显示状态恢复为普通状态。主窗体的消息重载处理函数如下：

```
  protected override void DefWndProc(ref System.Windows.Forms.Message m)
  {
    switch (m.Msg)
    {
      case UPDATE_SCREEN:
        bp_screen = bp_full;
        //全屏显示一个桌面
        foreach (Control cc in this.Controls)
        {
          cc.Visible = false;
        }
        this.Invalidate();
        break;
      case BEGIN_LISTEN:
        label1.Text =" 开始服务";
        break;
      case END_LISTEN:
        label1.Text = " 终止服务";
        break;
      case WINDOW_RESTORE:
        label1.Text = " 用户中断发送屏幕";
        //进入普通界面
        foreach (Control cc in this.Controls)
        {
          cc.Visible = true;
        }
        //恢复时显示普通图
        bp_screen = bp_small;
        windows_is_full = false;
        this.FormBorderStyle = original_style;
        this.WindowState = original_state;
```

```
            break;
        default:
            base.DefWndProc(ref m);
            break;
    }
}
```

在窗体的 Paint 事件中添加下面的代码即可显示客户端的桌面图片：

```
this.BackgroundImage=bp_full;
```

5.4 程序说明

本程在实际运行过程中容易发生连接断开的现象，请同学们仔细分析原因。完整项目代码篇幅较长，本文档仅展示关键的线程代码，项目中的其它功能模块还包括屏幕的截图和图片全屏显示，工作线程与主窗体的通信，请读者参考完整源代码。

5.5 思考与作业

1. 调试运行客户端与服务端程序；
2. 在不同的机器上分别运行服务端与客户端，实现远程桌面功能；
3. 增加屏幕控制功能，服务端可将鼠标或键盘数据发回客户端，客户端收到数据后模拟键盘事件和鼠标事件实现远程控制；