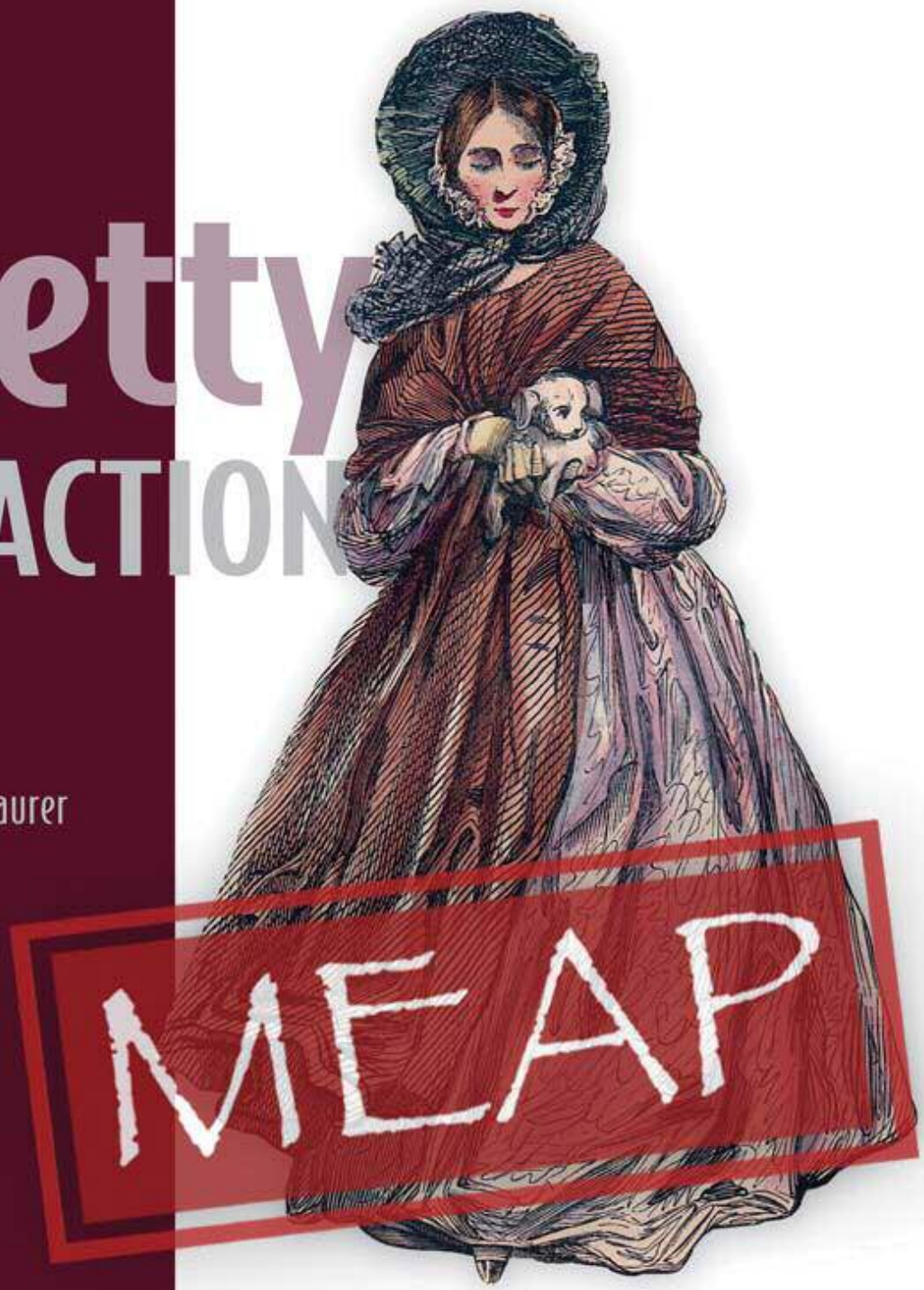
Netty
IN ACTION

Norman Maurer



MANNING

MEAP Edition
Manning Early Access Program
Netty in Action
Version 8

For more information on this and other Manning titles go to
www.manning.com

brief contents

1

Netty – Asynchronous and Event-Driven

This chapter covers

- What is Netty?
- Some History
- Introducing Netty
- Asynchrony and Events
- About this Book

WHAT IS NETTY?

Netty is a client/server framework that harnesses the power of Java's advanced networking while hiding its complexity behind an easy-to use API. Netty provides performance and scalability, leaving you free to focus on what really interests you - your unique application!

In this first chapter we'll explain how Netty provides value by giving some background on the problems of high-concurrency networking. Then we'll introduce the basic concepts and building blocks that make up the Netty toolkit and that we'll be studying in depth in the rest of the book.

SOME HISTORY

If you had started out in the early days of networking you would have spent a lot of time learning the intricacies of sockets, addressing and so forth, coding on the C socket libraries, and dealing with their quirks on different operating systems.

The first versions of Java (1995-2002) introduced just enough object-oriented sugar-coating to hide some of the intricacies, but implementing a complex client-server protocol still required a lot of boilerplate code (and a fair amount of peeking under the hood to get it right).

Those early Java APIs (`java.net`) supported only the so-called "blocking" functions provided by the native socket libraries. An unadorned example of server code using these calls is shown in Listing 1.1.

