Chapter 20 – Threading and Networking - Part II

# Objectives:

* Understanding Threads
* Handling Threads in VB
* Thread Odds and Ends
* Networking and Threads
* A Simple Chat Program

This chapter begins with an introduction to the concept of threads. Threading is the modern way to split the computing time your application gets into independently executing pieces. Even if you haven't worked with threads before, you shouldn’t find them too difficult.

You may be wondering what the reason is that we are bothering with discussing threads at all. Understanding and using threads are critical in preparing for the second major purpose of this chapter: building network applications using the sockets classes. Simply put, in order to write network applications using sockets, you got to use threads.

# Understanding Threads

Talking about threads is pretty easy. You almost intuitively know what they are and why they are needed. A microprocessor can only essentially execute one instruction at a time, right? I know you could have multiple cores, but each core can still only execute a single instruction on each core… For the longest time that's how we wrote programs too – they were big, monolithic monsters. Our operating systems even worked the same way. Good old Microsoft DOS – it could basically just run one program at a time. If you wanted to try to run a second program, you either had to quit the first one or at least put it on hold while the second one ran.

Today you would laugh if anybody told you to use an O/S that could only do one thing at a time. You want to be able to listen to MP3s, watch your instant messenger application, surf the web, work on a report in Word and write VB code all at the same time. Well, at least that's what I'm doing right now. This is called multitasking. There are many processes on the computer that are competing for its resources and the CPU switches back and forth between them rapidly, giving each a slice of time to get some of that particular process' work done. If you have enough memory and your CPU is fast enough, this activity is seamless. Your CPU, nevertheless, is still only running one thing at a time. Load enough programs up and you’ll start seeing the cracks in the multitasking as the necessary resources become scarce.

Now think about how you write programs. If you've ever written in a traditional 3rd generation language like Pascal, C, or even C++ for that matter, most of your programs have revolved around a main function that called everything else. You had no parallel work being carried out – in fact, you could even predict when a specific subprogram would be run. Welcome to the idea of adding threads. Threads are essentially pieces of your program that you break out and run independent of each other in parallel. Okay, now that sounds sort of threatening.

If I asked you to write a video game like any of the popular first-person shooters, how would you go about it? Oh, I'm sure you could come up with math routines to handle geometry and graphical texture shading, and that you could write keyboard and mouse routines to handle input. My question really ought to be: how well would this game run? Think about a giant Main loop that looks like this:

Module Module1

Sub Main()

While Not (GameOver)

GetKeyboardInput()

MovePlayer()

RandomlyMoveEnemy()

UpdateScreen()

End While

End Sub

End Module

Think about how this program would execute – the player enters some input and then the program must calculate the player’s move, then the program has to calculate the enemies' move, next the screen is updated and finally the player gets control back to make another move. My guess is that this is going to play choppy at best. I need to have continual input control for gaming realism (if you don't think it's important, try playing a modern first-person shooter on an older computer). The processor is just too slow and busy to update the graphics and your input is almost always "late."

What I really should do is take each important part of the game and run it independently in its own thread. I could have a GetKeyboardInput thread that constantly looks for keyboard input, grabs the input and delivers it to the MoveMyCharacter thread. The MoveMyCharacter thread is responsible for calculating and moving my character. Likewise, I could construct a MoveEnemyThread for handling the enemies. I could have the environment rendering taking place in yet another thread.

The beauty of all of this is that the threads are executing independently of each other in my program – to a degree this is somewhat analogous to the Windows operating system running multiple programs at the same time. Okay, so now how bad was that to understand?

# Handling Threads in VB

One of the big features that .NET delivered to VB programmers was threads. Let's jump right in and look at how to do threading with some simple code examples.

## Creating and Running Threads

To create a thread, we simply create new Thread objects. We then need to assign the address of the routine that we want to be run in that thread. Finally, we start the thread up with the Start method. This is so simple that the code example is pretty much self-explanatory:

'Chapter 20 - Program 1

Module Module1

'This subroutine will be assigned to its own thread

Sub Thread1sWork()

Dim intLoop As Integer

'Thread 1 is supposed to print out 300 "1"s

For intLoop = 1 To 300

'Console.Write("1")

Next

End Sub

'This subroutine will also be assigned to its own thread

Sub Thread2sWork()

Dim intLoop As Integer

For intLoop = 1 To 300

Console.Write("2")

Next

End Sub

'This subroutine gets assigned to its own thread

Sub Thread3sWork()

Dim intLoop As Integer

For intLoop = 1 To 300

Console.Write("3")

Next

End Sub

'Here's where the program will actually begin executing

Sub Main()

'Create three unique thread objects...

Dim Thread1 As System.Threading.Thread

Dim Thread2 As System.Threading.Thread

Dim Thread3 As System.Threading.Thread

'Take our three threads and assign each one the

'address of a routine that we want that thread to run.

'I want Thread1 to do Thread1sWork, Thread2 to do

'Thread2sWork and Thread3 to do Thread3sWork.

'In order to do this, we need to give the address of each

'routine to the appropriate thread. The AddressOf operator can

'do this for us.

Thread1 = New Threading.Thread(AddressOf Thread1sWork)

Thread2 = New Threading.Thread(AddressOf Thread2sWork)

Thread3 = New Threading.Thread(AddressOf Thread3sWork)

'Now we need to start each thread by invoking that thread's

'Start method

Thread1.Start()

Thread2.Start()

Thread3.Start()

'Pause so we can see the results. The Console.ReadLine method

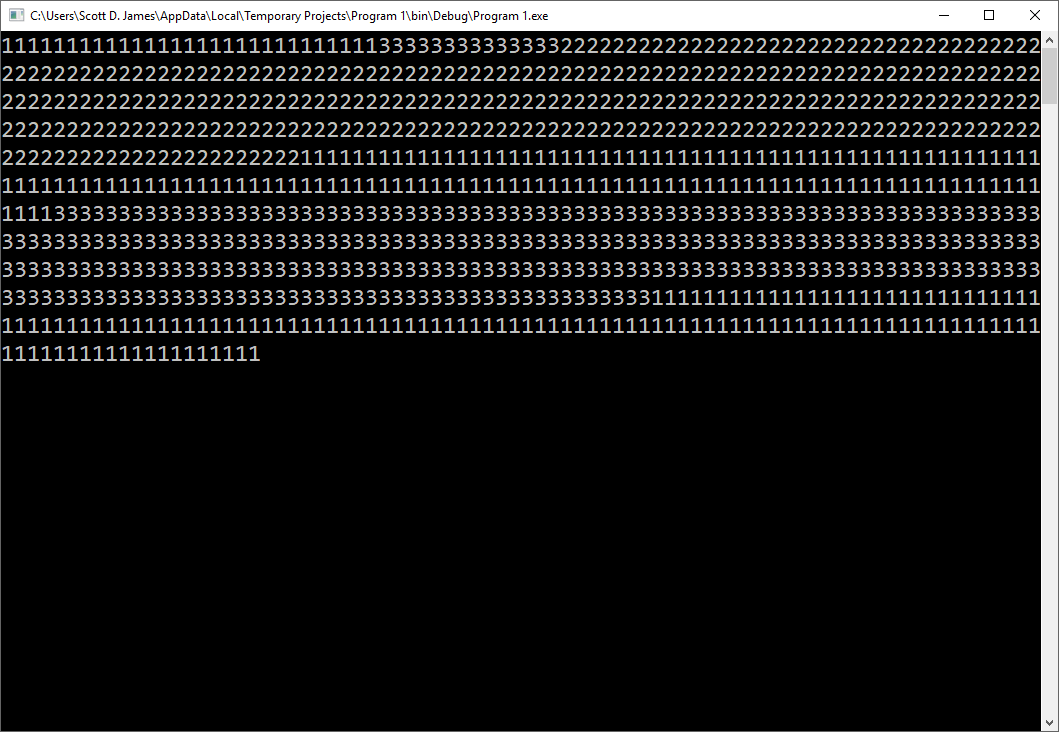
'waits until the enter key is pressed before continuing on

Console.ReadLine()

