Tips / techniques for high-performance C# server sockets

Asked 16 years ago Modified 12 years, 1 month ago Viewed 29k times

33

I have a .NET 2.0 server that seems to be running into scaling problems, probably due to poor design of the socket-handling code, and I am looking for guidance on how I might redesign it to improve performance.

Usage scenario: 50 - 150 clients, high rate (up to 100s / second) of small messages (10s of bytes each) to / from each client. Client connections are long-lived - typically hours. (The server is part of a trading system. The client messages are aggregated into groups to send to an exchange over a smaller number of 'outbound' socket connections, and acknowledgment messages are sent back to the clients as each group is processed by the exchange.) OS is Windows Server 2003, hardware is 2 x 4-core X5355.

Current client socket design: A `TcpListener` spawns a thread to read each client socket as clients connect. The threads block on `Socket.Receive`, parsing incoming messages and inserting them into a set of queues for processing by the core server logic. Acknowledgment messages are sent back out over the client sockets using async `Socket.BeginSend` calls from the threads that talk to the exchange side.

Observed problems: As the client count has grown (now 60-70), we have started to see intermittent delays of up to 100s of milliseconds while sending and receiving data to/from the clients. (We log timestamps for each acknowledgment message, and we can see occasional long gaps in the timestamp sequence for bunches of acks from the same group that normally go out in a few ms total.)

Overall system CPU usage is low (< 10%), there is plenty of free RAM, and the core logic and the outbound (exchange-facing) side are performing fine, so the problem seems to be isolated to the client-facing socket code. There is ample network bandwidth between the server and clients (gigabit LAN), and we have ruled out network or hardware-layer problems.

Any suggestions or pointers to useful resources would be greatly appreciated. If anyone has any diagnostic or debugging tips for figuring out exactly what is going wrong, those would be great as well.

Note: I have the MSDN Magazine article Winsock: Get Closer to the Wire with High-Performance Sockets in .NET, and I have glanced at the Kodart "XF.Server" component - it looks sketchy at best.

c# .net performance sockets

Share

asked Nov 26, 2008 at 4:17

Improve this question

Follow

McKenzieG1

14.2k ● 7 ● 38 ● 42

10 Answers

Sorted by: Highest score (default)

22

Socket I/O performance has improved in .NET 3.5 environment. You can use ReceiveAsync/SendAsync instead of BeginReceive/BeginSend for better performance. Chech this out:

http://msdn.microsoft.com/en-us/library/bb968780.aspx

Share Improve this answer

Follow

answered Dec 23, 2008 at 4:33

hakkyu

> Thanks for the link. We probably won't be on 3.5 for a while (for lots of reasons), but when we do switch I will take another look at these new methods. – McKenzieG1 Jan 9, 2009 at 17:37

18

A lot of this has to do with many threads running on your system and the kernel giving each of them a time slice. The design is simple, but does not scale well.

You probably should look at using Socket.BeginReceive which will execute on the .net thread pools (you can specify somehow the number of threads it uses), and then pushing onto a queue from the asynchronous

callback (which can be running in any of the .NET threads). This should give you much higher performance.

Share Improve this answer

Follow

answered Nov 26, 2008 at 4:52

grepsedawk

6,059 ● 5 ● 26 ● 22

> Agreed, though I might add that even though you "ruled out" network issues, I would consider swapping out various pieces (esp. the server nic) and making sure you have all the latest firmware and drivers. – Jason Hernandez Nov 26, 2008 at 5:26

8

A thread per client seems massively overkill, especially given the low overall CPU usage here. Normally you would want a small pool of threads to service all clients, using BeginReceive to wait for work async - then simply despatch the processing to one of the workers (perhaps simply by adding the work to a synchronized queue upon which all the workers are waiting).

Share Improve this answer

Follow

answered Nov 26, 2008 at 4:56

Marc Gravell

1.1m ● 272 ● 2.6k ● 3k

6

I am not a C# guy by any stretch, but for high-performance socket servers the most scalable solution is to use I/O Completion Ports with a number of active threads appropriate for the CPU(s) the process s running

on, rather than using the one-thread-per-connection model.