Blocking I/O Example

```
ServerSocket serverSocket = new ServerSocket(portNumber);       //1
Socket clientSocket = serverSocket.accept();                    //2
BufferedReader in = new BufferedReader(                         //3
new InputStreamReader(clientSocket.getInputStream()));
PrintWriter out =
new PrintWriter(clientSocket.getOutputStream(), true);
String request, response;
while ((request = in.readLine()) != null) {                     //4
if (“Done”.equals(request) (                                    //5
        break;
}
response = processRequest(request);                             //6
out.println(response);                                          //7
}                                                               //8
```

1. **A `ServerSocket` is created to listen for connection requests on a specified port.**
2. **The `accept()` call blocks until a connection is established. It returns a new `Socket` which will be used for the conversation between the client and the server.**
3. **Streams are created to handle the socket's input and output data. The `BufferedReader` reads text from a character-input stream. The `PrintWriter` prints formatted representations of objects to a text-output stream.**
4. **The processing loop begins. `readLine()` blocks until a string terminated by a line-feed or carriage return is read in.**
5. **If the client has sent "Done" the processing loop is exited.**
6. **The request is handled a processing method, which returns the server's response.**
7. **The response is sent to the client.**
8. **The processing loop continues.**

Obviously, this code is limited to handling only one connection at a time. In order to manage multiple, concurrent clients we need to allocate a new `Thread` for each new client `Socket` (and there is still plenty of code being written right now that does just that). But consider the implications of using this approach to support a very large number of simultaneous, long-lived connections. At any point in time many threads could be dormant, just waiting for input or output data to appear on the line. This could easily add up to a significant waste of resources, with a corresponding negative impact on performance. However, there is an alternative.

In addition to the blocking calls shown in the example, the native socket libraries have long included *nonblocking* I/O functions as well. These enable us to determine if there are data ready for reading or writing on any among a set of sockets. We can also set flags that will cause read/write calls to return immediately if there are no data; that is, if a blocking call would have blocked[1]. With this approach, at the cost of somewhat greater code complexity, we can obtain considerably more control over how networking resources are utilized.

JAVA NIO

In 2002, Java 1.4 introduced support for these nonblocking APIs in the package `java.nio` ("NIO").

"New" or "Nonblocking?"

"NIO" was originally an acronym for "New Input/Output." However, the Java API has been around long enough that it is no longer "new." The acronym is now commonly repurposed to signify "Non-blocking I/O." On the other hand, some (the author included) refer to blocking I/O as "OIO" or "Old Input/Output." You may also encounter "Plain I/O."

We've already shown an example of blocking I/O in Java. Figure 1.1 shows how that approach has to be expanded to handle multiple connections: by creating a dedicated thread for each

[1] "4.3BSD returned EWOULDBLOCK if an operation on a nonblocking descriptor could not complete without blocking." W. Richard Stevens, Advanced Programming in the UNIX Environment (1992), p. 364.

one - including connections that are idle! Clearly, the scalability of this method is going to be constrained by the number of threads you can create in the JVM.

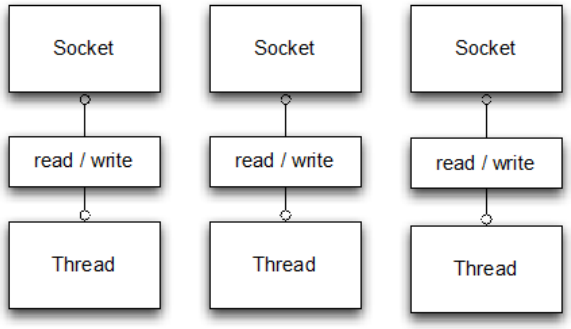


Figure 1.1 Blocking I/O

This may be acceptable if you know you will have a small number of connections to manage. But if you reach 10,000 or more concurrent connections the overhead of context-switching will begin to be noticeable. Furthermore, each thread has a default stack memory allocation between 128K and 1M. Considering the overall memory and operating system resources required to handle 100,000 or more concurrent connections, this appears to be a less than ideal solution.

SELECTORS

By contrast, figure 1.2 shows an approach using non-blocking I/O that mostly eliminates these constraints. Here we introduce the "`Selector`", which constitutes the linchpin of Java's non-blocking I/O implementation.

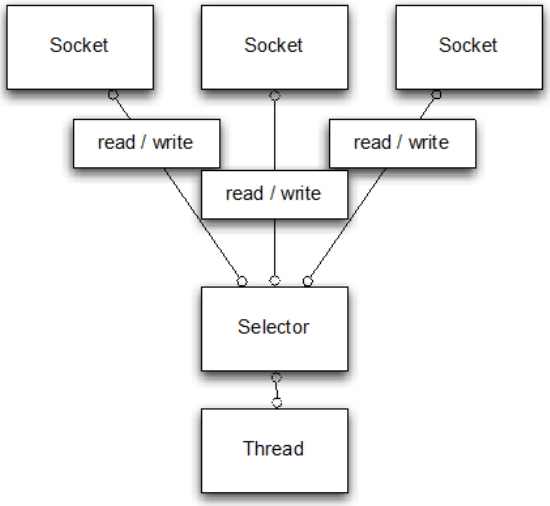


Figure 1.2 Nonblocking I/O

The `Selector` determines which of a set of registered sockets is ready for I/O at any point in time. As we explained earlier, this works because the I/O operations are set to non-blocking mode. Given this notification, a single associated thread can handle multiple concurrent connections. (A Selector is normally handled by one Thread but a specific implementation might employ multiple threads.) Thus, each time a read or write operation is performed it can be checked immediately for completion. Overall, this model provides much better resource usage than the blocking I/O model, since

- it can handle many connections with fewer threads, which means far less overhead due to memory and context-switching, and
- threads can be retargeted to other tasks when there is no I/O to handle.

You could build your applications directly on these Java NIO API constructs, but doing it correctly and safely is far from trivial. Implementing reliable and scalable event-processing to handle and dispatch data as efficiently as possible is a cumbersome and error-prone task best left to a specialist - Netty.

1.1 Introducing Netty

Not so long ago the idea that an application could support hundreds of thousands of concurrent clients would have been judged absurd. Today we take it for granted. Indeed, developers know that the bar keeps moving higher and that there will always be demands for greater throughput and availability - to be delivered at lower cost.

Let's not underestimate the importance of that last point. We have learned from long and painful experience that the direct use of low-level APIs not only exposes a high level of

complexity, but introduces a critical dependency on skill sets that are often in short supply. Hence a fundamental principle of Object Orientation: hide complexity behind abstractions.