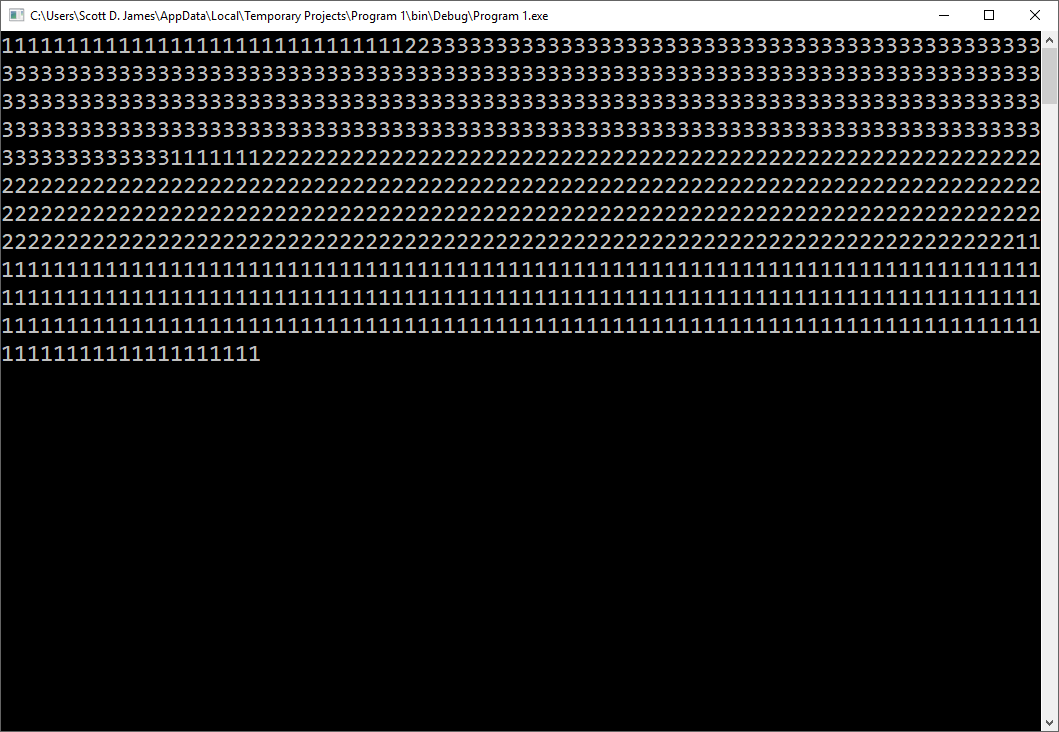
End Sub

End Module

Do you want to make any bets on the output? In a traditional language without threads, we would see 300 “1”s, 300 “2”s and then 300 “3”s. I hope that you realize that this is not the output we will get here. In fact, there's absolutely no guarantee exactly what will get generated as evidenced by the next screenshot. As you can see, all the work does gets done, but it's scheduled by the operating system and CPU. It's quite out of our hands as to how and when each piece will be executed:



Now let's run the program again. Notice that this output is significantly different from the previous one. This shows the unpredictability of thread execution. The threads are being executed by the operating system and that's about all that one can say.



## Thread Methods and Properties

Now that we've seen the world’s simplest thread program, we should probably pause to take a look at what we can do with the Thread class. Here’s a list of the Thread class’ methods, many of which we will not use, but it gives you a sense of the complexity that you design around a threaded program architecture:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| Abort | Begins the process of attempting to terminate a thread (the thread may not terminate) |
| AllocateDataSlot | Allocates an unnamed data slot within which the thread can store thread-specific data for all threads to access – we need these to facilitate interthread communication |
| AllocateNamedDataSlot | Allocates a named data slot for all threads to access |
| BeginCriticalRegion | Informs host that a region of code is to be executed in which the thread aborting or an unhandled exception could cause problems |
| BeginThreadAffinity | Notifies that instructions are about to execute that depend on the identity of the current physical operating system thread |
| EndCriticalRegion | Thread effects are limited now to the current task |
| EndThreadAffinity | Notifies that affinity code has completed executing |
| FreeNamedDataSlot | Frees an allocated named data slot |
| GetApartmentState | Returns the ApartmentState value |
| GetData | Retrieves the value from the current thread's specified named data slot |
| GetDomain | Returns the current thread's domain (the environment in which an application executes) |
| GetDomainID | Returns the application's unique domain identifier |
| GetNamedDataSlot | Returns the named data slot or a new data slot if the named data slot doesn't exist |
| Interrupt | Interrupts a thread that is blocked by a Wait, Sleep or Join operation |
| Join | Blocks the calling thread until the specified thread terminates |
| MemoryBarrier | The processor executing the current thread cannot reorder instructions in such a way that memory access prior to the MemoryBarrier call execute after the call |
| ResetAbort | Cancels an Abort operation on the current thread |
| SetApartmentState | Sets the apartment state of a thread before it is started |
| SetData | Sets data in a current thread data slot |
| SetProcessorAffinity | Sets the processor affinity for a managed thread, determining which processor will run the thread |
| Sleep | Blocks the current thread for the specified number of milliseconds |
| SpinWait | Causes a thread to wait the number of iterations specified |
| Start | Begins a thread's execution by directing the O/S to change the thread's state to ThreadStateRunning |
| TrySetApartmentState | Sets the apartment state of a thread before it is started |
| VolatileRead | Reads the value of a field, being the latest value written by any processor in a computer |
| VolatileWrite | Writes a value to a field immediately, do that the value is visible to all processors |
| Yield | Causes the calling thread to yield execution to another thread that is ready to run on the current processor |

Here is a list of the various Thread class properties:

|  |  |
| --- | --- |
| *Property* | *Purpose* |
| ApartmentState | Gets or sets the apartment state of this thread. The apartment state defines if the thread is running in a single-thread or multithreaded apartment. By assigning each thread its own storage location (an apartment), the apartment model lets threads protect their resources from one another |
| CurrentContext | Gets the thread's current context (properties that define the execution within which the object resides) |
| CurrentCulture | Gets the thread's current culture (properties that control date, time, currency formats, sorting order, et cetera) |
| CurrentPrincipal | Gets or sets the thread's current principal that specifies the thread's security context |
| CurrentThread | Gets the thread that is currently running |
| CurrentUICulture | Gets or sets the current culture used by the resource manager to look up culture-specific resources at runtime |
| ExecutionContext | Gets an ExecutionContext object that contains information about the various contexts of the current thread |
| IsAlive | Returns true if the thread's state is not started, stopped or aborted |
| IsBackground | Returns true if the thread is a background thread |
| IsThreadPoolThread | Returns true if the thread was allocated from the system thread pool |
| ManagedThreadId | Gets a unique identified for the current managed thread |
| Name | Gets or sets the thread's name |
| Priority | Gets or sets the thread's scheduling priority |
| ThreadState | Gets a value that specifies the thread's current state |

## Putting a Thread to Sleep

We can affect the execution of threads by putting them to sleep for periods of time. Let's revisit the last example and modify the program so that Thread3sWork gets done first, then Thread2’s work and finally Thread1's. This is exactly the opposite order of what we'd expect since Thread 1 starts executing first. What will do is put Thread 1 to sleep for 2 seconds, Thread 2 to sleep for 1 second and Thread 3 won't sleep at all; consequently 3 will get all of its work done first since the other two threads are sleeping.

Here’s the source code for this application:

'Chapter 20 - Program 2

Module Module1

'Create three thread objects here so that the whole module can

'see them. We need to do this so that we can access the threads

'to put them to sleep...

Dim Thread1 As System.Threading.Thread

Dim Thread2 As System.Threading.Thread

Dim Thread3 As System.Threading.Thread

Sub Thread1sWork()

Dim intLoop As Integer

'As soon as 1 starts up, make it sleep for 2000 ms.

Threading.Thread.Sleep(2000)

For intLoop = 1 To 300

Console.Write("1")

Next

End Sub

Sub Thread2sWork()

Dim intLoop As Integer

'As soon as 2 starts up, make it sleep for 1000 ms.

Threading.Thread.Sleep(1000)

For intLoop = 1 To 300

Console.Write("2")

Next

End Sub

Sub Thread3sWork()

Dim intLoop As Integer

For intLoop = 1 To 300

Console.Write("3")

Next

End Sub

'Here's the Main subroutine

Sub Main()

'Take our three threads and assign each to the

'routine we want it to run.

Thread1 = New Threading.Thread(AddressOf Thread1sWork)

Thread2 = New Threading.Thread(AddressOf Thread2sWork)

Thread3 = New Threading.Thread(AddressOf Thread3sWork)

'Now we need to start each thread

Thread1.Start()

Thread2.Start()

Thread3.Start()

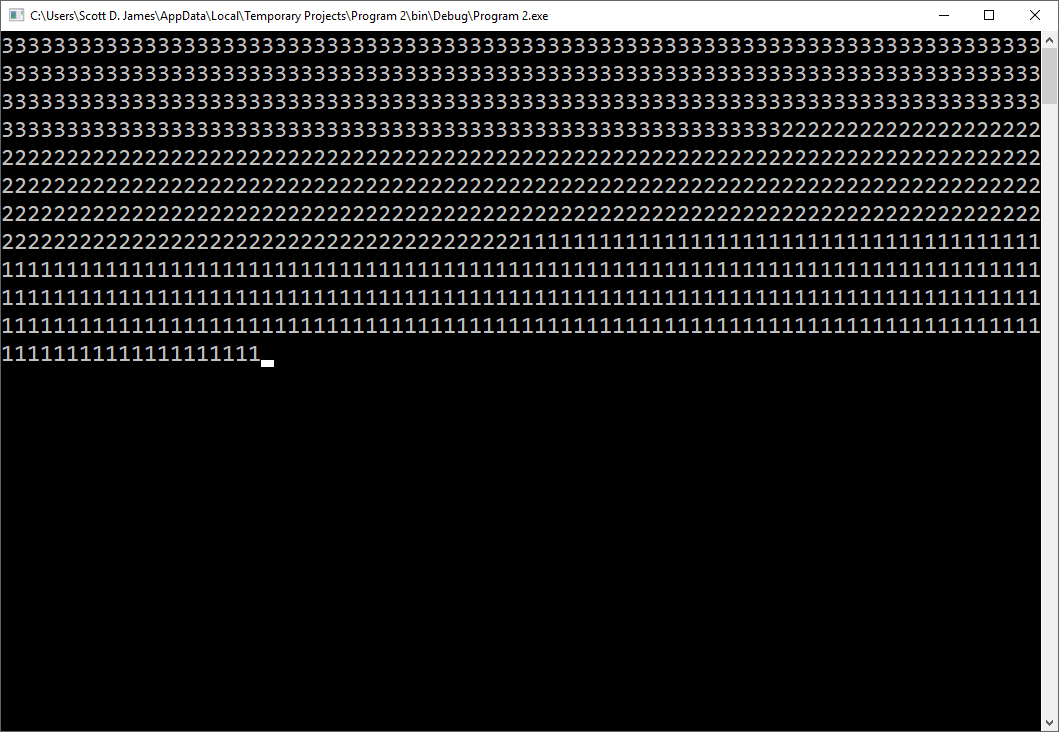
'Pause so we can see the results

Console.ReadLine()

