上机作业 1：编写简单的客户-服务器程序 (实验报告)

1711342 李纪

2019 年 10 月 26 日

摘要

这是我的上机作业 1 的实验报告，主要介绍了我的**客户-服务器程序**的功能和大致实现。具体的实现过程已在上机课时向助教讲解，如有疑惑之处，请直接参看源码或联系本文作者。请老师查阅，谢谢。

关键字： MFC，客户 - 服务器程序，UDP

目录

1 实验的目的

本实验要求利用 CAsyncSocket 类编写一个简单的客户-服务器程序，客户与服务器之间使用数据报方式传送信息，服务器在收到客户发来的 Time 或 Date 请求后，利用本地的时间和日期分别进行响应。通过该编程实验，可以加深对客户服务模型的理解，学习简单的 socket 编程方法。

2 实验方案和方法的选择

TCP/IP 技术的核心部分是传输层（TCP[1]和 UDP[2]）、互联层（IP）和主机网络层，这 3 层通常在操作系统的内核中实现。为了使应用程序方便地调用内核中的功能操作系统常常提供编程界面（有时也叫程序员界面或应用编程界面），其中，socket（套接字）调用就是 TCP/IP 网络操作系统为网络程序开发提供的典型网络编程界面。

socket 分为数据报套接字（datagram sockets）和流式套接字（stream sockets）两种形式，其中，数据报方式使用 UDP 支持主机之间面向非连接、不可靠的信息传输；流方式使用 TCP，支持主机之间面向连接的、顺序的、可靠的全双工字节流传输。

常用的网络操作系统（如 Windows 操作系统、UNIX 操作系统、Linux 操作系统等）都支持 socket 网络编程接口 · 程序员可以利用 socket 界面使用 TCP/IP 互联网功能，完成主机之间的通信。Windows 网络操作系统提供的 socket 被称为 Windows Sockets API. 程序员可以直接调用这些 API 编写自己的网络应用程序。在 Microsoft Visual C++ 中，这些 socketAPI 被封装成 CAsyncSocket 类，程序员的网络编程更加方便。

CAsyncSocket 对 Windows sockets API 在比较低的级别上进行了封装，利用 CAsyncSocket 编制网络应用程序不但比较灵活，而且能够避免直接调用 Windows Sockets API 函数的烦琐工作。[1]

---

[1] **传输控制协议**（英语：**T**ransmission **C**ontrol **P**rotocol，缩写：**TCP**）是一种面向连接的、可靠的、基于字节流的传输层通信协议，由 IETF 的 RFC 793 定义。在简化的计算机网络 OSI 模型中，它完成第四层传输层所指定的功能。

[2] **用户数据报协议**（英语：**U**ser **D**atagram **P**rotocol，缩写：**UDP**；又称**用户数据包协议**）是一个简单的面向数据报的通信协议，位于 OSI 模型的传输层。该协议由 David P. Reed 在 1980 年设计且在 RFC 768 中被规范。在 TCP/IP 模型中，UDP 为网络层以上和应用层以下提供了一个简单的接口。UDP 只提供数据的不可靠传递，它一旦把应用程序发给网络层的数据发送出去，就不保留数据备份（所以 UDP 有时候也被认为是不可靠的数据报协议）。