This principle has borne fruit in the form of frameworks that encapsulate solutions to common programming tasks, many of them typical of distributed systems. Nowadays most professional Java developers are familiar with one or more of these frameworks[2] and for many they have become indispensable, enabling them to meet both their technical requirements as well as their schedules.

WHO USES NETTY?

Netty is one of the most widely-used Java networking frameworks[3]. Its vibrant and growing user community includes large companies like Facebook 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. In turn, Netty has benefited from interaction with these projects, enhancing both its scope and flexibility through implementations of protocols such as FTP, SMTP, HTTP, WebSocket and SPDY as well as others, both binary and text-based.

Firebase and Urban Airship are among the startups using Netty, the former Firebase for long-lived HTTP connections, the latter for all kinds of push notifications.

Whenever you use Twitter, you are using Finagle, their Netty-based API for inter-system communication. Facebook uses Netty to do something similar in Nifty, their Apache Thrift service. Both companies see scalability and performance as critical concerns and both are ongoing contributors to Netty.

These examples are presented as detailed case studies in Chapters 16 and 17, so if you are interested in real-world usage of Netty you might want to have a look there straight away.

In 2011 the Netty project became independent from Red Hat with the goal of facilitating the participation of contributors from the broader developer community. Both Red Hat and Twitter continue to use Netty and remain among its most active contributors.

The table below highlights many of the technical and methodological features of Netty that you will learn about and use in this book.

Category	Netty Features
Design	Unified API for multiple transport types—blocking and nonblocking
	Simple but powerful threading model
	True connectionless datagram socket support
	Chaining of logics to support reuse
Ease of Use	Extensive Javadoc and large example set

[2] Spring is probably the best known of these and is actually an entire ecosystem of application frameworks addressing dependency injection, batch processing, database programming, etc.

[3] Netty was awarded the Duke's Choice Award in 2011. See https://www.java.net/dukeschoice/2011

	No additional dependencies except JDK 1.6+. (Some features are supported only in Java 1.7+. Optional features may have additional dependencies.)
Performance	Better throughput, lower latency than core Java APIs
	Less resource consumption thanks to pooling and reuse
	Minimized memory copying
Robustness	Eliminates `OutOfMemoryError` due to slow, fast, or overloaded connection.
	Eliminates unfair read/write ratio often found in NIO applications in high-speed networks
Security	Complete SSL/TLS and StartTLS support
	Runs in a restricted environment such as Applet or OSGI
Community	Release early and often
	Community-Driven

ASYNCHRONOUS AND EVENT-DRIVEN

All network applications need to be engineered for scalability, which may be 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"[4]. As we have said, Netty helps you to accomplish this goal by exploiting nonblocking I/O, often referred to as "asynchronous I/O."

We'll be using the word "asynchronous" and its cognates a great deal in this book, so this is a good time to introduce them. Asynchronous, that is, *un*-synchronized *events* are certainly familiar to you from everyday life. For example, you send an E-mail message; you may or may not get a response later, or you may receive a message even while you are sending one. Asynchronous events can also have an *ordered* relationship. For example, you typically don't receive an answer to a question until you have asked it, but you aren't prevented from doing something else in the meantime.

In everyday life asynchrony "just happens," so we may not think about it very often. But getting computer programs to work the way we do does poses special problems, which go far beyond questions of network calls however sophisticated they may be. In essence, a system that is both asynchronous and "event-driven" exhibits a particular, and to us, valuable kind of behavior: *it can respond to events occurring at any time in any order*.

This is the kind of system we want to build, and as we shall see, this is the paradigm Netty supports from the ground up.

[4] Bondi, André B. (2000). "Characteristics of scalability and their impact on performance". *Proceedings of the second international workshop on Software and performance - WOSP '00*. p. 195.

1.2 Building Blocks

As we explained earlier, nonblocking I/O does not force us to wait for the completion of an operation. Building on this capability, true asynchronous I/O goes an important step further: *an asynchronous method returns immediately and notifies the user when it is complete, directly or at a later time.*

This approach may take a while to absorb if you are used to the most common execution model where a method returns only when it has completed. As we shall see, in a network environment the asynchronous model allows for more efficient utilization of resources, since multiple calls can be executed in rapid succession.

Let's start by looking at different, but related ways of utilizing completion notification. We'll cover them all in due course, and over time they will become core components of your Netty applications.

1.2.1 Channels

A `Channel` is a basic construct of 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[5].

For now, think of a Channel as "open" or "closed", "connected" or "disconnected" and as a vehicle for incoming and outgoing data.

1.2.2 Callbacks

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

Netty uses callbacks internally when handling events (see Section 1.2.3). Once such a callback is triggered the event can be handled by an implementation of `interface ChannelHandler`. Listing 1.2 shows such a `ChannelHandler`. Once a new connection has been established the callback `channelActive()` will be called and will print out a message.

ChannelHandler triggered by a callback

```
public class ConnectHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelActive(Channel channel) throws Exception {   //1
        System.out.println(
```

[5] http://docs.oracle.com/javase/7/docs/api/java/nio/channels/Channel.html.

```
            "Client " + channel.remoteAddress() + " connected");
    }
}
```

1. **`ChannelActive()` is called when a new connection is established**

1.2.3 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.

The JDK ships with `interface java.util.concurrent.Future` but the provided implementations allow you only to check manually if the operation has completed or to block until it does. This is quite cumbersome so Netty provides its own implementation, `ChannelFuture`, for use when an asynchronous operation is executed.

`ChannelFuture` provides additional methods that allow the registration of one or more `ChannelFutureListener` instances. The callback method, `operationComplete()`, is called when the operation has completed. The listener can then determine whether the operation completed successfully or with an error. If the latter, we can retrieve the `Throwable` that was produced. In short, the notification mechanism provided by the `ChannelFutureListener` eliminates the need for checking operation completion manually.