End Sub

End Module

Here's the output from this application. You will notice that all of Thread3's work was completed, then all of Thread2's and finally all of Thread1's. This was strictly a result of putting the threads to sleep:



## Suspending, Resuming and Aborting a Thread

Now that we understand how to start threads and how to put them to sleep, we need to examine how to suspend, resume and abort threads. The next code ought to be straightforward to you. Here's a description of its purpose. When Thread3 starts up, it will get suspended. When Thread2 starts up, it will try to abort Thread1 and then check to see if Thread3 is still suspended. If it is, Thread2 will get Thread3 to resume its execution.

Source code-wise this is a pretty simple implementation:

'Chapter 20 - Program 3

'There are two obsolete methods used in this program:

'Suspend and Resume. These are not safe ways to handle

'dealing with threads because you may suspend a thread

'waiting on a resource, causing deadlock in the O/S.

'There are better, more modern ways to handle these

'issues but it is beyong what I need to you know about

'in this class. If you are interested to move along

'further, look into Workers Threads. The code here

'gets us to where we want without complexity and it

'is for illustrative purposes only!

Imports System.Threading

Module Module1

'Here are our three threads

Dim Thread1 As System.Threading.Thread

Dim Thread2 As System.Threading.Thread

Dim Thread3 As System.Threading.Thread

Sub Thread1sWork()

Dim intLoop As Integer

'1 is similar to the previous examples, except that we have

'to watch for ThreadAbortExceptions since we are going to try

'to stop this thread from Thread2.

Try

For intLoop = 1 To 300

Console.Write("1")

Next

Catch ex As ThreadAbortException

Console.Write(" \*\* Thread1 has been aborted \*\* ")

End Try

End Sub

Sub Thread2sWork()

Dim intLoop As Integer

'If Thread1 is still executing, try to stop it. It may

'finish up its work before it can be stopped though.

If Thread1.IsAlive() Then

Thread1.Abort()

End If

For intLoop = 1 To 300

'On each iteration of our work, see if Thread3 is still

'in suspension, if it is try to get it to resume doing

'its work.

If (Thread3.ThreadState = ThreadState.Suspended) Then

'This is an obsolete call that still works, leaving

'it here just to show concept...resume is a bad

'programming practice because you could mess up

'the state of the threads

Thread3.Resume()

End If

Console.Write("2")

Next

End Sub

Sub Thread3sWork()

Dim intLoop As Integer

'As soon as 3 starts up, try to suspend it. If it comes

'out of suspension, it will start performing its work.

'This is an obsolete call that still works, leaving

'it here just to show concept...suspend is a bad

'programming practice because you could mess up

'the state of the threads

Thread3.Suspend()

For intLoop = 1 To 300

Console.Write("3")

Next

End Sub

Sub Main()

Thread1 = New Threading.Thread(AddressOf Thread1sWork)

Thread2 = New Threading.Thread(AddressOf Thread2sWork)

Thread3 = New Threading.Thread(AddressOf Thread3sWork)

Thread1.Start()

Thread2.Start()

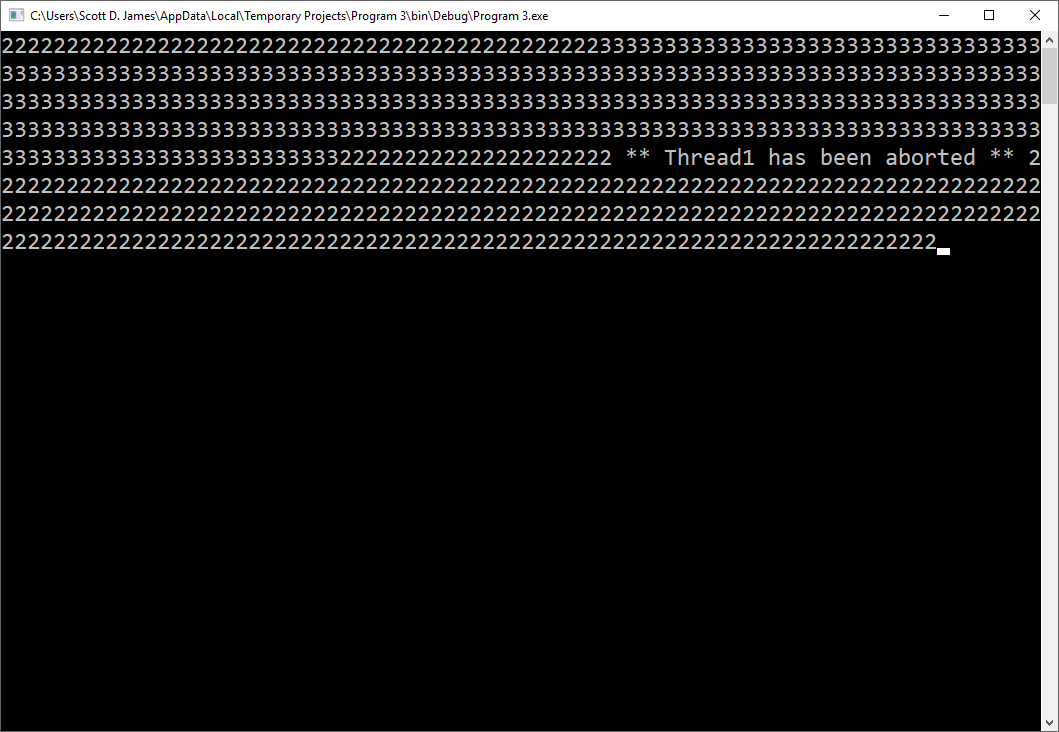
Thread3.Start()

Console.ReadLine()

End Sub

End Module

Here's the output from our application:



Notice in this instance that Thread1 was aborted. You may run the application and sometimes Thread1 will get its work done and terminate before it can be aborted. There’s no guarantee to how things will get processed.

## Understanding Race Conditions

As you've seen with the examples that we have looked at, threads aren't predictable in their execution. Think for a minute about the following problem. Let's assume that one thread is supposed to produce an output which becomes input for a second thread. Can we guarantee that this will always happen correctly? By just using the standard threading keywords that we've looked at so far, the answer is no. This problem is called a race condition in which the threads get out of synchronization.

Let's take a close look at the following example. We can build an employee class instance that is supposed to be shared between two executing threads. Thread one is supposed to set the name in the class to Bill and Bill’s wage in the class to 30000. Thread two is supposed to set the name to Sue and Sue’s wage to 45000.

An ideal output would look like this:

Name: Bill

Wage: 30000

Name: Sue

Wage: 45000

Name: Bill

Wage: 30000

and so on. Let's write the code for an application to do this:

'Chapter 20 - Program 4

Imports System.Threading

'Here is the Employee class

Class Employee

Public name As String

Public wage As Single

End Class

Module Module1

'Create a single instance of class Employee that both threads

'will share

Dim myEmp As New Employee()

Sub DoWork()

'Create an infinite loop of updating employee information in

'the Employee class.

While (True)

If Thread.CurrentThread.Name() = "Thread1" Then

'Since thread one is executing set the employee's name

'to Bill

myEmp.name = "Bill"

'Now give other threads a chance to do some of their

'work. This really is to simulate a busy system. If

'your system is busy enough, we wouldn't need this

'code in here.

Thread.Sleep(CInt((100 \* Rnd()) + 1))

'Okay, our thread is running again at this point, so

'now set the employee's wage to 30000.

myEmp.wage = 30000.0

Else

'Thread 2 is running so set the employee's name to

'Sue and the wage to 45000.

myEmp.name = "Sue"

'again, give other threads a chance to do work

Thread.Sleep(CInt((100 \* Rnd()) + 1))

myEmp.wage = 45000.0

End If

'Now print the values that are in the Name and Wage

'members.

