## Netty (Doesn't?) suck at UDP Servers

I've been using Netty 4 (and the now defunct 5 alpha) at work for a UDP server that receives messages from OBD devices. So far my experience has been spotty, and here's why...

# The UDP Example

Here's the UDP Quote Of The Moment example from [Netty's own guide](http://netty.io/wiki/):

*public final class QuoteOfTheMomentServer {*

*private static final int PORT = Integer.parseInt(System.getProperty("port", "7686"));*

*public static void main(String[] args) throws Exception {*

*EventLoopGroup group = new NioEventLoopGroup();*

*try {*

*Bootstrap b = new Bootstrap();*

*b.group(group)*

*.channel(NioDatagramChannel.class)*

*.option(ChannelOption.SO\_BROADCAST, true)*

*.handler(new QuoteOfTheMomentServerHandler());*

*b.bind(PORT).sync().channel().closeFuture().await();*

*} finally {*

*group.shutdownGracefully();*

*}*

*}*

*}*

With something akin to the above code on our test environment we were seeing ~250 packets/sec being dealt with, even if an external executor group was used for our blocking code. After a few tests had been ran we noticed that running our application multiple times on different ports could easily triple our throughput, which signalled something wasn't quite right within our application.

The fact that Netty's own guide gets things wrong is more a result of the actual problem rather than bad documentation - and this is probably why it's not been fixed. The problem itself is that even with an NioEventLoopGroup with 2\*Cores threads, the DatagramChannel will only use one thread. It's a simple consequence of the way Netty handles channels - one thread for any given channel. UDP is connectionless and so we only ever have the main DatagramChannel. Having the boss thread delegate packets to worker threads would solve this problem (very easy to say, much harder to actually implement), but this hasn't made it's way into Netty yet.

The simple answer here is to just re-bind the event group 2\*Cores times and all the threads will be used. Whilst this sounds like a simple solution, we actually need to consider some options and compatibility issues first...

# SO\_REUSEADDR? No wait, it's SO\_REUSEPORT...

If we try to rebind to a port that's already been bound we'll just get a SocketException: Address already in use quite obviously. To get around this we can set the only cross platform option .option(ChannelOption.SO\_REUSEADDR, true) and then we can bind multiple times. Okay! We're done right?

Not exactly... SO\_REUSEADDR is great for broadcast/multicast packets, but in my use case the OS layer will only deliver packets to one of the bound sockets and no others. There's also the issue that the implementation of SO\_REUSEADDR is not well defined at all, to the point where on Windows another application can bind a socket to the port without this option set and kick all the previous bound sockets off... but that's just one of many reasons Windows is rarely used for servers (Although there is a fix for this).

Now there's two solutions we can use to fix this issue, the first is a nice and simple bind to many different ports and let a load balancer do all the heavy lifting on port mapping. It's not a nice solution, but it's the only one that's available on every system.

Now the actual solution is the socket option SO\_REUSEPORT which will happily distribute packets to listening sockets in a fair fashion. Of course this comes with downsides though, it's actually not available through Netty on anything but Linux (3.9+) through Netty's native epoll transport layer. On top of this, like SO\_REUSEADDR it's not well defined across platforms, and Windows doesn't have the option at all, since for Windows SO\_REUSEADDR also does this.

Below is a Linux only implementation which will ensure all threads are used:

*public final class QuoteOfTheMomentServer {*

*private static final int PORT = Integer.parseInt(System.getProperty("port", "7686"));*

*private static final int THREADS = Runtime.getRuntime().availableProcessors() \* 2; // Default EventLoopGroup Size*

*public static void main(String[] args) throws Exception {*

*EventLoopGroup group = new EpollEventLoopGroup(THREADS);*

*try {*

*Bootstrap b = new Bootstrap();*

*b.group(group)*

*.channel(EpollDatagramChannel.class)*

*.option(ChannelOption.SO\_BROADCAST, true)*

*.option(EpollChannelOption.SO\_REUSEPORT, true)*

*.handler(new QuoteOfTheMomentServerHandler());*

*List<ChannelFuture> futures = new ArrayList<>(THREADS);*

*// Bind THREADS times*

*for(int i = 0; i < THREADS; ++i) {*

*futures.add(bootstrap.bind(host, port).await());*

*}*

*// Now wait for all to be closed (if ever)*

*for (final ChannelFuture future : futures) {*

*future.channel().closeFuture().await();*

*}*

*} finally {*

*group.shutdownGracefully();*

*}*

*}*

*}*

Since many of the developers I work with aren't too familiar with Linux, it was important that we fell back to the Nio implementation on other operating systems, which creates somewhat messy and confusing platfom specific code. Not really something expected in Java. It also leaves a TODO for some future developer to deal with:

\* TODO - Evaluate JDK9 for SO\_REUSEPORT availability and remove platform specific code