Each of Netty's outbound I/O operations returns a `ChannelFuture;` that is, none of them block at all. This gives you an idea of what we meant when we said that Netty is "asynchronous and event-driven from the ground up."

Listing 1.3 shows simply that a `ChannelFuture` is returned as part of an I/O operation. Here the `connect()` call will return directly without blocking and the call will complete in the background. When this will happen may depend on several factors but is abstracted away from this code. Because the thread is not blocked while awaiting completion of the operation, it is possible to do other work in the meantime, thus using resources more efficiently.

Callback in action

```
Channel channel = ...;
// Does not block
ChannelFuture future = channel.connect(                                    //1
    new InetSocketAddress("192.168.0.1", 25);
```

1. **Asynchronous connection to a remote peer**

Listing 1.4 shows how to utilize the `ChannelFutureListener` described above. First we connect to a remote peer. Then we register a new `ChannelFutureListener` with the `ChannelFuture` returned by the `connect()` call. When the listener is notified that the connection is established we check the status. If it was successful we write some data to the `Channel`. Otherwise we retrieve the `Throwable` from the `ChannelFuture`.

Note that error handling is entirely up to you subject, of course, to any constraints imposed by the specific error. For example, in case of a connection failure you could try to reconnect or establish a connection to another remote peer.

Callback in action

```
Channel channel = ...;
// Does not block
ChannelFuture future = channel.connect(                                    //1
    new InetSocketAddress("192.168.0.1", 25);
future.addListener(new ChannelFutureListener( {                          //2
    @Override
    public void operationComplete(ChannelFuture future) {
        if (future.isSuccess(){                                          //3
            ByteBuf buffer = Unpooled.copiedBuffer(
                "Hello",Charset.defaultCharset());                       //4
            ChannelFuture wf = future.channel().write(buffer);           //5
            ....
        } else {
            Throwable cause = future.cause();                            //6
            cause.printStacktrace();
        }
    }
});
```

1. **Connect asynchronously to a remote peer. The call returns immediately and provides a `ChannelFuture`.**
2. **Register a `ChannelFutureListener` to be notified once the operation completes.**
3. **When `operationComplete()` is called check the status of the operation.**
4. **If it is successful create a `ByteBuf` to hold the data.**
5. **Send the data asynchronously to the remote peer. This again returns a `ChannelFuture`.**
6. **If there was an error at 3. access the `Throwable` that describes the cause.**

If you are wondering whether a `ChannelFutureListener` isn't just a more elaborate version of a callback, you are correct. In fact, callbacks and `Future`s are complementary mechanisms; in combination they make up one of the key building blocks of Netty itself.

1.2.4 Events and Handlers

Netty uses different 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.

These actions might include

- logging
- data transformation
- flow-control
- application logic

Since Netty is a networking framework, it distinguishes clearly between events that are related to inbound or outbound data flow. Events that may be triggered because of some incoming data or change of state include:

- active or inactive connection
- data read
- user event
- error

Outbound events are the result operations that will trigger an action in the future. These include:

- opening or closing a connection to remote peer
- writing or flushing data to a socket

Every event can be dispatched to a user-implemented method of a handler class. This illustrates how an event-driven paradigm translates directly into application building blocks.

Figure 1.3 shows how an event can be handled by a chain of such event handlers.

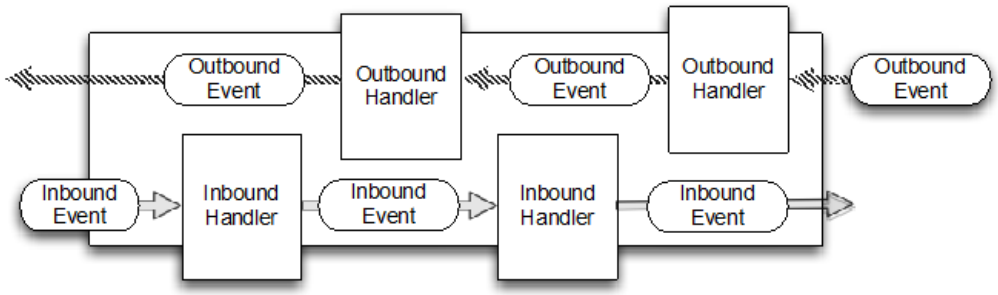


Figure 1.3 Event Flow

In Netty the `ChannelHandler` provides the basic abstraction for handlers like the ones shown here. We'll have a lot more to say about it in due course, but for now think of each handler instance as a kind of callback to be executed in response to a specific event.

Netty provides an extensive set of predefined handlers that you can use out of the box. Among these are codecs for various protocols including HTTP and SSL/TLS. Internally, `ChannelHandlers` use events and futures themselves, making consumers of the features Netty's abstractions.

1.2.5 Putting it All Together

FUTURES, CALLBACKS AND HANDLERS

As we explained above Netty's asynchronous programming model is built on the concepts of futures and callbacks. Below these is the event layer that is employed to trigger handler methods. The synergy of all these elements provides great power for your own designs.

Intercepting operations and transforming inbound or outbound data on the fly require 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. A

key goal of Netty's design is to promote "separation of concerns": the business logic of your application is decoupled from the network infrastructure.

SELECTORS, EVENTS AND EVENT LOOPS

Netty abstracts the `Selector` away from the application by firing events, eliminating all handwritten dispatch code that would otherwise be required. Under the covers an `EventLoop` is assigned to each Channel to handle all of the events, including

- registration of interesting events
- dispatching events to `ChannelHandlers`
- scheduling further actions.

The `EventLoop` itself is driven by exactly one thread, which handles all of the I/O events for one `Channel` and does not change during the lifetime of the `EventLoop`. This simple and powerful threading model eliminates any concern you might have about synchronization in your `ChannelHandlers`, so you can focus on providing the right callback logic to be executed when there are interesting data to process. The API is simple and compact.