Console.WriteLine("Name: " & myEmp.name)

Console.WriteLine("Wage: " & myEmp.wage)

End While

End Sub

Sub Main()

Dim thread1 As Thread

Dim thread2 As Thread

thread1 = New Thread(AddressOf DoWork)

thread2 = New Thread(AddressOf DoWork)

'Give the threads names so that we can determine which

'thread is executing in our loop. We will then know which

'employee's information to place in the employee object.

thread1.Name = "Thread1"

thread2.Name = "Thread2"

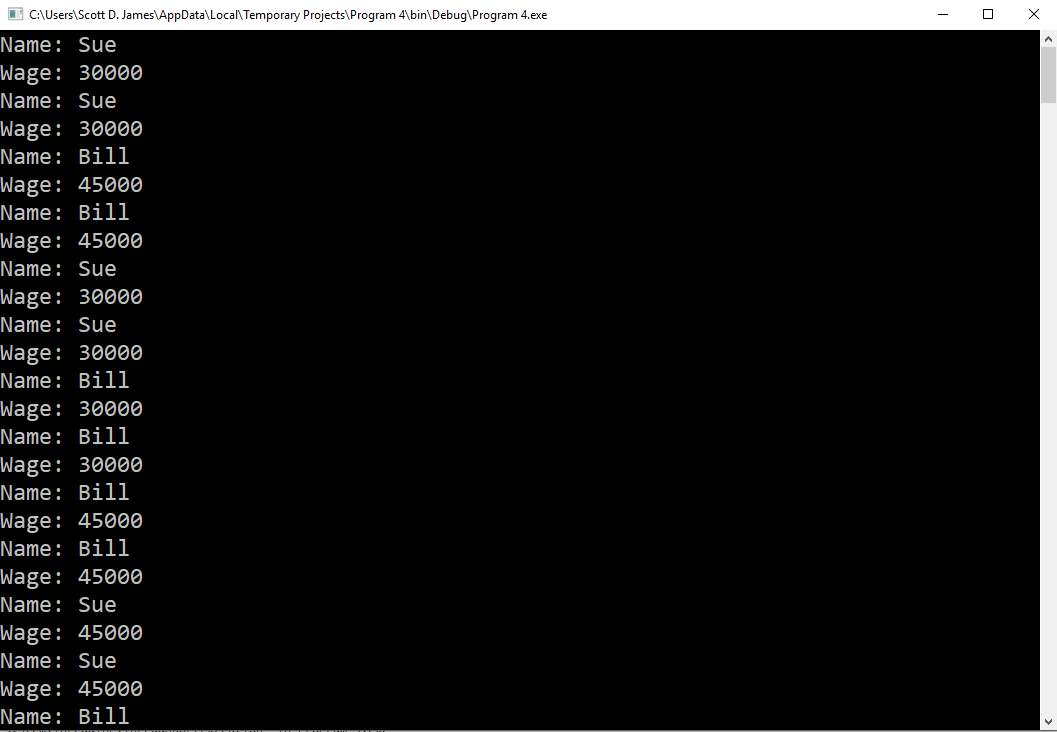
thread1.Start()

thread2.Start()

End Sub

End Module

The next screenshot shows the application in action. Notice that the output is far from ideal. We can see that both Bill and Sue end up making 30000 and then 45000 in this screen:



The important question is why did this happen? It's due to what’s known as a race condition. Both threads were busy doing their work and due to some anomaly, they got out of synchronization. This is a very common issue that occurs when you write threaded applications. Another issue that’s related is deadlock on a system resource – thread two needs something that thread one has in order for thread two to continue, but thread one is waiting on thread two to complete its processing before it will release the resource.

Fortunately for us, this is a reasonably simple problem to fix. What we need to do is use a SyncLock block. This basically places a lock on a piece of data that some thread is using. That piece of data remains locked until the thread that created the SyncLock ends the SyncLock block. If another thread tries to modify the data while it is SyncLock, that secondary thread just has to wait until the data becomes unlocked.

Let's rewrite our last application to take advantage of SyncLock to make our race condition problem go away:

'Chapter 20 - Program 5

Imports System.Threading

Class employee

Public name As String

Public wage As Single

End Class

Module Module1

Public myEmp As New employee()

Sub DoWork()

'Create an infinite loop of data updating

While (True)

'Lock the resource that is getting corrupted

'This says nobody else can touch this until

'the SyncLock block ends

SyncLock myEmp

If Thread.CurrentThread.Name() = "Thread1" Then

myEmp.name = "Bill"

'give other threads a chance to do work

Thread.Sleep(CInt((100 \* Rnd()) + 1))

myEmp.wage = 30000.0

Else

myEmp.name = "Sue"

'give other threads a chance to do work

Thread.Sleep(CInt((100 \* Rnd()) + 1))

myEmp.wage = 45000.0

End If

'Print out the employee information

Console.WriteLine("Name: " & myEmp.name)

Console.WriteLine("Wage: " & myEmp.wage)

'The data has been successfully read and written to/from

'the employee class so we can release the SyncLock

End SyncLock

End While

End Sub

Sub Main()

Dim thread1 As Thread

Dim thread2 As Thread

thread1 = New Thread(AddressOf DoWork)

thread2 = New Thread(AddressOf DoWork)

thread1.Name = "Thread1"

thread2.Name = "Thread2"

thread1.Start()

thread2.Start()

End Sub

End Module

Here's the program output from the SyncLock example. Notice that the employees correctly alternate between Bill and Sue and that all wage values are correct:



As you can see, there are a lot of issues that you must consider when you are working with threads. Threading isn't particularly difficult; rather, you just need to be defensive when you write your code. If you want more information on threading in general, I would suggest that you consult a book on operating systems. Also, before you plan to do anything overly complex with threading in VB, you need to read up on the nuances of working with threads and be prepared to do some experimenting before trying to write production code.

# Thread Odds and Ends

Now that we've introduced the idea of threads and worked through several examples of using them, we’ll turn to the Process class. Many of the concepts and items here are a mix between threads and commands like Shell.

## Another Way to Launch Applications

The Process.Start(*processname*) method will run a program similar to what the Shell command performs.

## Terminating a Process

You can use the Process.GetCurrentProcess.Kill() method to end the current executing process. In addition, if you want to stop a particular process whose process ID number is known, you can also use the Process.GetCurrentProcessById(*idnumber*).Kill() method to terminate the particular process number.

## Preventing Two Copies of the Same Program from Executing at the Same Time

Another common concern that we have with some applications is that we do not want more than one copy of that program to be run. Here is some sample code that shows you how to handle this issue using methods from the Process class:

'Chapter 20 - Program 6

Module Module1

Sub Main()

Dim MatchingNames As Process()

Dim TargetName As String

'Get this application's name

TargetName = Process.GetCurrentProcess.ProcessName

'Call GetProcessByName sending in the target name. This

'method will return an array of ProcessNames that match

'the target name

MatchingNames = Process.GetProcessesByName(TargetName)

If (MatchingNames.Length = 1) Then

'If length = 1, then only this copy of the application

'was found in the array, so go ahead and continue to let

'the application run.

Else

'There was more than 1 copy, so this one should end.

MsgBox("Error: Process is already running!")

End If

End Sub

'Note: If you want to see this program in action, get a version

'built into an .EXE. Set a breakpoint in the first few lines

'of this program and run it. Then hop over to the Debug folder

'and run the .EXE version. You'll see the MsgBox appear when

'the second copy runs...

End Module

Here’s the full list of Process class properties that we care about:

|  |  |
| --- | --- |
| *Property* | *Description* |
| [BasePriority](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.basepriority.aspx) | Gets the base priority of the associated process. |
| [EnableRaisingEvents](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.enableraisingevents.aspx) | Gets or sets whether the [Exited](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.exited.aspx) event should be raised when the process terminates. |
| [ExitCode](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.exitcode.aspx) | Gets the value that the associated process specified when it terminated. |
| [ExitTime](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.exittime.aspx) | Gets the time that the associated process exited. |
| [Handle](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.handle.aspx) | Gets the native handle of the associated process. |
| [HandleCount](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.handlecount.aspx) | Gets the number of handles opened by the process. |
| [HasExited](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.hasexited.aspx) | Gets a value indicating whether the associated process has been terminated. |
| [Id](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.id.aspx) | Gets the unique identifier for the associated process. |
| [MachineName](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.machinename.aspx) | Gets the name of the computer the associated process is running on. |
| [MainModule](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.mainmodule.aspx) | Gets the main module for the associated process. |
| [MainWindowHandle](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.mainwindowhandle.aspx) | Gets the window handle of the main window of the associated process. |
| [MainWindowTitle](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.mainwindowtitle.aspx) | Gets the caption of the main window of the process. |
| [MaxWorkingSet](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.maxworkingset.aspx) | Gets or sets the maximum allowable working set size for the associated process. |
| [MinWorkingSet](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.minworkingset.aspx) | Gets or sets the minimum allowable working set size for the associated process. |
| [Modules](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.modules.aspx) | Gets the modules that have been loaded by the associated process. |
| [NonpagedSystemMemorySize64](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.nonpagedsystemmemorysize64.aspx) | Gets the amount of nonpaged system memory allocated for the associated process. |
| [PagedMemorySize64](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.pagedmemorysize64.aspx) | Gets the amount of paged memory allocated for the associated process. |
| [PagedSystemMemorySize64](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.pagedsystemmemorysize64.aspx) | Gets the amount of pageable system memory allocated for the associated process. |
| [PeakPagedMemorySize64](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.peakpagedmemorysize64.aspx) | Gets the maximum amount of memory in the virtual memory paging file used by the associated process. |
| [PeakVirtualMemorySize64](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.peakvirtualmemorysize64.aspx) | Gets the maximum amount of virtual memory used by the associated process. |
| [PeakWorkingSet64](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.peakworkingset64.aspx) | Gets the maximum amount of physical memory used by the associated process. |
| [PriorityBoostEnabled](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.priorityboostenabled.aspx) | Gets or sets a value indicating whether the associated process priority should temporarily be boosted by the operating system when the main window has the focus. |
| [PriorityClass](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.priorityclass.aspx) | Gets or sets the overall priority category for the associated process. |
| [PrivateMemorySize64](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.privatememorysize64.aspx) | Gets the amount of private memory allocated for the associated process. |
| [PrivilegedProcessorTime](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.privilegedprocessortime.aspx) | Gets the privileged processor time for this process. |
| [ProcessName](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.processname.aspx) | Gets the name of the process. |
| [ProcessorAffinity](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.processoraffinity.aspx) | Gets or sets the processors on which the threads in this process can be scheduled to run. |
| [Responding](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.responding.aspx) | Gets a value indicating whether the user interface of the process is responding. |
| [SessionId](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.sessionid.aspx) | Gets the Terminal Services session identifier for the associated process. |
| [StandardError](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.standarderror.aspx) | Gets a stream used to read the error output of the application. |
| [StandardInput](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.standardinput.aspx) | Gets a stream used to write the input of the application. |
| [StandardOutput](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.standardoutput.aspx) | Gets a stream used to read the output of the application. |
| [StartInfo](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.startinfo.aspx) | Gets or sets the properties to pass to the [Start](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.start.aspx) method of the Process. |
| [StartTime](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.starttime.aspx) | Gets the time that the associated process was started. |
| [SynchronizingObject](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.synchronizingobject.aspx) | Gets or sets the object used to marshal the event handler calls that are issued as a result of a process exit event. |
| [Threads](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.threads.aspx) | Gets the set of threads that are running in the associated process. |
| [TotalProcessorTime](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.totalprocessortime.aspx) | Gets the total processor time for this process. |
| [UserProcessorTime](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.userprocessortime.aspx) | Gets the user processor time for this process. |
| [VirtualMemorySize64](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.virtualmemorysize64.aspx) | Gets the amount of virtual memory allocated for the associated process. |
| [WorkingSet64](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.workingset64.aspx) | Gets the amount of physical memory allocated for the associated process. |