3 实验过程

注意：为了进一步了解我的程序，建议先看程序**用户手册**。下面介绍关键代码段，请结合源代码来看。具体实现请直接参看源代码。

3.1 服务器的实现

```
void CReReServerDlg::OnBnClickedStart()
{
  if (!isListen)
  {
    // Port number
    CString temp;
    // 获取输入的端口号
    portText.GetWindowTextW(temp);
    // 如果未输入端口号，请重新输入
    if (temp == "")
    {
      MessageBox(_T("Port empty."));
      return;
    }
    if (protocolType == UDP)
    {
      // 生成 UDP Socket
      BOOL createFlag = usock.Create((UINT)(_ttoi(temp)), SOCK_DGRAM,
    FD_READ);
      // 生成失败会报错
      if (createFlag == 0)
      {
        MessageBox(_T("Socket create failed."));
        usock.Close();
        return;
      }
    }
    // TCP 未实现
    else if (protocolType == TCP)
    {
      // TODO: 在此添加 TCP 的代码
    }

    // 禁用输入端口控件
    portText.EnableWindow(false);
    // UDP Socket 开启状态设为 “开启”
    isListen = true;
```

```
    displayString(_T("Listening on: ") + temp + _T(". Protocol: ") + \
      (protocolType == TCP ? _T("TCP") : _T("UDP")) + _T(".\r\n"));
    startButton.SetWindowTextW(_T("Stop"));
  }
  else if (isListen == true)
  {
    if (protocolType == UDP)
    {
        // 关闭 socket
      usock.Close();
    }
    else if (protocolType == TCP)
    {
      // TODO: 在此添加 TCP 的代码
    }
    displayString(_T("Stop listening.\r\n"));
    startButton.SetWindowTextW(_T("Start"));
    // 激活端口输入控件
    portText.EnableWindow(true);
    // UDP Socket 开启状态设为 “关闭”
    isListen = false;
  }
}
```

Listing 1: 打开服务器的实现

Start 按钮用于打开服务器，生成新的 UDP Socket 实例。

```
int ServerUDP::reply(CString req, CString& ret)
{
  // 将接收到的请求一律转为小写
  req = req.MakeLower();
  // date 请求的实现
  if (req == _T("date"))
  {
    //获取当前时间日期
    CTime m_time = CTime::GetCurrentTime();
    //格式化时间
    ret = m_time.Format(_T("%x"));
    return 1;
  }
  // time 请求的实现
  if (req == _T("time"))
  {
    //获取当前时间日期
    CTime m_time = CTime::GetCurrentTime();
```

```
    //格式化时间
    ret = m_time.Format(_T("%X"));
    return 2;
  }
  // file [路径] 请求的实现
  if (req.Left(5) == _T("file "))
  {
    CString csFullPath = req.Mid(5);
    ret = _T("");
    CFileFind find;
    // 获取文件目录
    BOOL IsFind = find.FindFile(csFullPath + _T("/*"));
    while (IsFind)
    {
      IsFind = find.FindNextFile();
      if (find.IsDots()) {
        continue;
      }
      else {
        ret += (find.GetFileName() + _T("\r\n"));
      }
    }
    return 3;
  }
  // who are u 请求的实现
  if (req == _T("who are u"))
  {
    char hostname[40];
    // 获取主机名
    gethostname(hostname, sizeof(hostname));
    ret = (CString)hostname + _T("\r\n");
    return 4;
  }
  return 0;
}
```

Listing 2: 请求回复的实现

回复请求通过调用 ServerUDP 类中的 reply 函数实现。

```
void ServerUDP::OnReceive(int nErrorCode)
{
  // A buffer for the incoming data.
  char m_szBuffer[4096];
  // What ReceiveFrom Returns.
  int nRead;
```

```
CString host, strp;
UINT port;
// 获取接收信息
nRead = ReceiveFrom(m_szBuffer, sizeof(m_szBuffer), host, port,0);

// 按 ReceiveFrom 的返回值来判断如何继续
switch (nRead)
{
// 如果未发生错误， ReceiveFrom() 则返回已接收的字节数。
// 如果连接已关闭，则返回0。
// 否则，将返回值 SOCKET_ERROR，并通过调用 GetLastError() 来检索特定的错误代码。
case 0:
  // this->Close();
  break;
case SOCKET_ERROR:
  AfxMessageBox(_T("Error occurred"));
  Close();
default:
  // terminate the string
  m_szBuffer[nRead] = _T('\0');
  CString strTextOut = (CString)m_szBuffer;
  strp.Format(_T("%d"), port);

  // display the datetime
  // 获取当前时间日期 （用于服务器日志的显示）
  CTime m_time = CTime::GetCurrentTime();

  CReReServerDlg* pdlg = (CReReServerDlg*)AfxGetMainWnd();

  // 显示服务器日志
  pdlg->displayString(m_time.Format("%x %X") + _T("\r\n"));
  pdlg->displayString(_T("Receive message[") + strTextOut + _T("] from ") + host + _T(" in port ") + strp + _T(".\r\n"));
  pdlg->displayString(_T("Send message:\r\n"));
  CString response = _T("");

  // 调用 reply 函数
  if (!reply(strTextOut, response))
  {
    // 请求不支持，回复错误请求
    response = _T("Wrong cmd.");
  }
  pdlg->displayString(response + _T("\r\n"));
  char retchar[4096];