1.3 About this Book

We started out by discussing the difference between blocking and non-blocking processing to give you a fundamental understanding of the advantages of the latter approach. We then moved on to an overview of Netty's features, design and benefits. These include the mechanisms underlying Netty's asynchronous model, including callbacks, futures and their use in combination. We also touched on Netty's threading model, how events are used and how they can be intercepted and handled. Going forward, we will explore in much greater depth how this rich collection of tools can be utilized to meet the very specific needs of your applications.

Along the way we will present case studies of companies whose engineers themselves explain why they chose Netty and how they use it.

So let's begin. In the next chapter, we'll delve into the basics of Netty's API and programming model, starting with writing an echo server and client.

2

Your First Netty Application

In this chapter we'll make certain you have a working development environment and test it out by building a simple client and server. Although we won't start studying the Netty framework in detail until the next chapter, here we will take a closer look at an important aspect of the API that we touched on in the introduction; namely, implementing application logic with `ChannelHandler`s.

By the end of the chapter you will have gained some hands-on experience with Netty and should feel comfortable working with the examples in the book.

2.1 Setting up the development environment

If you already have a working development environment with Maven then you might want to just skim this section.

If you only want to compile and run the book's examples the only tools you really need are the Java Development Kit (JDK) and Apache Maven, both freely available for download.

But we'll assume that you are going to want to tinker with the example code and pretty soon start writing some of your own. So although you *can* get by with just a plain text editor,

we recommend that if you aren't already using an Integrated Development Environment (IDE) for Java, you obtain and install one as described below.

1. **Obtain and install the Java Development Kit.**

   Your operating system may already have a JDK installed. To find out, type `"javac -version"` on the command line. If you get back something like `"javac 1.6..."` or `"javac 1.7..."` you're all set and can skip this step.

   Otherwise, get version 6 or later of the Java Development Kit (JDK) from java.com/en/download/manual.jsp (not the Java Runtime Environment (JRE), which can run Java applications but not compile them). There is a straightforward installer executable for each platform. Should you need installation instructions you'll find them on the same site.

   It's a good idea to set the environment variable `JAVA_HOME` to point to the location of your JDK installation. On Windows, the default will be something like `"C:\Program Files\Java\jdk1.7.0_55"`. It's also a good idea to add `"%JAVA_HOME%\bin"` (on Linux `"$JAVA_HOME/bin"`) to your execution path.

2. **Download and install an IDE.**

   These are the most widely used Java IDE's, all freely available. All of them have full support for our build tool, Apache Maven.

- Eclipse: www.eclipse.org
- NetBeans: www.netbeans.org
- Intellij Idea Community Edition: www.jetbrains.com

   NetBeans and Intellij are distributed as installer executables. Eclipse is usually distributed as a `zip` archive, although there are a number of customized versions that have self-installers.

3. **Download and install Apache Maven.**

   Maven is a widely used build management tool developed by the Apache Software Foundation. The Netty project uses it, as do this book's examples. You don't need to be a Maven expert just to build and run the examples, but if you want to expand on them we recommend reading the Maven Introduction in Appendix A.

Do you need to install Maven?

Both Eclipse and NetBeans come with an embedded Maven installation which will work fine for our purposes "out of the box." If you will be working in an environment that has its own Maven repository, your administrator probably has an installation package preconfigured to work with it.

At the time of this book's publication, the latest Maven version was 3.2.1. You can download the appropriate `tar.gz` or `zip` file for your system from http://ma-

ven.apache.org/download.cgi. Installation is simple: just extract the contents of the archive to any folder of your choice (We'll call this "<install_dir>".) This will create the directory <install_dir>/apache-maven-3.2.1.

After you have unpacked the Apache Maven archive, you may want to add <install-dir>/apache-maven-3.2.1/bin to your execution path so you can run Maven by executing "mvn" (or "mvn.bat") on the command line.

You should also set the environment variable M2_HOME to point to <install_dir>/apache-maven-3.2.1.

4. **Configure the Toolset**

   If you have set the JAVA_HOME and M2_HOME system variables as suggested, you may find that when you start your IDE it has already discovered the locations of your Java and Maven installations. If you need to perform manual configuration, all the IDE versions we have mentioned have menu items for setting up Maven under "Preferences" or "Settings". Please consult the documentation of the IDE for details.

This should complete the setup of your development environment. Next we'll explain how Maven is used to build the examples, then we'll try it out.

2.2 Netty client / server overview

In this section we'll build a complete Netty client and server. Although you may be focused on writing Web-based services where the client is a browser, you'll gain a more complete understanding of the Netty API by implementing both the client and server.

Figure 2.1 presents a high-level view of the echo client and server.

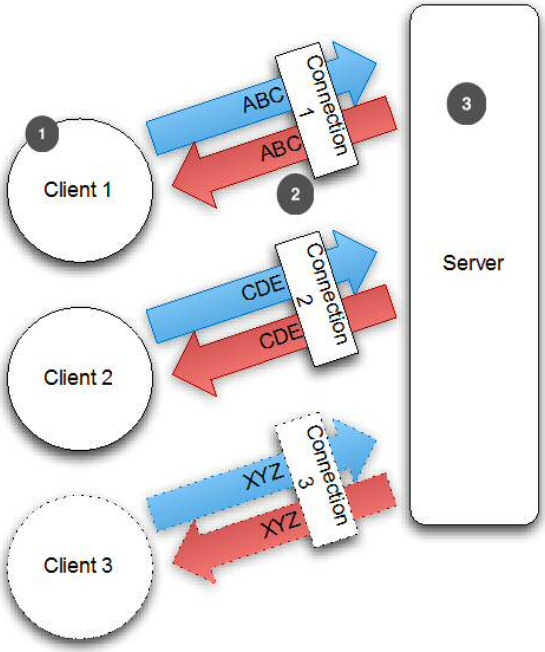


Figure 2.1.Echo client / server

The figure shows multiple concurrent clients connected to the server. In theory, the number of clients that can be supported is limited only by the system resources available and any constraints that might be imposed by the JDK version used.