Here are the Process class methods:

|  |  |
| --- | --- |
| *Method* | *Description* |
| [BeginErrorReadLine](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.beginerrorreadline.aspx) | Begins asynchronous read operations on the redirected [StandardError](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.standarderror.aspx) stream of the application. |
| [BeginOutputReadLine](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.beginoutputreadline.aspx) | Begins asynchronous read operations on the redirected [StandardOutput](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.standardoutput.aspx) stream of the application. |
| [CancelErrorRead](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.cancelerrorread.aspx) | Cancels the asynchronous read operation on the redirected [StandardError](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.standarderror.aspx) stream of an application. |
| [CancelOutputRead](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.canceloutputread.aspx) | Cancels the asynchronous read operation on the redirected [StandardOutput](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.standardoutput.aspx) stream of an application. |
| [Close](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.close.aspx) | Frees all the resources that are associated with this component. |
| [CloseMainWindow](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.closemainwindow.aspx) | Closes a process that has a user interface by sending a close message to its main window. |
| [EnterDebugMode](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.enterdebugmode.aspx) | Puts a Process component in a state to interact with operating system processes that run in a special mode by enabling the native property SetDebugPrivilege on the current thread. |
| [GetCurrentProcess](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.getcurrentprocess.aspx) | Gets a new Process component and associates it with the currently active process. |
| [GetProcessById(Int32)](http://msdn.microsoft.com/en-us/library/76fkb36k.aspx) | Returns a new Process component, given the identifier of a process on the local computer. |
| [GetProcessById(Int32, String)](http://msdn.microsoft.com/en-us/library/wxw29dec.aspx) | Returns a new Process component, given a process identifier and the name of a computer on the network. |
| [GetProcesses](http://msdn.microsoft.com/en-us/library/1f3ys1f9.aspx) | Creates a new Process component for each process resource on the local computer. |
| [GetProcesses(String)](http://msdn.microsoft.com/en-us/library/x8b2hzk8.aspx) | Creates a new Process component for each process resource on the specified computer. |
| [GetProcessesByName(String)](http://msdn.microsoft.com/en-us/library/z3w4xdc9.aspx) | Creates an array of new Process components and associates them with all the process resources on the local computer that share the specified process name. |
| [GetProcessesByName(String, String)](http://msdn.microsoft.com/en-us/library/725c3z81.aspx) | Creates an array of new Process components and associates them with all the process resources on a remote computer that share the specified process name. |
| [Kill](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.kill.aspx) | Immediately stops the associated process. |
| [LeaveDebugMode](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.leavedebugmode.aspx) | Takes a Process component out of the state that lets it interact with operating system processes that run in a special mode. |
| [OnExited](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.onexited.aspx) | Raises the [Exited](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.exited.aspx) event. |
| [Refresh](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.refresh.aspx) | Discards any information about the associated process that has been cached inside the process component. |
| [Start](http://msdn.microsoft.com/en-us/library/e8zac0ca.aspx) | Starts (or reuses) the process resource that is specified by the [StartInfo](http://msdn.microsoft.com/en-us/library/system.diagnostics.process.startinfo.aspx) property of this Process component and associates it with the component. |
| [WaitForExit](http://msdn.microsoft.com/en-us/library/fb4aw7b8.aspx) | Instructs the Process component to wait indefinitely for the associated process to exit. |
| [WaitForExit(Int32)](http://msdn.microsoft.com/en-us/library/ty0d8k56.aspx) | Instructs the Process component to wait the specified number of milliseconds for the associated process to exit. |
| [WaitForInputIdle](http://msdn.microsoft.com/en-us/library/8d7363e2.aspx) | Causes the Process component to wait indefinitely for the associated process to enter an idle state. This overload applies only to processes with a user interface and, therefore, a message loop. |
| [WaitForInputIdle(Int32)](http://msdn.microsoft.com/en-us/library/kcdbkyt4.aspx) | Causes the Process component to wait the specified number of milliseconds for the associated process to enter an idle state. This overload applies only to processes with a user interface and, therefore, a message loop. |

# Networking and Threads

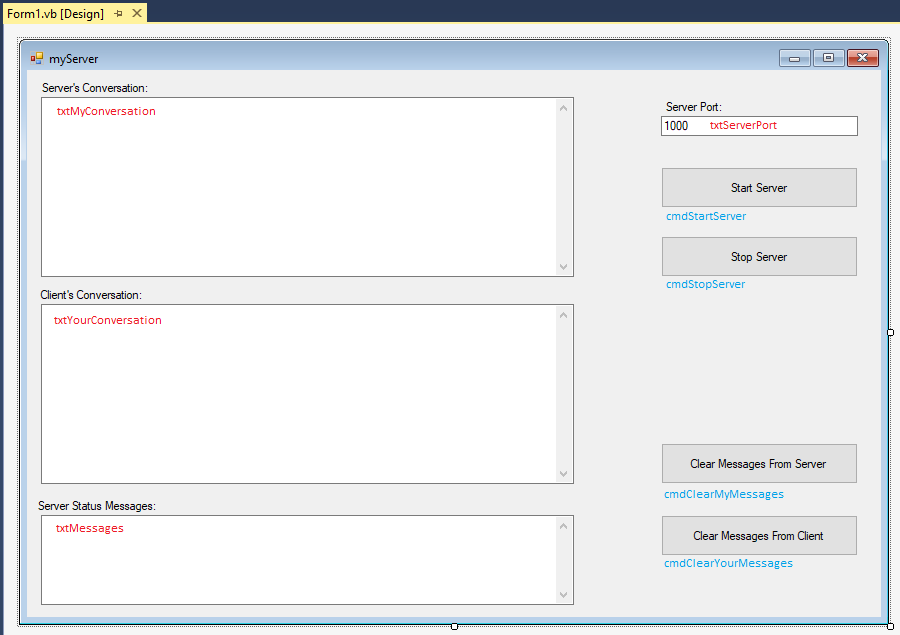
We introduced all that threading information to just get here. You might be questioning why you need to understand threads to write networking programs in VB. After all, we already examined networking in a previous chapter and there wasn't any need for threads in any of the code that we wrote there.

In this section, we will examine what VB provides for networking support with regard to writing our own programs implementing our own protocols. There are several classes available for our use and we saw many of them in Chapter 18. The main problem is that these were all “canned” classes created for a specific purpose. One class that you will find yourself commonly using when developing networked applications under .NET is the System.Net.Sockets namespace. We did not examine this class in Chapter 18.

The Sockets class supports both TCP (connected) and UDP (connectionless) network application development. That's the good news…the bad news is that you will need to spin your network listening portion off into its own thread. That's why it took so long to get you into what the .NET network model looks like.

## Simple Client/Server Chat Application

Now it's time to visit a Sockets based VB application. In this example, we will be building a chat server and a chat client. Both versions of the application have textboxes that allow two users to type text messages to each other. Let's begin by taking a look at the server's form:



The three larger TextBoxes are all multiline with a vertical ScrollBar. The uppermost textbox is where the server user will type in his or her conversation (txtMyConversation). The middle textbox is where the client's messages will appear (txtYourConversation). Finally, the last textbox (txtMessages) will show status messages about the state of the server.

