

Re: Test Availability of TCP Port?

Source: <http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.framework/2004-02/0283.html>

From: William Stacey [MVP] (*staceywREMOVE_at_mvps.org*)

Date: 02/12/04

Date: Thu, 12 Feb 2004 14:55:18 -0500

That gets into some fun stuff. It's also a very big, and not so easy, a subject as you know. Obviously, many books are written just on that subject alone. Funny you bring that up, as I have been thinking about this for a server I am doing and thinking about threads and queues, etc. I am not saying this is the best model (or even good for your needs), but I am thinking about:

- 1) Keeping the thread count as low as reasonable and leveraging the work each thread can do.
- 2) Keep each thread working as long and as fast as it can in its time quantum (i.e. limit forced sleeps or unneeded waits, etc.)
- 3) Keep separation of duty to simplify the logic (going towards your point which makes a lot of sense to me.). Keep async and delegates to a minimum, if another option may be better in terms of other points here.
- 4) Block/wait on IO when nothing to do (i.e. no polling.)

Cook that up, and I think it lends itself to a consumer/producer model with blocking circular queues. This is pretty similar to how the TCP/IP stack works, so it must be an ok model. You have a consumer thread receiving/wait on input from your socket in a tight loop. Its only job in life is to get data off the wire and put it into an inQueue as fast as it can. You might have a "server" object that blocks/waits on the inQueue. This object actually does something with the object, such as parse it, validate it, *drop it on the floor, or build another object and enqueue that object into another queue – say a Router input queue that your Router Object does something with. The really neat and clean thing about this model, is the server object can be hooked-up to many different output queues. So the server could send objects to a Forwarder inQueue or a Routing inQueue or the outQueue that the Reply thread is waiting on to return result (i.e. error, bad format, etc) to the user and the user's endpoint. You carry around the client's endpoint (in the case of udp for example) in the object when you get in the first input queue. You may have one thread that does nothing but block on outQueue waiting for an object. It then dequeues that object and sends it over the wire as a reply to the user. You can also send Control Messages between your threads using the queues (i.e. shutdown messages, pause, notify, etc.) So it allows you to leverage the infrastructure for general "inband" communication without resorting to a seperate "out-of-band" signal solution. You can also pass counter info from each "object" to a

microsoft.public.dotnet.framework: Re: Test Availability of TCP Port?

perfObject and/or Remoting object that allows GUI clients of your server to enumerate your servers function and perf data – all with your queues.

Performance wise...who knows. You would have to test other designs for your server to know for sure, but we can infer things. In this model, your threads block doing nothing, when no work is in the queues – and this is good. Your not polling wasting cpu. You have separation of duty and letting the cpu divide the work amongst your threads – a fair and balanced approach. You also don't have threads competing to do the same thing. For example, it probably does not make sense to have two threads blocking on udp socket receives, as two threads would only compete for the same job and end up doing less work overall. On that note, you also gain real advantage of multiple cpu's in a clean manner as one thread can be filling the input queue and your server thread object, draining it and doing other work without running into each other. The contention is for the queue locks, so we need to pick or build a good *blocking queue (fixed # of slots) with fast locks (i.e. spin locks, CS, etc. which I have spent some time on if you care to review it.) You may need to resort to async io for certain things that may be cleaner then this, but general design is CQ between threads as a common message passing infrastructure – in effect, we have a "Pipe" between our consumers/producers.

Interesting subject, so please feel free to add, change, delete. Cheers!

--

William Stacey, MVP

"Kevin Z Grey" <anonymous@discussions.microsoft.com> wrote in message news:26B63E9A-ADED-4927-A8F9-914D3DF039E8@microsoft.com...

> Neither. Something on the scale of multi-threaded message routing for a VOIP proxy.

>

> My server basically handles messages. Certain messages get treated differently and have different end-points. I have one object per "user" connection that handles all the logic for that connection. I'm doing it this way to limit the inter-thread communication. The more thread to thread chatter I have, the more likely a deadlock will occur (even when designed properly, software has bugs).

>

> The issue at hand (for me at least) isn't how to use TCP or how to properly use TCP or how to properly listen, etc, etc. The issue that I am grappling with is proper server design with respect to the .NET Framework. How to properly pass messages between threads, etc. But all of these are future enhancements to the already existing server.