The interaction between the echo client and server is very simple; after the client establishes a connection it sends one or more messages to the server, which echoes each message back to the client. Admittedly, this application is not terribly useful. But the point of this exercise is to understand the request-response interaction itself, which is a basic pattern of client / server systems.

We'll start by examining the server-side code.

2.3 Writing the echo server

All Netty servers require the following:

- A server handler. This component implements the server's "business logic", which determines what happens when a connection is made and information is received from the client.
- Bootstrapping. This is the startup code that configures the server. At a minimum it sets the port to which the server will "bind"; that is, on which it will listen for connection requests.

In the next sections we'll examine the logic and bootstrap code for the Echo Server.

2.3.1 Implementing the server logic with ChannelHandlers

In the first chapter we introduced futures and callbacks and illustrated their use in an event-driven design model. We also discussed `interface ChannelHandler`, whose implementations receive event notifications and react accordingly. This core abstraction represents the container for our business logic.

The Echo Server will react to an incoming message by returning a copy to the sender. Therefore, we will need to provide an implementation of interface `ChannelInboundHandler`, which defines methods for acting on inbound events. Our simple application will require only a few of these methods, so subclassing the concrete class `ChannelInboundHandlerAdapter` class should work well. This class provides a default implementation for `ChannelInboundHandler`, so we need only override the methods that interest us, namely

- `channelRead()` - called for each incoming message
- `channelReadComplete()` - called to notify the handler that the last call made to `channelRead()` was the last message in the current batch
- `exceptionCaught()` - called if an exception is thrown during the read operation

The class we provide is `EchoServerHandler`, as shown in Listing 2.2.

Listing 2.2 EchoServerHandler

```
@Sharable                                                          //1
public class EchoServerHandler extends ChannelInboundHandlerAdapter {

      @Override
      public void channelRead(ChannelHandlerContext ctx, Object msg) {
        ByteBuf in = (ByteBuf) msg;
        System.out.println(
                "Server received: " + in.toString(CharsetUtil.UTF_8));  //2
        ctx.write(in);                                                  //3
      }

      @Override
      public void channelReadComplete(ChannelHandlerContext ctx) {
        ctx.writeAndFlush(Unpooled.EMPTY_BUFFER)                       //4
                .addListener(ChannelFutureListener.CLOSE);
      }

      @Override
      public void exceptionCaught(ChannelHandlerContext ctx,
        Throwable cause) {
        cause.printStacktrace();                                       //5
        ctx.close();                                                   //6
      }
}
```

1. **The annotation `@Sharable` marks this class as one whose instances can be shared among channels.**
2. **Log the message to the console.**
3. **Writes the received message back to the sender. Note that this does not yet "flush" the outbound messages.**

4. **Flushes all pending messages to the remote peer. Closes the channel after the operation is complete.**
5. **Prints the exception stack trace.**
6. **Closes the channel.**

This way of using `ChannelHandlers` promotes the design principle of separation of concerns and simplifies the iterative development of business logic as requirements evolve. The handler is straightforward; each of its methods can be overridden to ”hook” into the event lifecycle at the appropriate point. Obviously, we override `channelRead` because we need to handle all received data; in this case we echo it back to the remote peer.

Overriding `exceptionCaught` allows us to react to any `Throwable` subtypes. In this case we log it and close the connection which may be in an unknown state. It is usually difficult to recover from connection errors, so simply closing the connection signals to the remote peer that an error has occurred. Of course, there may be scenarios where recovering from an error condition is possible, so a more sophisticated implementation could try to identify and handle such cases.

What happens if an Exception is not caught?

Every `Channel` has an associated `ChannelPipeline`, which represents a chain of `ChannelHandler` instances. Adapter handler implementations just forward the invocation of a handler method on to the next handler in the chain. Therefore, if a Netty application does not override `exceptionCaught` somewhere along the way, those errors will travel to the end of the `ChannelPipeline` and a warning will be logged. For this reason you should supply at least one `ChannelHandler` that implements `exceptionCaught`.

In addition to `ChannelInboundHandlerAdapter` there are numerous `ChannelHandler` subtypes and implementations to learn about if you plan to implement real-world applications or write a framework that uses Netty internally. These are covered in detail in chapters 6 and 7. For now, these are the key points to keep in mind:

- `ChannelHandlers` are invoked for different types of events.
- Applications implement or extend `ChannelHandlers` to hook into the event lifecycle and provide custom application logic.

2.3.2 Bootstrapping the server

Having discussed the core business logic implemented by the `EchoServerHandler`, all that remains is to examine the bootstrapping of the server itself.

This will involve

- listening for and accepting incoming connection requests
- configuring `Channels` to notify an `EchoServerHandler` instance about inbound messages

Transports

In this section you'll encounter the term "transport". In the multi-layered view of networking protocols, the transport layer provides services for end-to-end or host-to-host communications.

The basis of internet communications is the TCP transport. When we use the term "NIO transport" we are referring to a transport implementation which is mostly identical to TCP except for some server-side performance enhancements that are provided by the Java NIO implementation.

Transports will be discussed in detail in Chapter 4.

Listing 2.3 is the complete code for the `EchoServer` class.

