**Brief decription of Netty code and architecture**

From the book Netty in Action

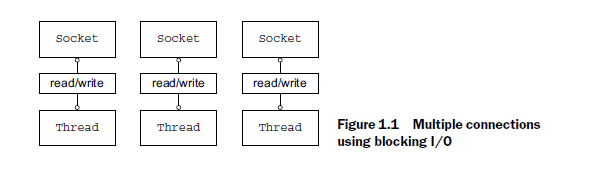
Netty is an asynchronous event-driven **network** application framework for rapid development of maintainable **high performance** protocol servers & clients

* You don’t have to be a networking expert to build applications with Netty.
* Using Netty is much easier than using the underlying Java APIs directly.
* Netty promotes good design practices, such as keeping your application logic decoupled from the network layer.

High-performance systems that can scale to 100,000 or more users require more than first-class coding skills; they demand expertise in several complex areas: networking, multithreading, and concurrency. Netty captures this domain knowledge in a form that can be used even by developers who are not expert in networking.

You have already written some Java Stream Socket code which can handle only one connection at a time.

To manage multiple, concurrent clients, you need to allocate a new Thread for each new client Socket, as shown below.



Let’s consider the implications of such an approach. First, at any point many threads could be dormant, just waiting for input or output data to appear on the line.

This is likely to be a waste of resources. Second, each thread requires an allocation of stack memory whose default size ranges from 64 KB to 1 MB, depending on the OS.

Third, even if a Java virtual machine (JVM) can physically support a very large number of threads, the overhead of context-switching will begin to be troublesome long before that limit is reached, say by the time you reach 10,000 connections.

While this approach to concurrency might be acceptable for a small-to-moderate number of clients, the resources needed to support 100,000 or more simultaneous connections make it far from ideal. Fortunately, there is an alternative.

Java support for non-blocking I/O was introduced in 2002, with the JDK 1.4 package java.nio. The native

socket libraries have long included *non-blocking* calls, which provide considerably more control over the utilization of network resources.

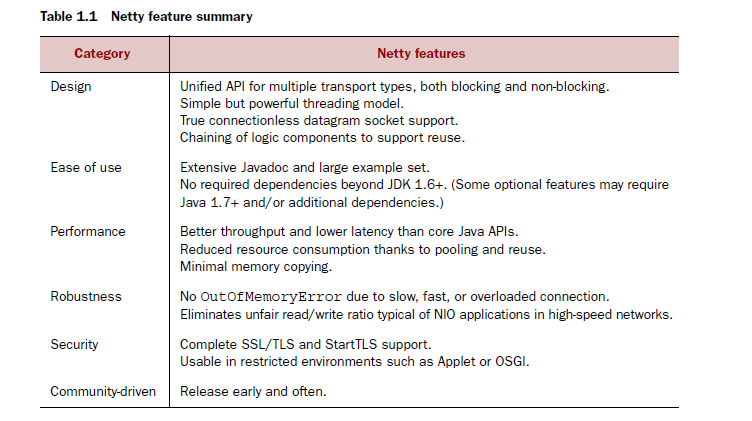
Although many applications have been built using the Java NIO API directly, doing so correctly and safely is far from trivial and is a cumbersome and error-prone task best left to a high-performance networking expert.

Not so long ago supporting thousands upon thousands of concurrent clients would have been judged impossible.

Today, as system users we take this capability for granted, and as developers we expect the bar to move even higher. We know there will always be demands for greater throughput and scalability—to be delivered at lower cost.

Don’t underestimate the importance of that last point. We’ve learned from long and painful experience that the direct use of low-level APIs exposes complexity and introduces a critical dependency on skills that tend to be in short supply. Hence, a fundamental concept of object orientation: hide the complexity of underlying implementations behind simpler abstractions.

In the networking domain, Netty is the preeminent framework for Java. Harnessing the power of Java’s advanced APIs behind an easy-to-use API, Netty leaves you free to focus on what really interests you—the unique value of your application.



Netty has a vibrant and growing user community that includes large companies such as Apple, Twitter, Facebook, Google, Square, and Instagram, as well as popular open source projects such as Infinispan, HornetQ, Vert.x, Apache Cassandra, and Elasticsearch, all of which have employed its powerful network abstractions in their core code.

Whenever you use Twitter, you are using Finagle, their Netty-based framework for inter-system communication. Facebook uses Netty in Nifty, their Apache Thrift service. Scalability and performance are critical concerns for both companies, and both are regular contributors to Netty.