```

```
    // WideChar 占用空间太大，将其转换为占用较小空间的 Multibyte
    int len = WideCharToMultiByte(CP_ACP, 0, response, -1, NULL, 0, NULL,
     NULL);
    WideCharToMultiByte(CP_ACP, 0, response, -1, retchar, len, NULL, NULL);

    // 发送请求回复
    SendTo(retchar, strlen(retchar), port, host, 0);
  }

  CAsyncSocket::OnReceive(nErrorCode);
}
```

Listing 3: 服务器收到请求时如何处理::OnReceive

服务器收到请求时通过调用 ServerUDP 类中的 OnReceive 函数实现。

3.2 客户端的实现

```
void CReReClientDlg::OnBnClickedSend()
{
  // TODO: 在此添加控件通知处理程序代码

  CString cmd, host, strport;
  // 获取输入的请求
  cmdText.GetWindowTextW(cmd);
  // 获取输入的 IP
  addressText.GetWindowTextW(host);
  // 获取输入的端口号
  portText.GetWindowTextW(strport);
  int port = _wtoi(strport);

  char cmdchar[4096];

  // 生成 UDP Socket
  BOOL createFlag = usock.Create(0, SOCK_DGRAM, FD_READ);
  if (createFlag == 0)
  {
    MessageBox(_T("Socket create failed."));
    usock.Close();
    return;
  }

  // WideChar 占用空间太大，将其转换为占用较小空间的 Multibyte
  int len = WideCharToMultiByte(CP_ACP, 0, cmd, -1, NULL, 0, NULL, NULL);
  WideCharToMultiByte(CP_ACP, 0, cmd, -1, cmdchar, len, NULL, NULL);
```

```
28
29    // 发送请求
30    BOOL sendToFlag = usock.SendTo(cmdchar, strlen(cmdchar), port, host, 0);
31    if (sendToFlag == 0)
32    {
33      MessageBox(_T("Message send failed."));
34      usock.Close();
35      return;
36    }
37  }
```

Listing 4: 发送的实现

Start 按钮用于发送请求，生成新的 UDP Socket 实例，向服务器发送请求。

```
1   void ClientUDP::OnReceive(int nErrorCode)
2   {
3     // TODO: 在此添加专用代码和/或调用基类
4
5     // A buffer for the incoming data.
6     char m_szBuffer[4096];
7     // What ReceiveFrom Returns.
8     int nRead;
9     // To save ReceiveFrom IP
10    CString host;
11    // To save ReceiveFrom Port
12    UINT port;
13    // 如果错误，需要显示错误信息
14    int error;
15    CString tempStr;
16    CString errStr=_T("Error occurred");
17    // 接收服务器的回复
18    nRead = ReceiveFrom(m_szBuffer, sizeof(m_szBuffer), host, port,0);
19
20    switch (nRead)
21    {
22    // 如果未发生错误， ReceiveFrom() 则返回已接收的字节数。
23    // 如果连接已关闭，则返回0。
24    // 否则，将返回值 SOCKET_ERROR，并通过调用 GetLastError() 来检索特定的错误
        代码。
25    case 0:
26      // this->Close();
27      break;
28    case SOCKET_ERROR:
29      error = GetLastError();
```

```
    tempStr.Format(_T("%d"), error);
    AfxMessageBox(_T("Error occurred: ") + tempStr);
    Close();
  default:
    // terminate the string
    m_szBuffer[nRead] = _T('\0');
    CString strTextOut(m_szBuffer);

    // 显示收到的信息
    CReReClientDlg* pdlg = (CReReClientDlg*)AfxGetMainWnd();
    pdlg->displayString(strTextOut);
    this->Close();
  }

  CAsyncSocket::OnReceive(nErrorCode);
}
```