Listing 2.3 EchoServer

```
public class EchoServer {
       private final int port;

       public EchoServer(int port) {
         this.port = port;
       }
       public static void main(String[] args) throws Exception {
         if (args.length != 1) {
                 System.err.println(
                         "Usage: " + EchoServer.class.getSimpleName() +
                   " <port>");
          }
          int port = Integer.parseInt(args[0]);                                  //1
          new EchoServer(port).start();                                          //2
       }

       public void start() throws Exception {
         EventLoopGroup group = new NioEventLoopGroup();                         //3
          try {
                 ServerBootstrap b = new ServerBootstrap();
                 b.group(group)                                                  //4
                         .channel(NioServerSocketChannel.class)                  //5
                         .localAddress(new InetSocketAddress(port))              //6
                         .childHandler(new ChannelInitializer<SocketChannel>(){  //7
                         @Override
                         public void initChannel(SocketChannel ch)
                                 throws Exception {
                                         ch.pipeline().addLast(new
       EchoServerHandler());
                                 }
                         });
                 ChannelFuture f = b.bind().sync();                              //8
                 f.channel().closeFuture().sync();                               //9
          } finally {
                 group.shutdownGracefully().sync();                              //10
         }
       }
}
```

1. **Set the port value (throws a `NumberFormatException` if the port argument is malformed)**
2. **Call the server's `start()` method.**

3. **Create the `EventLoopGroup`**
4. **Create the `ServerBootstrap`**
5. **Specify the use of an NIO transport `Channel`**
6. **Set the socket address using the selected port**
7. **Add an `EchoServerHandler` to the `Channel`'s `ChannelPipeline`**
8. **Bind the server; `sync` waits for the server to close**
9. **Close the channel and block until it is closed**
10. **Shutdown the `EventLoopGroup`, which releases all resources.**

In this example, the code creates a `ServerBootstrap` instance (step **4**). Since we are using the NIO transport, we have specified the `NioEventLoopGroup(`**3**`)` to accept and handle new connections and the `NioServerSocketChannel (`**5**`)` as the channel type. After this we set the local address to be an `InetSocketAddress` with the selected port (**6**). The server will bind to this address to listen for new connection requests.

Step **7** is key: here we make use of a special class, `ChannelInitializer`. When a new connection is accepted, a new child `Channel` will be created and the `ChannelInitializer` will add an instance of our `EchoServerHandler` to the `Channel`'s `ChannelPipeline`. As we explained earlier, this handler will then be notified about inbound messages.

While NIO is scalable, its proper configuration is not trivial. In particular, getting multithreading right isn't always easy. Fortunately, Netty's design encapsulates most of the complexity, especially via abstractions such as `EventLoopGroup`, `SocketChannel` and `ChannelInitializer`, each of which will be discussed in more detail in chapter 3.

At step **8**, we bind the server and wait until the bind completes. (The call to `sync()` causes the current `Thread` to block until then.) At step **9** the application will wait until the server's `Channel` closes (because we call `sync()` on the `Channel`'s `CloseFuture`). We can now shut down the `EventLoopGroup` and release all resources, including all created threads (**10**).

NIO is used in this example because it is currently the most widely-used transport, thanks to its scalability and thoroughgoing asynchrony. But a different transport implementation is also possible. For example, if this example used the OIO transport, we would specify `OioServerSocketChannel` and OioEventLoopGroup. Netty's architecture, including more information about transports, will be covered in Chapter 4. In the meantime, let's review the important steps in the server implementation we just studied.

The primary code components of the server are

- the `EchoServerHandler` that implements the business logic
- the `main()` method that bootstraps the server

The steps required to perform the latter are:

- Create a `ServerBootstrap` instance to bootstrap the server and bind it later.
- Create and assign an `NioEventLoopGroup` instance to handle event processing, such as accepting new connections and reading / writing data.
- Specify the local `InetSocketAddress` to which the server binds.
- Initialize each new `Channel` with an `EchoServerHandler` instance.
- Finally, call `ServerBootstrap.bind()` to bind the server.

At this point the server is initialized and ready to be used.

In the next section we'll examine the code for the client side of the system, the "Echo Client".

2.4 Writing an echo client

Now that all of the code is in place for the server, let's create a client to use it.

The client will

- connect to the server
- send one or more messages
- for each message, wait for and receive the same message back from the server
- close the connection

Writing the client involves the same two main code areas we saw in the server: business logic and bootstrapping.

2.4.1 Implementing the client logic with ChannelHandlers

Just as we did when we wrote the server, we'll provide a `ChannelInboundHandler` to process the data. In this case, we will extend the class `SimpleChannelInboundHandler` to handle all the needed tasks, as shown in listing 2.4. We do this by overriding three methods that handle events that are of interest to us:

- `channelActive()` - called after the connection to the server is established
- `channelRead0()` - called after data is received from the server
- `exceptionCaught()` - called if an exception was raised during processing

Listing 2.4 ChannelHandler for the client

```
@Sharable                                                          //1
public class EchoClientHandler extends
      SimpleChannelInboundHandler<ByteBuf> {
      @Override
      public void channelActive(ChannelHandlerContext ctx) {
        ctx.write(Unpooled.copiedBuffer("Netty rocks!",                //2
        CharsetUtil.UTF_8);                                                }

      @Override
      public void channelRead0(ChannelHandlerContext ctx, ByteBuf in) {
        System.out.println(
                "Client received: " + in.toString(CharsetUtil.UTF_8));
      }

      @Override
      public void exceptionCaught(ChannelHandlerContext ctx,             //4
        Throwable cause) {
        cause.printStracktrace();
        ctx.close();
      }
}
```

1. **The annotation `@Sharable` marks this class as one whose instances can be shared among channels.**
2. **When notified that the channel is active send a message.**
3. **Log a hexadecimal dump of the received message.**
4. **On exception log the error and close channel.**

The `channelActive()` method is invoked once we establish a connection. The logic is simple: once the connection is established, a sequence of bytes is sent to the server. The contents of the message doesn't matter; Here we used the encoded string of "Netty rocks!" By overriding this method we ensure that something is written to the server as soon as possible.