In your case, with an 8-core machine you would want 16 total threads with 8 running concurrently. (The other 8 are basically held in reserve.)

Share Improve this answer Follow

edited Nov 26, 2008 at 5:36

answered Nov 26, 2008 at 5:15

John Dibling

101k ● 32 ● 191 ● 331

6 CLR already uses I/O completion ports for sockets. So, you get that benefit by default on .NET. – feroze Nov 29, 2009 at 15:53

WCF will also use IO Completion ports for answering each of your service calls. But it's a good point to make that lightweight IO ports are specifically designed for this task. – Spence Aug 26, 2010 at 11:15

4

The `Socket.BeginConnect` and `Socket.BeginAccept` are definitely useful. I believe they use the `ConnectEx` and `AcceptEx` calls in their implementation. These calls wrap the initial connection negotiation and data transfer into one user/kernel transition. Since the initial send/recieve buffer is already ready the kernel can just send it off - either to the remote host or to userspace.

They also have a queue of listeners/connectors ready which probably gives a bit of boost by avoiding the latency involved with userspace accepting/receiving a connection and handing it off (and all the user/kernel switching).

To use `BeginConnect` with a buffer it appears that you have to write the initial data to the socket before connecting.

Share Improve this answer Follow

edited Aug 24, 2009 at 4:16

answered Nov 26, 2008 at 5:09

Luke Quinane

16.6k ● 13 ● 72 ● 89

4

As others have suggested, the best way to implement this would be to make the client facing code all asynchronous. Use BeginAccept() on the TcpServer() so that you dont have to manually spawn a thread. Then use BeginRead()/BeginWrite() on the underlying network stream that you get from the accepted TcpClient.

However, there is one thing I dont understand here. You said that these are long lived connections, and a large number of clients. Assuming that the system has reached steady state, where you have your max clients (say 70) connected. You have 70 threads listening for the client packets. Then, the system should still be responsive.

Unless your application has memory/handle leaks and you are running out of resources so that your server is paging. I would put a timer around the call to Accept() where you kick off a client thread and see how much time that takes. Also, I would start taskmanager and PerfMon, and monitor "Non Paged Pool", "Virtual Memory", "Handle Count" for the app and see whether the app is in a resource crunch.

While it is true that going Async is the right way to go, I am not convinced if it will really solve the underlying problem. I would monitor the app as I suggested and make sure there are no intrinsic problems of leaking memory and handles. In this regard, "BigBlackMan" above was right - you need more instrumentation to proceed. Dont know why he was downvoted.

Share Improve this answer

Follow

answered Nov 29, 2009 at 16:02

feroze

7,564 ● 7 ● 43 ● 64

3

Random intermittent ~250msec delays might be due to the Nagle algorithm used by TCP. Try disabling that and see what happens.

Share Improve this answer

Follow

answered Jan 18, 2010 at 18:04

Addys

2,501 ● 15 ● 24

1

One thing I would want to eliminate is that it isn't something as simple as the garbage collector running. If all your messages are on the heap, you are generating 10000 objects a second.

Take a read of Garbage Collection every 100 seconds

The only solution is to keep your messages off the heap.

Share Improve this answer

Follow

edited May 23, 2017 at 12:09

Community Bot

1 ● 1

answered Feb 3, 2011 at 10:43

Tom Thorne

214 ● 2 ● 9

0

I had the same issue 7 or 8 years ago and 100ms to 1 sec pauses , the problem was Garbage Collection .. Had about 400 Meg in use from 4 gig BUT there were a lot of objects.

I ended up storing messages in C++ but you could use ASP.NET cache (which used to use COM and moved them out of the heap)

Share Improve this answer

Follow

answered Nov 3, 2012 at 10:54

user1496062

1,317 ● 1 ● 8 ● 22

-1

I don't have an answer but to get more information I'd suggest sprinkling your code with timers and logging avg and max time taken for suspect operations like adding to the queue or opening a socket.

At least that way you will have an idea of what to look at and where to begin.

Share Improve this answer

Follow

answered Nov 26, 2008 at 4:53

mjallday

10.1k ● 9 ● 53 ● 73