The upper right TextBox (txtServerPort) allows a user to specify the port that the chat server is supposed to run on. By default, the application starts on port 1000, but you can change this value. Typically, you shouldn't use many ports below 1024 since there are standard network services that often run on those ports including SMTP, POP3, HTTP, FTP and so forth.

The four command buttons that are on the server’s right side are pretty much self-explanatory from the text on the buttons. The "Start Server" button (cmdStartServer) is responsible for starting the server up. Likewise, "Stop Server" (cmdStopServer) shuts the server down. The "Clear Messages From Server" button (cmdClearMyMessages) clears out the upper textbox (the server's conversation), while the "Clear Messages From Client" button (cmdClearYourMessages) clears out the middle textbox containing the client's conversation.

That's the description of the entire form. Let's now discuss how the application will work. Once the user's runs the application, he or she will press the "Start Server" button which starts up a server and begins listening for a client connection request. Once a client request is received, the server accepts the connection and the user is then able to type messages in the upper textbox to send to the client. We will programmatically change the ReadOnly property on the upper textbox to False and change the enabled state of the buttons once we have a valid connection to the client.

The user may stop communicating with the client anytime by pressing the "Stop Server" button. The server may also be stopped if the client decides to end the communication link. In either case, the "Start Server" button may be clicked again to restart the server listening for a new client connection request. This server application does not automatically restart itself.

All information that the user types from the server will be transmitted to the client over a network stream object between the two computers. Since the network stream object is simply a subclass of the normal stream object, we can attach a binary writer to send information to the client. Likewise, we will create a binary reader object to pull network traffic coming into the server. Since we never know when we will get data from the client, we will spin the code that reads the information (the binary reader) off into its own thread.

Now you can see why the threading piece was so important to learn. Because we don't know when we'll get information from the client, we could tie our program up waiting for information to show up. By placing the code handler for data arrival in its own thread, our server can continue to run independently from the data arrival thread. When data does show up, that's fine as our thread will handle it for us.

Now that we've discussed the basic logic for this application, let's go ahead and look at the source code. I've placed a lot of comments in the code to describe to you how the application works:

'Chapter 20 - Program 7 - Server

'This is the server side of a chat program

'We need to import all of the following for our chat program

Imports System.Threading

Imports System.Net.Sockets

Imports System.IO

Public Class frmServer

'This TcpListener object represents our server

Dim Server As TcpListener

'We need to create a Socket object to associate with our

'server. Our socket will run on port 1000 by default.

Dim aConnection As Socket

'We need a NetworkStream through which data is transferred

'between the client and the server

Dim NetStream As NetworkStream

'These are the objects that we will use for reading and

'writing data across the network stream

Dim NetWriter As BinaryWriter

Dim NetReader As BinaryReader

'We will have to start up a thread that specifically listens

'for network stream traffic coming to our server from the

'client. This is the thread object we will use for that

'purpose.

Dim GetDataThread As Thread

Private Sub frmServer\_Load(sender As Object, e As EventArgs)

Handles MyBase.Load

'Let's set up the form with the controls that need

'to be turned off or disabled set up correctly

txtMyConversation.ReadOnly = True

txtYourConversation.ReadOnly = True

txtMessages.ReadOnly = True

cmdStopServer.Enabled = False

cmdClearMyMessages.Enabled = False

cmdClearYourMessages.Enabled = False

'.NET gets mad if there are threads used across controls,

'so disable it... Again, not procedurally the correct way

'to do this, but okay for this program

CheckForIllegalCrossThreadCalls = False

End Sub

Private Sub cmdStartServer\_Click(sender As Object, e As EventArgs)

Handles cmdStartServer.Click

'This routine is called whenever the user clicks on the Start

'server button.

'Clear out all of the textboxes on the form

txtMyConversation.Text = ""

txtYourConversation.Text = ""

txtMessages.Text = ""

'Try to start up the server.

Try

txtMessages.Text &= "Starting Server..." & vbCrLf

'Create the server and point it at the port from

'the textbox value that the user entered.

Server = New TcpListener(Net.IPAddress.Parse("127.0.0.1"),

CInt(txtServerPort.Text))

Server.Start()

'At this point we are preparing to start the server

'so enable the Stop Server button and disable the

'Start Server button

cmdStopServer.Enabled = True

cmdStartServer.Enabled = False

'We wait here until we get a connection request from a

'client... The server is running and when we get a connection,

'we will accept it and place it in the Socket object we created.

txtMessages.Text &= "Listening for client connection..." & vbCrLf

Application.DoEvents()

aConnection = Server.AcceptSocket()

txtMessages.Text &= "...client connection accepted" & vbCrLf

'If we get to this point with no exceptions, then we have

'accepted a request from a client. Now we need to get the

'NetworkStream that is associated with our Socket object.

NetStream = New NetworkStream(aConnection)

'The last major setup piece that we need to do is to

'create objects for transferring data across the

'NetworkStream. Bind our Reader and Writer to the

'NetworkStream object

NetWriter = New BinaryWriter(NetStream)

NetReader = New BinaryReader(NetStream)

txtMessages.Text &= vbCrLf &

"Network stream and reader/writer objects created" &

vbCrLf

'Set up our thread to listen for data arriving from the

'client

txtMessages.Text &= "Preparing thread to watch for data..." & vbCrLf

GetDataThread = New Thread(AddressOf GetDataFromClient)

GetDataThread.Start()

'Now that our connection to the client has been established,

'all stream/reader/writer objects have been created and we

'have our data listener thread running, turn on the user's

'conversation textbox so that they can begin sending

'messages to the client.

txtMyConversation.ReadOnly = False

'Also activate the Clear buttons

cmdClearMyMessages.Enabled = True

cmdClearYourMessages.Enabled = True

'We need to worry about not being able to create the stream

'reader/writer

Catch IOEx As IOException

txtMessages.Text &= "Error in setting up Server -- Closing" & vbCrLf

'We need to worry about trying to create another server on the same

'port

Catch SocketEx As SocketException

txtMessages.Text &=

"Server already exists -- just restarting listening"

& vbCrLf

End Try

End Sub

Sub GetDataFromClient()

'This is the routine that we spin off into its own thread to

'listen for and retrieve network traffic coming to the server

'from the client.

'This is a string that we use to pull the data off of the

'network stream

Dim TheData As String

txtMessages.Text &= "Data watching thread active" & vbCrLf

'Here's the main listening loop that will continue until we

'receive the ~~END~~ message from the client or the connection

'abruptly stops.

Try

Do

'Pull data from the network into our string

TheData = NetReader.ReadString

'Shove the contents of the string into the textbox

txtYourConversation.Text = TheData

Loop While (TheData <> "~~END~~") And aConnection.Connected

StopListening()

'Errors can occur if we try to write to the client and it's

'not there

Catch IOEx As IOException

txtMessages.Text &= "Closing connection with client..." \_

& vbCrLf

StopListening()

End Try

End Sub

Private Sub cmdStopServer\_Click(sender As Object, e As EventArgs)

Handles cmdStopServer.Click

'Whenever the user presses the Stop Server button, we simply call

'the StopListening routine

StopListening()

End Sub

Sub StopListening()

'This is called whenever we try to stop listening to the client

txtMyConversation.ReadOnly = True

cmdStartServer.Enabled = True

cmdStopServer.Enabled = False

cmdClearMyMessages.Enabled = False

cmdClearYourMessages.Enabled = False

txtMessages.Text &= "Attempting to close connection to client..." & vbCrLf

'If we are still validly connected to the client, let it know

'that we are planning on ending our communication session with

'it. The string ~~END~~ is our way of saying the conversation

'is over.

Try

NetWriter.Write("~~END~~")

Catch Ex As Exception

'We don't need to do anything, but there was a problem

'communicating with the client

End Try

Try

'Destroy all of the objects that we created

NetWriter.Close()

NetReader.Close()

NetStream.Close()

Server.Stop()

NetWriter = Nothing

NetReader = Nothing

NetStream = Nothing

Server = Nothing

Try

GetDataThread.Abort()

Catch Ex As Exception

'We don't care since we are aborting thread

End Try

Catch Ex As Exception

'We don't have to do anything since we are leaving anyway

Finally

txtMessages.Text &= "Server has been stopped" & vbCrLf

End Try

End Sub

Private Sub cmdClearMyMessages\_Click(sender As Object, e As EventArgs)

Handles cmdClearMyMessages.Click

'Clears out the server's conversation textbox

txtMyConversation.Text = ""

End Sub

Private Sub cmdClearYourMessages\_Click(sender As Object, e As EventArgs)

Handles cmdClearYourMessages.Click

'Clears out the conversation from the client textbox

txtYourConversation.Text = ""

End Sub

Private Sub txtMyConversation\_TextChanged(sender As Object, e As EventArgs)

Handles txtMyConversation.TextChanged

'This routine is called anytime that the user changes any of the

'text in the myconversation textbox. It is responsible for

'sending the textbox contents to the client. While this is a

'very easy way to implement a chat program, it is not very

'efficient since every time that the user types a character we

'send the entire textbox's contents and not just the changed

'characters... There are several things ways that you could

'simplify this problem.

Try

'Assuming that our network connection to the client is

'still good, we should be able to send the textbox's

'contents over the network stream

NetWriter.Write(txtMyConversation.Text)

'We tried to write to the client and it wasn't there

Catch SocketEx As SocketException

txtMessages.Text &= "Error writing to client" & vbCrLf

Catch Ex As Exception

'Some other exception occurred, don't worry about it

'at this point.