In turn, Netty has benefited from these projects, enhancing both its scope and flexibility through implementations of protocols such as FTP, SMTP, HTTP, and WebSocket, as well as others, both binary and text-based.

***Asynchronous and event-driven***

Asynchronous, that is, *un-synchronized*, events are certainly familiar. Consider email: you may or may not get a response to a message you have sent, or you may receive an unexpected message even while sending one.

Asynchronous events can also have an *ordered* relationship. You generally get an answer to a question only *after* you have asked it, and you may be able to do something else while you are waiting for it.

Netty is asynchronous and event driven. This capability is critical for achieving the highest levels of *scalability*, defined as “the ability of a system, network, or process to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth.

Non-blocking network calls free us from having to wait for the completion of an operation. Fully asynchronous I/O builds on this feature and carries it a step further: an asynchronous method returns immediately and notifies the user when it is complete, directly or at a later time.

Putting these elements together, with non-blocking I/O we can handle very large numbers of events much more rapidly and economically than would be possible with blocking I/O. From the point of view of networking, this is key to the kind of systems we want to build, and as you’ll see, it is also key to Netty’s design from the ground up.

**Netty’s Building Blocks**

Netty’s primary building blocks:

■ Channels

■ Callbacks

■ Futures

■ Events and handlers

These building blocks represent different types of constructs: resources, logic, and notifications. Your applications will use them to access the network and the data that flows through it.

**Channels**

A Channel is a basic construct of Java NIO. It represents an open connection to an entity such as a hardware device, a file, a network socket, or a program component that is capable of performing one or more distinct I/O operations, for example reading or writing. Think of a Channel as a vehicle for incoming (inbound) and outgoing (outbound) data. As such, it can be open or closed, connected or disconnected.

**Callbacks**

A *callback* is simply a method, a reference to which has been provided to another method. This enables the latter to call the former at an appropriate time. Callbacks are used in a broad range of programming situations and represent one of the most common ways to notify an interested party that an operation has completed.

Netty uses callbacks internally when handling events; when a callback is triggered the event can be handled by an implementation of interface **ChannelHandler**.

**Futures**

A Future provides another way to notify an application when an operation has completed. This object acts as a placeholder for the result of an asynchronous operation; it will complete at some point in the future and provide access to the result.

**Events and handlers**

Netty uses distinct events to notify us about changes of state or the status of operations. This allows us to trigger the appropriate action based on the event that has occurred. Such actions might include

■ Logging

■ Data transformation

■ Flow-control

■ Application logic

Netty is a networking framework, so events are categorized by their relevance to inbound or outbound data flow. Events that may be triggered by inbound data or an associated change of state include

■ Active or inactive connections

■ Data reads

■ User events

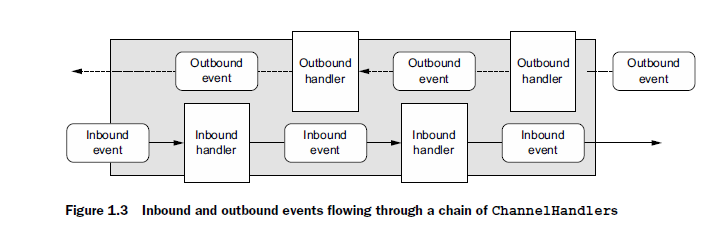
■ Error events

An outbound event is the result of an operation that will trigger an action in the future, which may be

■ Opening or closing a connection to a remote peer

■ Writing or flushing data to a socket

Every event can be dispatched to a user-implemented method of a handler class. This is a good example of an event-driven paradigm translating directly into application building blocks. Below shows how an event can be handled by a chain of such event handlers.



**Putting it all together**

Netty’s asynchronous programming model is built on the concepts of Futures and callbacks, with the dispatching of events to handler methods happening at a deeper level.

Taken together, these elements provide a processing environment that allows the logic of your application to evolve independently of any concerns with network operations. This is a key goal of Netty’s design approach.

Intercepting operations and transforming inbound or outbound data on the fly requires only that you provide callbacks or utilize the Futures that are returned by operations. This makes chaining operations easy and efficient and promotes the writing of reusable, generic code.

**Facebook and Twitter**

Twitter used Netty to build their own RPC framework on top of Netty (called Finagle). They used this to break up a monolith application into services in order to scale. Finagle and Netty handle every request Twitter sees. Finagle was born of the need to scale Twitter to billions of users across the whole globe. As of 2013 they can handle 143,00 tweets per second.