Listing 5: 收到服务器的回复时的处理

客户端收到回复时通过调用 ClientUDP 类中的 OnReceive 函数实现。

4 实验现象和结论的分析

可以使用 Wireshark 来捕捉和分析客户端与服务器之间的 UDP 流。我们测试了所有支持的请求（共四种），客户端请求发往服务器之后，服务器都有相应的回复。

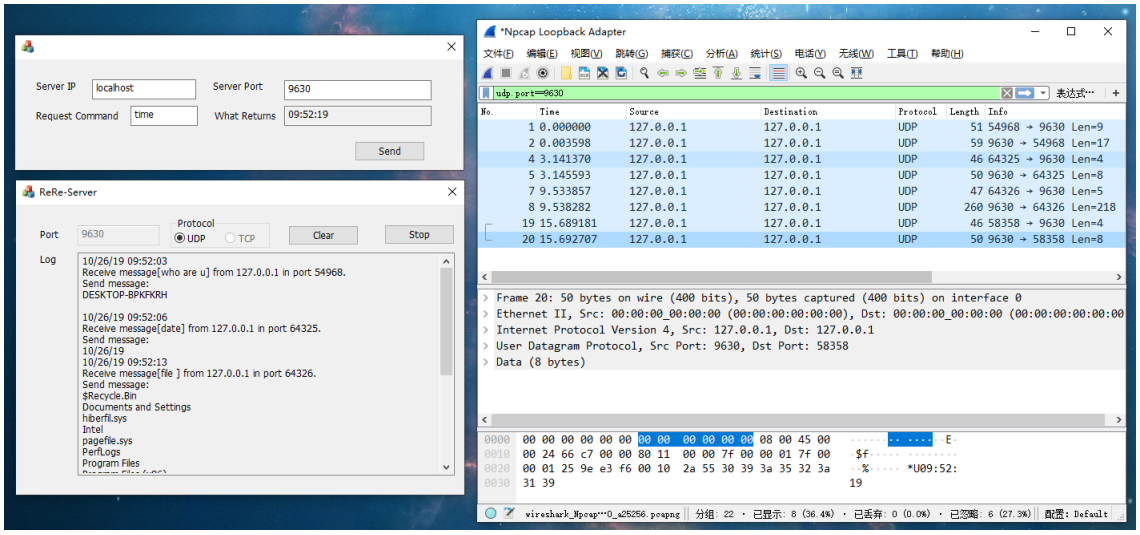

图 1: 客户端与服务器之间的 UDP 流

也可以从 Wireshark 中看到 UDP 报头的具体值。

UDP报头

偏移	字节	0								1								2								3							
字节	位	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	来源连接端口																目的连接端口														
4	32	报文长度																校验和														