End Try

End Sub

Private Sub frmServer\_FormClosing(sender As Object, e As FormClosingEventArgs)

Handles Me.FormClosing

'If the user closes the form, make sure we shut things down properly

StopListening()

End Sub

End Class

Let's stop and talk about how the main logic of the server works. We begin by creating a TcpListener object on the port that we want our server to run on. Once the server has been started by calling the TcpListener's Start method, we need to wait for a client connection, which we process by telling the TcpListener to perform its AcceptSocket method.

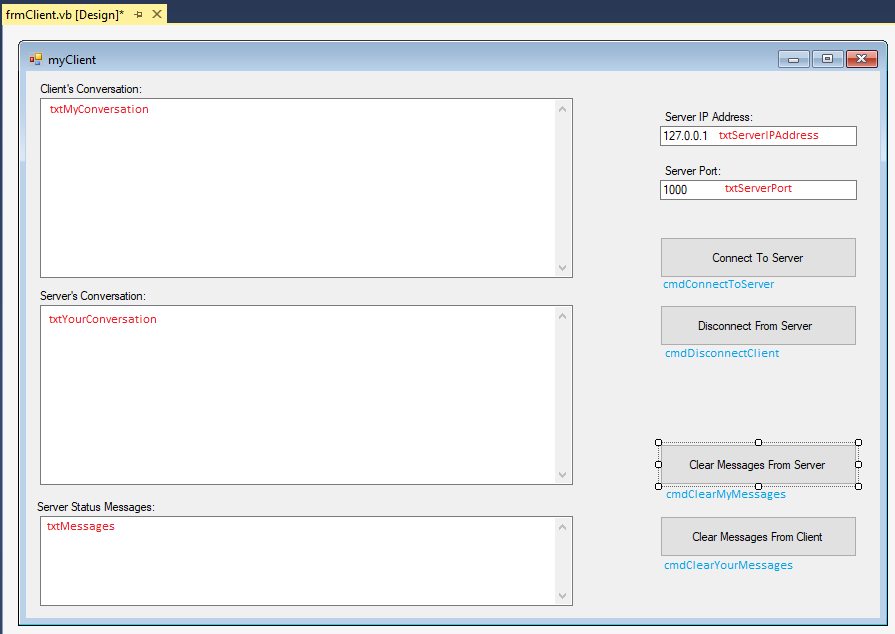
Our program will not continue until a connection is made. If this bothers you, you can fix the problem by spinning the server creation into a thread of its own just like the thread that reads the incoming network traffic.

After we receive and accept a connection, we set up a network stream to the client's socket that connected to our server. We then create binary reader and binary writer objects to get and send data across the network stream. Finally, we set up the thread that handles the data arrival of incoming network traffic.

The data arrival thread simply is a loop that reads the incoming data off the network stream using the binary reader. This information is then printed in the middle “your” conversation textbox. We won't exit this thread until one of three possible things happen: (1) the server user decides to stop the server, (2) we receive the "~~END~~" string from the client, indicating the user has stopped the client or (3) our network connection to the client stops for some reason.

As far as sending data to the client across the network stream goes, we use the binary writer. The TextChanged event on the upper “my” conversation textbox, where the server user types in his or her conversation, uses the binary writer to send the entire contents of the textbox across the stream. As I mentioned in the comments, this isn't the most efficient way to perform this job, but it's simple and it works. A better way would be to only send the data that has changed since the last send.

Now that you know how the server works, let's take a look at the client. We'll begin with a screenshot of the client's form:



If you don't see much different between the client form and the server form, you are correct. The three multiline textboxes are identical in name and purpose to those in the server. We have added one additional textbox on the right side of the client for the IP address of the server (txtServerIPAddress). The client needs to know the IP address of the server and the port that the server is listening on. Again, we defaulted to 1000 for the IP address and we are using the loopback address of 127.0.0.1 to allow the client and server to run on the same computer for testing purposes.

We replaced the start and stop server buttons with "Connect To Server" (cmdConnectToServer) and "Disconnect From Server" (cmdDisconnectClient) buttons. The clear message buttons are the same as in the server program.

Procedurally, our application works much the same way as the server. Assuming that the server program is already running, a user can start the client application and click on the "Connect To Server" button. The client will then create a TcpClient object which represents the client side of a client/server pair. The TcpClient object will then perform a Connect method to request a connection to the server.

Once the server accepts the connection, our client application continues by creating a network stream between the client and the server. We then create a binary reader and a binary writer to move data across the network stream. Finally, we spin off a thread to read the incoming network traffic from the server to the client.

The client code looks very similar to the server’s code. Again, you will find that the code contains lots of comments to explain what it does. Here's the source code for our chat client:

'Chapter 20 - Program 7 - Client

'This is the client side of a chat program

'We need to import all of the following for our chat program

Imports System.Threading

Imports System.Net.Sockets

Imports System.IO

Public Class frmClient

'This TcpClient object represents a client

Dim Client As TcpClient

'We need a NetworkStream through which data is transferred

'between the client and the server

Dim NetStream As NetworkStream

'These are the objects that we will use for reading and

'writing data across the network stream

Dim NetWriter As BinaryWriter

Dim NetReader As BinaryReader

'We will have to start up a thread that specifically listens

'for network stream traffic coming to our client from the

'server. This is the thread object we will use for that purpose.

Dim GetDataThread As Thread

Private Sub frmServer\_Load(sender As Object, e As EventArgs)

Handles MyBase.Load

'Let's set up the form with the controls that need

'to be turned off or disabled set up correctly

txtMyConversation.ReadOnly = True

txtYourConversation.ReadOnly = True

txtMessages.ReadOnly = True

cmdDisconnectClient.Enabled = False

cmdClearMyMessages.Enabled = False

cmdClearYourMessages.Enabled = False

'.NET gets mad if there are threads used across controls, so disable it.

CheckForIllegalCrossThreadCalls = False

End Sub

Private Sub cmdConnectToServer\_Click(sender As Object, e As EventArgs)

Handles cmdConnectToServer.Click

'This routine is called whenever the user clicks on the Connect

'to server button.

'Clear out all of the textboxes on the form

txtMyConversation.Text = ""

txtYourConversation.Text = ""

txtMessages.Text = ""

'Try to make a connection to the server

Try

txtMessages.Text &= "Attempting connection..." & vbCrLf

'Create the client and point it at the server's address

'and port from the textbox values that the user entered.

'We will get an exception here if the server is not already

'up and running.

Client = New TcpClient()

Client.Connect(txtServerIPAddress.Text,

CInt(txtServerPort.Text))

'If we get to this point with no exceptions, then we have

'requested a connection to the server and it was accepted.

'Now we need to get the NetworkStream that is associated

'with our TcpClient.

NetStream = Client.GetStream()

'The last major setup piece that we need to do is to

'create objects for transferring data across the

'NetworkStream. Bind the reader and writer.

NetWriter = New BinaryWriter(NetStream)

NetReader = New BinaryReader(NetStream)

txtMessages.Text &= vbCrLf &

"Network stream and reader/writer objects created" &

vbCrLf

'At this point we are connected, so enable the disconnect

'button and disable the connect button

cmdDisconnectClient.Enabled = True

cmdConnectToServer.Enabled = False

'Set up our thread to listen for data arriving from the

'server

txtMessages.Text &= "Preparing thread to watch for data..." & vbCrLf

GetDataThread = New Thread(AddressOf GetDataFromServer)

GetDataThread.Start()

'Now that our connection to the server has been established,

'all stream/reader/writer objects have been created and we

'have our data listener thread running, turn on the user's

'conversation textbox so that they can begin sending

'messages to the server.

txtMyConversation.ReadOnly = False

'Turn on the clear messages buttons

cmdClearMyMessages.Enabled = True

cmdClearYourMessages.Enabled = True

'Catch errors in trying to create the binary reader/writer

Catch IOException As IOException

txtMessages.Text &= "Error in setting up Client -- Closing" & vbCrLf

'Catch errors in trying to connect when a server is not present

Catch SocketEx As SocketException

txtMessages.Text &= "Cannot find server -- please try again later" &

vbCrLf

End Try

End Sub

Sub GetDataFromServer()

'This is the routine that we spin off into its own thread to

'listen for and retrieve network traffic coming to the client

'from the server.

'This is a string that we use to pull the data off of the

'network stream

Dim TheData As String

txtMessages.Text &= "Data watching thread active" & vbCrLf

'Here's the main listening loop that will continue until we

'receive the ~~END~~ message from the server or our connection

'abruptly stops.

