

## HowTo PortScan in C#

by Randy Charles Morin

This is the fourth article in a series on Internet Programming with C#. The first three articles addressed Internet Protocols; SMTP, POP3 and NNTP. This article will address a popular utility that is commonly written in all new programming languages, that is, the PortScan.

The PortScan involves attempting to perform a TCP connect on a range of ports of a given server and reporting which ports are open to receive connections. The PortScan is often used to determine which open ports could be used to attack a given server over the Internet.

I'll be using two classes in C# that I find extremely interesting. The TcpClient class will be used to attempt TCP connections with the target server and the ThreadPool class will be used to speed the utility by attempting to connect on many threads simultaneously.

The ThreadPool class works by placing wares on a queue and allowing the pool of threads to handle each ware. Let me first define ware as the item being consumed in the consumer design pattern. In this case, the ware is the port number that we will attempt to connect to.

### Listing 1: Definition of Ware

```
public class Ware
{
    public int id;
    public Ware(int _id)
    {
        id = _id;
    }
}
```

Next we'll create a generic class with two data members.

### Listing 2: Data Members of our Class

```
class Class1
{
    public string ServerName;
    public int QueueLength;

    public Class1()
    {
        QueueLength = 0;
    }
}
```

The ServerName is a string representation of the IP address or domain name for our server. The QueueLength is the current number of ports not yet processed by our utility. It's important to maintain this number because we don't want utility to exit before we are finished processing all the ports. The threads in the ThreadPool class are by default background threads. Background threads in the dot-NET framework will be terminated

once all foreground threads have completed. As such, we hold our primary foreground thread open until the background threads are finished doing their work.

We need two methods in our class. The first method Produce is used to add wares onto our consumption queue. The second method Consume is used to remove wares from our consumption queue.

### Listing 3: Produce Method

```
public void Produce(Ware ware)
{
    ThreadPool.QueueUserWorkItem(
        new WaitCallback(Consume), ware);
    QueueLength++;
}
```

The Produce method calls a static method in the ThreadPool class. The QueueUserWorkItem method adds a ware to the consumption queue. It's important that we specify a delegate and the ware in the QueueUserWorkItem method call. This delegate wraps the method that will consume our ware.

The ThreadPool class will then allocate a pool of consumer threads and that will begin consuming wares from our consumption queue. When a consumer thread pops a ware from our consumption queue, it then calls into our second Consume method.

### Listing 4: Consume Method

```
public void Consume(Object obj)
{
    Ware ware = (Ware) obj;
    if (ware.id % 100 == 0)
    {
        Console.WriteLine("Thread {0} consumes {1}",
            Thread.CurrentThread.GetHashCode(), //{0}
            ((Ware)obj).id); //{1}
    };

    try
    {
        TcpClient client = new TcpClient();
        client.Connect(ServerName, ware.id);
    }
    catch(Exception )
    {
        QueueLength--;
        return;
    };

    Console.WriteLine("Server {0}, Port {1}", ServerName, ware.id);
    QueueLength--;
}
```

The Consume method can be called by a number of different consumer threads. In order to see this in action, we will display the thread id and ware id for every 100 wares. This will also help us see the progress of our port scanning.

Next our Consume method attempts to connect to the server at the specified port. If the connection fails, then the exception is caught and the method ends. But if the connection succeeds, then we print a message indicating that we successfully connected to the server on the said port. In either case, we decrement the QueueLength.

Lastly, we will define our main function. This is where we will push the wares onto our consumption queue.

**Listing 5: Main Function**

```
static void Main(string[] args)
{
    if (args.Length < 1)
    {
        Console.WriteLine("Usage: portscan [servername]");
        return;
    }

    Console.WriteLine("TCP Pinging {0}", args[0]);

    Class1 obj = new Class1();
    obj.ServerName = args[0];

    for (int i=1;i<256*256;i++)
    {
        obj.Produce(new Ware(i));
    };

    while (obj.QueueLength!= 0)
    {
        Thread.Sleep(1000);
    };
}
```

We begin our main function by checking if the parameter count is less than 1. If it is, then we show the user how to use our function by displaying the usage. Otherwise, we print the name of the server that we will be scanning and begin populating our consumption queue.

We populate the consumption queue by allocating the Class1 object, setting the ServerName member from the program arguments and we repeatedly call the Produce method with the ports numbers. Finally we loop until the QueueLength reaches zero.

The utility will output the thread and port for each port divisible by 100. It will also output all the ports where we were able to successfully perform a TCP connection.

**Listing 6: Utility Output**

```
TCP Pinging www.kbcafe.com
Server www.kbcafe.com, Port 80
Thread 40 consumes 100
Thread 41 consumes 200
Thread 41 consumes 300
Thread 41 consumes 400
Server www.kbcafe.com, Port 443
Thread 42 consumes 500
```

For more on the PortScan you might want to check out previously articles I've written on the subject [<http://www.kbcafe.com/articles/port.scan.html>].

## **About the Author**

Randy Charles Morin is the Chief Architect of SportMarkets Development from Toronto, Canada and lives with his wife, Bernadette and two kids, Adelaine and Brayden in Brampton, Canada. He is the author of the KBCafe.com website [<http://www.kbcafe.com>], many programming books and many articles.