图 2: UDP 报头结构

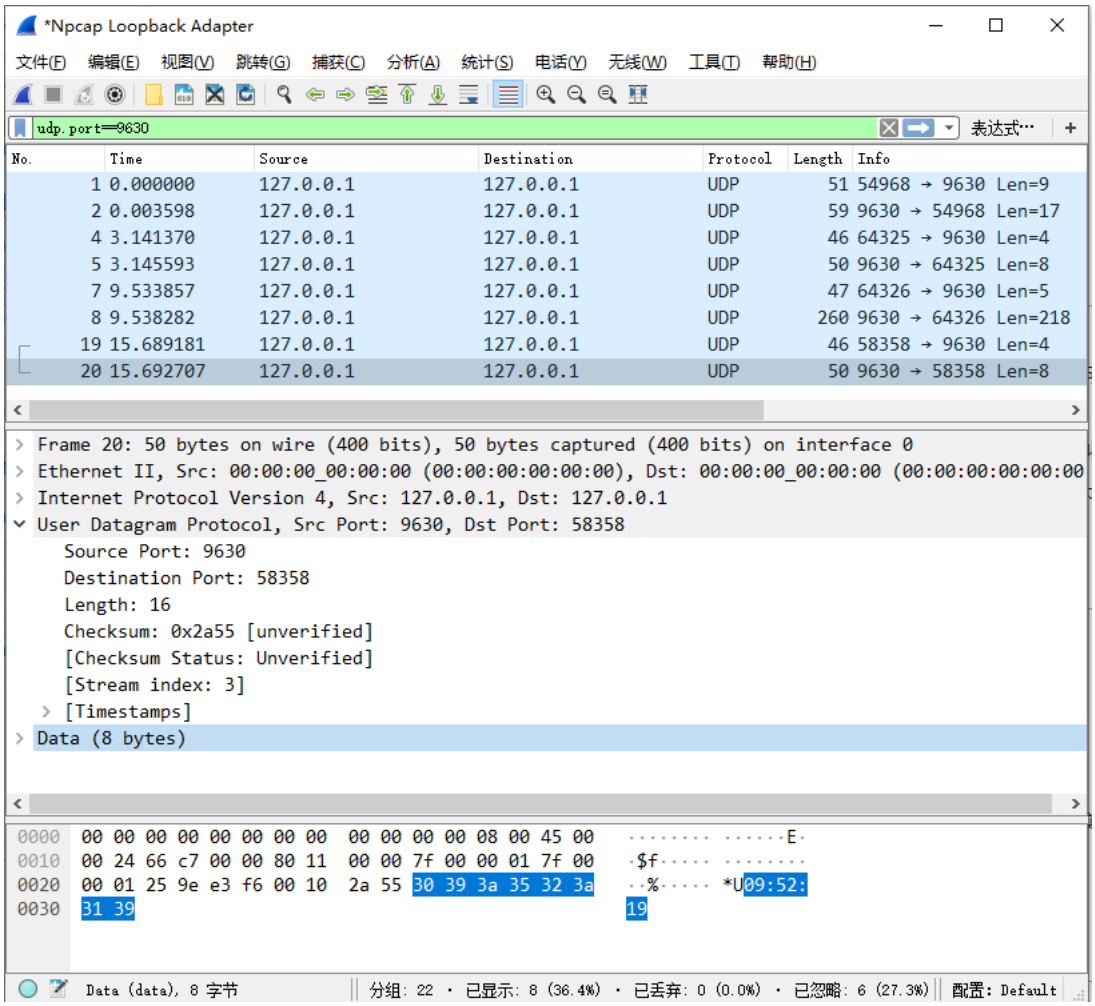

图 3: 该数据包的 UDP 报头

5 用户手册

我的客户端的名字叫 **ReReClient**，服务器的名字叫 **ReRe-Server**。下面先介绍它们的使用方法。

5.1 注意事项

1. 应该先打开服务器，开启服务器的服务后，再使用客户端与服务器进行交互。
2. 服务器的 TCP 选项不可选。（以 TCP 为主要传输协议的服务尚未实现）
3. 服务器和客户端是**支持多开**的。

5.2 客户端 —— ReReClient

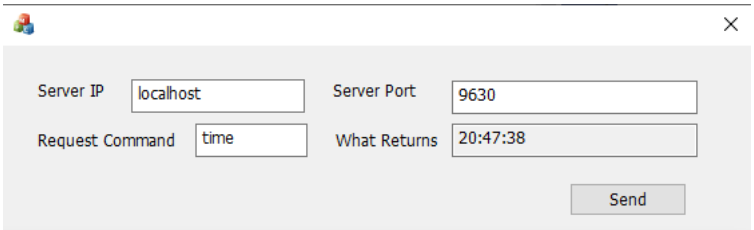

图 4: ReReClient

5.2.1 客户端界面详解：

- Server IP: 可以填写服务器端任意网卡的 IPv4 地址。
- Server Port：填写服务器端口号。
- Request Command：填写请求命令，支持的请求列表在 5.4。
- What Returns：显示服务器回复的信息。
- Send：发送请求。

5.2.2 客户端注意事项：

1. 如果输入的请求不在请求列表 5.4的范围内，在 What Returns 中会显示 “Wrong cmd.”。
2. 服务器的 IP 和端口号如果连续输错两次，客户端会退出。
3. 默认的 Server IP 为 localhost，Server Port 为 9630。