Try

Do

'Pull data from the network into our string

TheData = NetReader.ReadString

'Shove the contents of the string into the textbox

txtYourConversation.Text = TheData

Loop While (TheData <> "~~END~~")

DisconnectClient()

'Catch errors when trying to read or write and the stream

'is not there

Catch IOEx As IOException

txtMessages.Text &= "Closing client connection..." \_

& vbCrLf

DisconnectClient()

End Try

End Sub

Private Sub cmdDisconnectClient\_Click(sender As Object, e As EventArgs)

Handles cmdDisconnectClient.Click

'Whenever the user presses the Disconnect button, we simply call

'the DisconnectClient routine

DisconnectClient()

End Sub

Sub DisconnectClient()

'This is called whenever we try to disconnect from the server

txtMyConversation.ReadOnly = True

cmdConnectToServer.Enabled = True

cmdDisconnectClient.Enabled = False

cmdClearMyMessages.Enabled = False

cmdClearYourMessages.Enabled = False

txtMessages.Text &= "Attempting to disconnect from server..." & vbCrLf

'If we are still validly connected to the server, let it know

'that we are planning on ending our communication session with

'it. The string ~~END~~ is our way of saying the conversation

'is over.

Try

NetWriter.Write("~~END~~")

Catch Ex As Exception

'We don't need to do anything, but there was a problem

'communicating with the server

End Try

Try

'Destroy all of the objects that we created

NetWriter.Close()

NetReader.Close()

NetStream.Close()

Client.Close()

NetWriter = Nothing

NetReader = Nothing

NetStream = Nothing

Client = Nothing

Try

GetDataThread.Abort()

Catch Ex As Exception

'We don't care since we are trying to stop the thread

End Try

Catch Ex As Exception

'We don't have to do anything since we are leaving anyway

Finally

txtMessages.Text &= "Disconnected...client closed" & vbCrLf

End Try

End Sub

Private Sub txtMyConversation\_TextChanged(sender As Object, e As EventArgs)

Handles txtMyConversation.TextChanged

'This routine is called anytime that the user changes any of the

'text in the myconversation textbox. It is responsible for

'sending the textbox contents to the server. While this is a

'very easy way to implement a chat program, it is not very

'efficient since every time that the user types a character we

'send the entire textbox's contents and not just the changed

'characters... There are several things ways that you could

'simplify this problem.

Try

'Assuming that our network connection to the server is

'still good, we should be able to send the textbox's

'contents over the network stream

NetWriter.Write(txtMyConversation.Text)

'We tried to write and we couldn't get to the server

Catch SocketEx As SocketException

txtMessages.Text &= vbCrLf & "Error writing to server"

Catch Ex As Exception

'Some other exception occurred, don't worry about it

'at this point.

End Try

End Sub

Private Sub cmdClearMyMessages\_Click(sender As Object, e As EventArgs)

Handles cmdClearMyMessages.Click

'Clears out the my conversation textbox

txtMyConversation.Text = ""

End Sub

Private Sub cmdClearYourMessages\_Click(sender As Object, e As EventArgs)

Handles cmdClearYourMessages.Click

'Clears out the conversation from the server textbox

txtYourConversation.Text = ""

End Sub

Private Sub frmServer\_FormClosing(sender As Object, e As FormClosingEventArgs)

Handles Me.FormClosing

'If the user closes the form, make sure we shut things down properly

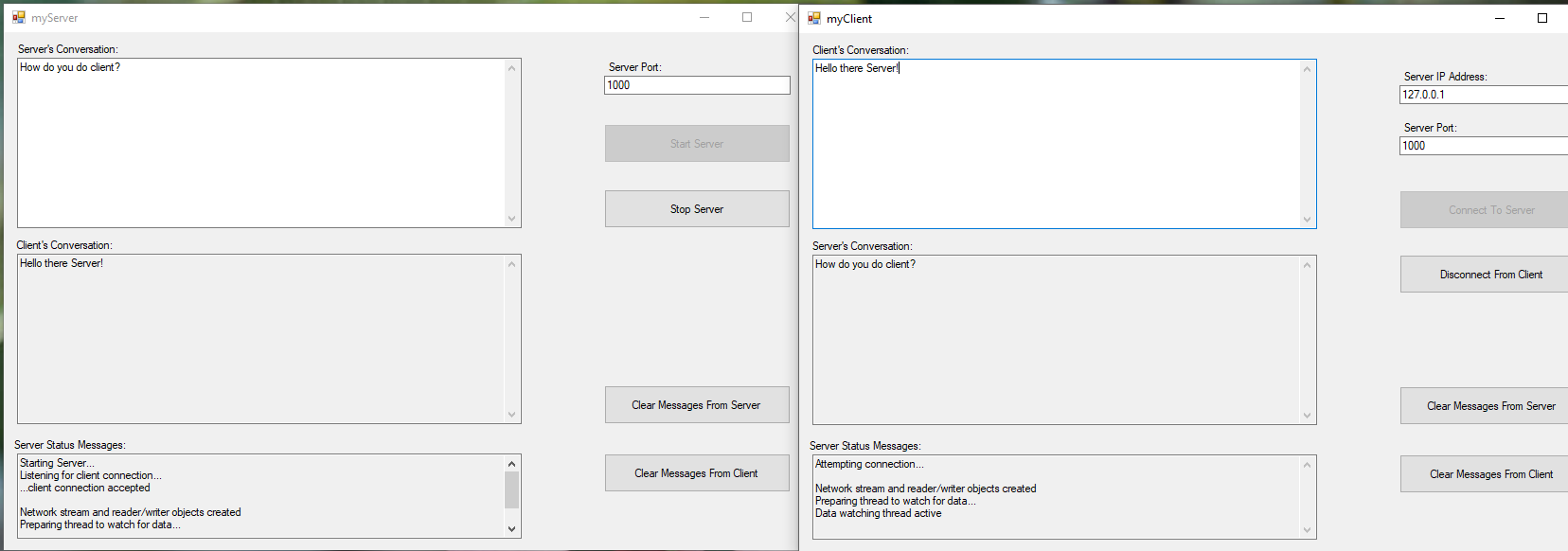
DisconnectClient()

End Sub

End Class

Okay, let's go ahead and run both applications. Make sure that the server is running and that you have pressed on the "Start Server" button before trying to click on the client's "Connect To Server" button. Once the connection is made, you can type in either application's upper textbox and your message will be instantly sent to the other application's middle textbox.

Here's a screenshot showing both applications in action:



Now that you've seen how to create a chat client and server, you could create almost any other type of server you want, such as an SMTP client, a POP3 client, a time server, an HTTP client and so forth. Every one of those protocols has a defined set of standards that discuss how communications are to take place. Essentially all that you would need to do is modify your data arrival thread to look for specific keywords from the protocol and then to take the appropriate action when those keywords arrive. Remember that we already have access to a ton of built in network classes, so it’s a waste of time to reinvent what’s already there as a socket application.

Network programming is a lot of fun and it can sometimes be very challenging. It also is an integral part of today's computing world. You may find that you will need to write this type of code since application users of today expect programs to be network-saavy.

We will end this chapter with a review of the classes that we used for our network programming.

Here is the list of useful TCPClient properties and methods:

|  |  |
| --- | --- |
| *Method/Property* | *Purpose* |
| Active | A Boolean value that indicates if a connection has been made |
| BeginConnect | Starts an asynchronous request for a remote host connection |
| Available | Gets the amount of data that has been received from the network that is available to be read |
| Client | Gets or sets the Socket object related to this TcpClient object |
| Close | Closes the TCP connection |
| Connect | Connects to a TCP host using the specified port |
| ConnectAsync | Connects the client to the specified port on the specified host asynchronously |
| Connected | Gets a value indicating whether the client is connected or not |
| EndConnect | Asynchronously accepts an incoming connection attempt |
| ExclusiveAddressUse | Gets or sets whether the TCPClient only allows one client to use a port |
| GetStream | Gets the stream used for reading and writing through the socket |
| LingerState | Gets or sets information about the linger state of the associated socket |
| NoDelay | Gets or sets a value that disables a delay when send or receive buffers are not full |
| ReceiveBufferSize | Gets or sets the receive buffer size |
| ReceiveTimeout | Gets or sets the receive timeout in milliseconds |
| SendBufferSize | Gets or sets the send buffer size |
| SendTimeout | Gets or sets the send timeout in milliseconds |

Here’s a list of selected methods and properties from the TcpListener class:

|  |  |
| --- | --- |
| Method/Property | Purpose |
| AcceptSocket | Waits for a connection from a client and then returns a socket |
| AcceptSocketAsync | Accepts a pending connection request as an asynchronous operation |
| AcceptTcpClient | Waits for a client to connect and then returns a TcpClient |
| AcceptTCPClientAsync | Accepts a pending connection request as an asynchronous operation |
| Active | A Boolean value that indicates if a connection has been made |
| AllowNatTravel | Enables or disables NAT traversal on a TCPListener instance |
| BeginAcceptSocket | Begins an asynchronous operation to accept an incoming connection attempt |
| BeginAcceptTcpClient | Begins an asynchronous operation to accept an incoming connection attempt |
| EndAcceptSocket | Asynchronously accepts an incoming connection attempt and creates a new socket to handle remote host communication |
| EndAcceptTcpClient | Asynchronously accepts an incoming connection attempt and creates a new TCPClient to handle remote host communication |
| ExclusiveAddressUse | Gets or sets whether the TCPListener allows only one underlying socket to listen to a specific port |
| LocalEndpoint | Gets the active endpoint for the TcpListener socket, which is an IP address and the port number |
| Pending | A Boolean value indicating if there are any pending connection requests |
| Server | Gets or sets the Socket object related to this TcpListener object |
| Start | This method starts the TcpListener listening for network requests |
| Stop | This method stops the TcpListener from listening |