Facebook used Netty to build Nifty, a Java implementation of Thrift (their RPC framework)

**NIO using Netty**

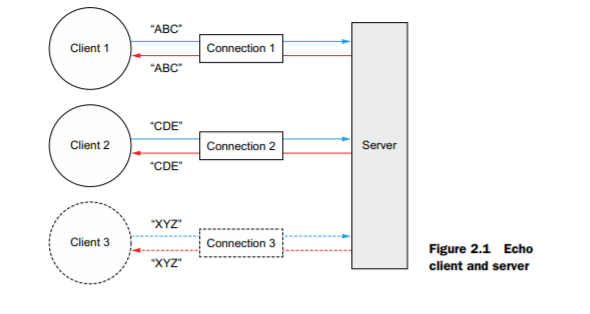
Netty is a framework for building client server systems using Java’s nio library

I am using the code from the book Netty in Action

In this lab we will use Netty to build a simple client server program which echoes text back to a client from a server.

Figure 2.1 presents a high-level view of the Echo client and server you’ll be writing.

The figure shows multiple clients connected simultaneously to the server. The number of clients that can be supported is limited, in theory, only by the system resources available (and any constraints that might be imposed by the JDK version in use). The interaction between an Echo client and the server is very simple; after the client establishes a connection, it sends one or more messages to the server, which in turn echoes each message to the client. While this may not seem terribly useful by itself, it exemplifies the request-response interaction that’s typical of client/server systems.



Make a folder called ***netty***

cd to ***netty*** and clone the book’s code as follows:

git clone <https://github.com/normanmaurer/netty-in-action.git>

Make sure Maven is installed on your machine - st the command line type

mvn –v

if its not install it from <https://maven.apache.org/download.cgi>

cd into ***netty-in-action***

type

mvn install

cd into chapter 2

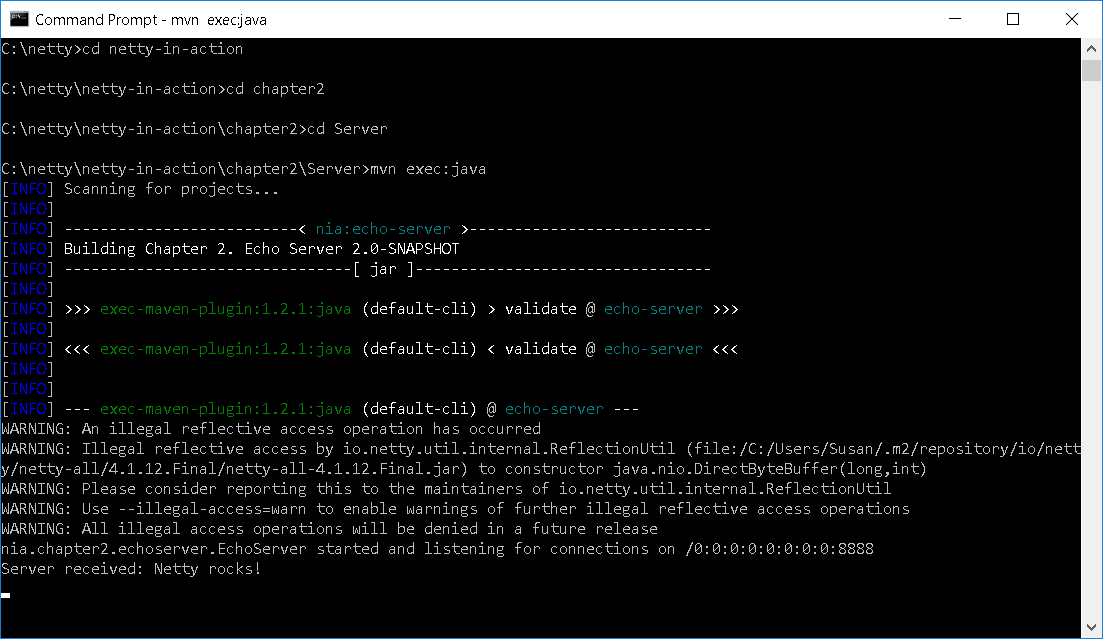
type

mvn clean package

now open two command windows and cd to ***chapter2*** in both windows

In one cd to ***Server*** and run

mvn exec:java



All Netty servers require the following:

At least one ChannelHandler—This component implements the server’s processing of data received from the client—its business logic.

Bootstrapping—This is the startup code that configures the server. At a minimum, it binds the server to the port on which it will listen for connection requests.

In the other cd to ***Client*** and run

mvn exec:java

You should see text echoed back from server to client