5.3 服务器 —— ReRe-Server

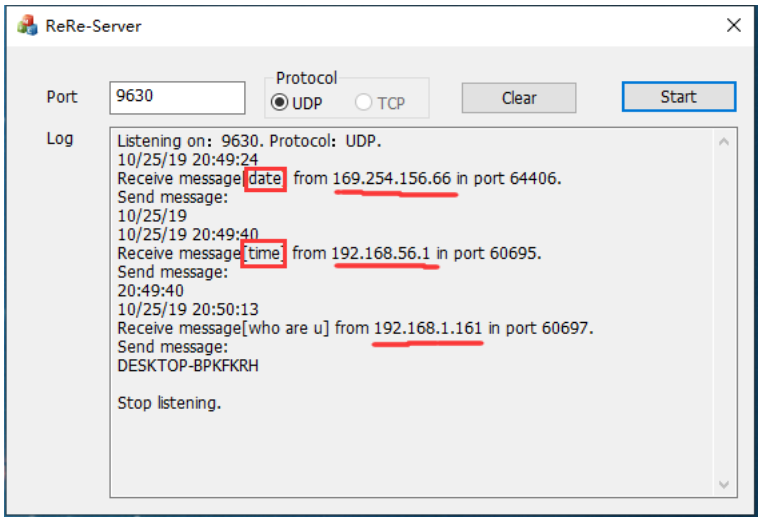

图 5: ReRe-Server

图片中的 IP 均为测试机器的网卡的 IPv4 地址，可参考附录 A。

5.3.1 服务器界面详解：

- Port：填写服务器端口号。
- Protocol: 选择采用 UDP 协议或 TCP 协议（以 TCP 为主要传输协议的服务尚未实现）
- Clear：清空日志。
- Start / Stop：开启 / 关闭服务。
- Log：服务器日志。

5.3.2 服务器注意事项：

1. 服务器采用的端口号不能和服务器所在机器的进程占用的端口号一样。
2. 默认端口号为 9630。

5.4 支持的请求

注意：所有请求均不区分大小写。

1. Time：获取服务器的系统时间。（时、分、秒）
2. Date：获取服务器的系统时间。（日期）
3. File [路径]：请求服务器指定路径下的文件和文件夹列表，但就算路径为空在 “File” 后也必须有一个空格。（在 Windows10 系统下默认路径为系统盘。）
4. Who are u：获取服务器的名字。

6 测试环境

本程序在测试坏境下编译及运行成功，不保证在任何机器上都能成功运行。

测试环境如下：

- 操作系统：Microsoft Windows 10 专业版 (x64) (version: 17763)
- IDE：Visual Studio 2019 (version: 16.3.5)
- Windows SDK version: 10.0
- 平台工具集：Visual Studio 2019 (v142)
- C++ 语言标准：std:C++17

A 测试机器的网卡

```
C:\Users\DELL
λ ipconfig.exe

Windows IP 配置


以太网适配器 以太网:

   媒体状态  . . . . . . . . . . . . : 媒体已断开连接
   连接特定的 DNS 后缀 . . . . . . . :

以太网适配器 Npcap Loopback Adapter:

   连接特定的 DNS 后缀 . . . . . . . :
   本地链接 IPv6 地址. . . . . . . . : fe80::882c:8d59:7965:9c42%15
   自动配置 IPv4 地址  . . . . . . . : 169.254.156.66
   子网掩码  . . . . . . . . . . . . : 255.255.0.0
   默认网关. . . . . . . . . . . . . :

以太网适配器 VirtualBox Host-Only Network:

   连接特定的 DNS 后缀 . . . . . . . :
   本地链接 IPv6 地址. . . . . . . . : fe80::e99f:46f7:ad59:7f2%20
   IPv4 地址 . . . . . . . . . . . . : 192.168.56.1
   子网掩码  . . . . . . . . . . . . : 255.255.255.0
   默认网关. . . . . . . . . . . . . :

无线局域网适配器 本地连接* 9:

   媒体状态  . . . . . . . . . . . . : 媒体已断开连接
   连接特定的 DNS 后缀 . . . . . . . :

无线局域网适配器 本地连接* 10:

   媒体状态  . . . . . . . . . . . . : 媒体已断开连接
   连接特定的 DNS 后缀 . . . . . . . :

无线局域网适配器 WLAN:

   连接特定的 DNS 后缀 . . . . . . . :
   本地链接 IPv6 地址. . . . . . . . : fe80::819e:16e0:69a4:5fda%17
   IPv4 地址 . . . . . . . . . . . . : 192.168.1.161
   子网掩码  . . . . . . . . . . . . : 255.255.255.0
   默认网关. . . . . . . . . . . . . : 192.168.1.1
```

图 6: 本机网卡 IPv4

References

[1] 张建忠、徐敬东. 计算机网络技术与应用. 北京清华大学学研大厦 A 座: 清华大学出版社, 2019.