Why do SocketChannel writes always complete for the full amount even on non-blocking sockets?

Asked 16 years, 2 months ago Modified 12 years, 8 months ago

Viewed 3k times

5

Using the Sun Java VM 1.5 or 1.6 on Windows, I connect a non-blocking socket. I then fill a `ByteBuffer` with a message to output, and attempt to `write()` to the SocketChannel.

I expect the write to complete only partially if the amount to be written is greater than the amount of space in the socket's TCP output buffer (this is what I expect intuitively, it's also pretty much my understanding of the docs), but that's not what happens. The `write()` *always* seems to return reporting the full amount written, even if it's several megabytes (the socket's SO_SNDBUF is 8KB, much, much less than my multi-megabyte output message).

A problem here is that I can't test the code that handles the case where the output is partially written (registering an interest set of `WRITE` to a selector and doing a `select()` to wait until the remainder can be written), as that case never seems to happen. What am I not understanding?

java sockets network-programming nio

Share

Improve this question

Follow

asked Oct 1, 2008 at 15:11

jpdaigle

1,265 ● 10 ● 12

Update: I've tried the Sun VM 1.5, 1.6 and JRockit VM 1.5 to make sure this network behaviour wasn't VM-specific.
– jpdaigle Oct 1, 2008 at 15:25

6 Answers

Sorted by: Highest score (default)

7

I managed to reproduce a situation that might be similar to yours. I think, ironically enough, your recipient is consuming the data faster than you're writing it.

```
import java.io.InputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class MyServer {
  public static void main(String[] args) throws Except
    final ServerSocket ss = new ServerSocket(12345);
    final Socket cs = ss.accept();
    System.out.println("Accepted connection");

    final InputStream in = cs.getInputStream();
    final byte[] tmp = new byte[64 * 1024];
    while (in.read(tmp) != -1);

    Thread.sleep(100000);
  }
}
```

```
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;

public class MyNioClient {
  public static void main(String[] args) throws Except
    final SocketChannel s = SocketChannel.open();
    s.configureBlocking(false);
    s.connect(new InetSocketAddress("localhost", 12345
    s.finishConnect();

    final ByteBuffer buf = ByteBuffer.allocate(128 * 1
    for (int i = 0; i < 10; i++) {
      System.out.println("to write: " + buf.remaining(
s.write(buf));
      buf.position(0);
    }
    Thread.sleep(100000);
  }
}
```

If you run the above server and then make the above client attempt to write 10 chunks of 128 kB of data, you'll see that every write operation writes the whole buffer without blocking. However, if you modify the above server not to read anything from the connection, you'll see that only the first write operation on the client will write 128 kB, whereas all subsequent writes will return `0`.

Output when the server is reading from the connection:

```
to write: 131072, written:  131072
to write: 131072, written:  131072
to write: 131072, written:  131072
...
```

Output when the server is not reading from the connection:

```
to write: 131072, written:  131072
to write: 131072, written:  0
to write: 131072, written:  0
...
```

Share Improve this answer

Follow

answered Oct 1, 2008 at 16:12

Alexander

9,380 ● 2 ● 28 ● 23

2

I've been working with UDP in Java and have seen some really "interesting" and completely undocumented behavior in the Java NIO stuff in general. The best way to determine what is happening is to look at the source which comes with Java.

I also would wager rather highly that you might find a better implementation of what you're looking for in any other JVM implementation, such as IBM's, but I can't guarantee that without look at them myself.

Share Improve this answer

Follow

answered Oct 1, 2008 at 15:27

Spencer Kormos

8,431 ● 3 ● 30 ● 47

0

I'll make a big leap of faith and assume that the underlying network provider for Java is the same as for C...the O/S allocates more than just `SO_SNDBUF` for every

socket. I bet if you put your send code in a for(1,100000) loop, you would eventually get a write that succeeds with a value smaller than requested.

Share Improve this answer

Follow

answered Oct 1, 2008 at 15:16

Clay

1,157 ● 1 ● 9 ● 21

1 Thanks for the idea. I just did a few hundred multi-megabyte writes and not a single one of them returned with a value less than the full amount. – jpdaigle Oct 1, 2008 at 15:23

Any chance you can watch the traffic with a sniffer and see how the packets are being sent? Might give you a clue if the sending library is even trying to break up the packets into smaller than MTU chunks. – stu Oct 1, 2008 at 21:55

0

You really should look at an NIO framework like MINA or Grizzly. I've used MINA with great success in an enterprise chat server. It is also used in the Openfire chat server. Grizzly is used in Sun's JavaEE implementation.

Share Improve this answer

Follow

answered Oct 2, 2008 at 2:36

Heath Borders

32k ● 21 ● 156 ● 265

0

Where are you sending the data? Keep in mind that the network acts as a buffer that is at least equal in size to your SO_SNDBUF plus the receiver's SO_RCVBUF. Add

this to the reading activity by the receiver as mentioned by Alexander and you can get a lot of data soaked up.

Share Improve this answer

Follow

answered Oct 2, 2008 at 17:30

Darron

21.6k ● 5 ● 51 ● 54

Well yes but in TCP there can't be any data in flight that isn't also in the send buffer. – user207421 Nov 10, 2011 at 22:19

0

I can't find it documented anywhere, but IIRC[1], send() is guaranteed to either a) send the supplied buffer completely, or b) fail. It will never complete the send partially.

[1] I've written multiple Winsock implementations (for Win 3.0, Win 95, Win NT, etc), so this may be Winsock-specific (rather than generic sockets) behavior.

Share Improve this answer

Follow

answered Apr 2, 2012 at 8:56

keithmo

4,943 ● 1 ● 22 ● 18

This is not correct. In non-blocking mode it can return `0 <= n <= length`. See the Javadoc. – user207421 Apr 22, 2021 at 4:13