Next we override the method `channelRead0()`. This method is called whenever data are received. Note that the message sent by the server may be received in chunks. That is, when the server sends 5 bytes it's not guaranteed that all 5 bytes will be received at once - even for just 5 bytes, the `channelRead0()` method could be called twice, the first time with a `ByteBuf` (Netty's `byte` container) holding 3 bytes and the second time with a `ByteBuf` holding 2 bytes. The only thing guaranteed is that the bytes will be received in the order in which they were sent. (Note that this is true only for stream-oriented protocols such as TCP.)

The third method to override is `exceptionCaught()`. Just as in the `EchoServerHandler` (Listing 2.2), the `Throwable` is logged and the channel is closed, in this case terminating the connection to the server.

SimpleChannelInboundHandler vs. ChannelInboundHandler

You may be wondering why we used `SimpleChannelInboundHandler` in the client instead of the `ChannelInboundHandlerAdapter` we used in the `EchoServerHandler`. This has to do with the interaction of two factors: how our business logic processes messages and how Netty manages resources.

In the client, when `channelRead0()` completes, we have the incoming message and we are done with it. When this method returns, `SimpleChannelInboundHandler` takes care of releasing the reference to the `ByteBuf` that holds the message.

In `EchoServerHandler`, on the other hand, we still have to echo the incoming message back to the sender, and the `write()` operation, which is asynchronous, may not complete until after `channelRead()` returns (see item **2** in Listing 2.2). For this reason we use `ChannelInboundHandlerAdapter`, which does *not* release the message at this point. Finally, the message is released in `channelReadComplete()` when we call `ctxWriteAndFlush()` (item **3**).

Chapters 5 and 6 will cover message resource management in more detail.

2.4.2 Bootstrapping the client

As you can see in Listing 2.5, bootstrapping a client is similar to bootstrapping a server. Of course, the client needs both host and port parameters for the server connection.

Listing 2.5 Main class for the client

```
public class EchoClient {
      private final String host;
      private final int port;

      public EchoClient(String host, int port) {
        this.host = host;
        this.port = port;
      }

      public void start() throws Exception {
        EventLoopGroup group = new NioEventLoopGroup();
        try {
                Bootstrap b = new Bootstrap();                          //1
                b.group(group)                                                  //2
                        .channel(NioSocketChannel.class)                        //3
                        .remoteAddress(new InetSocketAddress(host, port))       //4
                        .handler(new ChannelInitializer<SocketChannel>() {      //5
                                @Override
                                public void initChannel(SocketChannel ch)
                                        throws Exception {
                                        ch.pipeline().addLast(
                                                        new EchoClientHandler());
                                }
                        });
                ChannelFuture f = b.connect().sync();                   //6
                f.channel().closeFuture().sync();                               //7
        } finally {
                group.shutdownGracefully().sync();                              //8
        }
      }

      public static void main(String[] args) throws Exception {
        if (args.length != 2) {
                System.err.println(
                        "Usage: " + EchoClient.class.getSimpleName() +
                        " <host> <port>");
                return;
        }

        // Parse options.
        final String host = args[0];
        final int port = Integer.parseInt(args[1]);
        new EchoClient(host, port).start();
      }
}
```

1. **Create `Bootstrap`.**
2. **Specify `EventLoopGroup` to handle client events. The `NioEventLoopGroup` implementation is used, since we are using NIO Transport.**
3. **The channel type used is the one for NIO-Transport.**
4. **Set the server's `InetSocketAddress`.**
5. **When a connection is established and a new channel is created add an `EchoClientHandler` instance to the channel pipeline.**
6. **Connect to the remote peer; wait until the connect completes.**
7. **Block until the `Channel` closes.**
8. **`shutdownGracefully()` invokes the shutdown of the thread pools and the release of all resources.**

As before, the NIO transport is used here. Note that you can use different transports in the client and server, for example the NIO transport on the server side and the OIO transport on the client side. In chapter 4 we'll examine the specific factors and scenarios that would lead you to select one transport rather than another.

Let's review the important points we introduced in this section:

- A `Bootstrap` instance is created to initialize the client.
- An `NioEventLoopGroup` instance is assigned to handle the event processing, which includes creating new connections and processing inbound and outbound data.
- An `InetSocketAddress` is created for the connection to the server.
- An `EchoClientHandler` will be installed in the pipeline when the connection is established.
- After everything is set up, `Bootstrap.connect()` is called to connect to the remote peer - in this case, an Echo Server.

Having finished the client it's time to build the system and test it out.

2.5 Building and running the Echo Server and Client

In this section we'll cover all the steps needed to compile and run the Echo Server and Client.

2.5.1 Building the example

The Echo Client / Server Maven project

Appendix A uses the configuration of the Echo Client / Server project to explain in detail how multi-module Maven projects are organized. This is not required reading for building and running the Echo project, but highly recommended for deeper understanding of the book examples and of Netty itself.

To build the Client and Server artifacts, go to the `chapter2` directory under the code samples root directory and execute

```
mvn clean package
```

This should produce something very much like the output shown in Listing 2.6 (we have edited out a few non-essential build step reports).

Listing 2.6 Build Output

```
chapter2>mvn clean package
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Build Order:
[INFO]
[INFO] Echo Client and Server
[INFO] Echo Client
[INFO] Echo Server
```

```
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Echo Client and Server 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ echo-parent ---
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Echo Client 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ echo-client ---
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile)
      @ echo-client ---
[INFO] Changes detected - recompiling the module!
 [INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] Echo Client and Server ......................... SUCCESS [  0.118 s]
[INFO] Echo Client .................................... SUCCESS [  1.219 s]
[INFO] Echo Server .................................... SUCCESS [  0.110 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.561 s
[INFO] Finished at: 2014-06-08T17:39:15-05:00
[INFO] Final Memory: 14M/245M
[INFO] ------------------------------------------------------------------------
```

Notes:

- The Maven Reactor lays out the build order: the parent pom, then the subprojects.
- The Netty artifacts are not found in the user's local repository, so Maven downloads them from the network repository, here acting as a mirror to the public Maven repositories.
- The `clean` and `compile` phases of the build lifecycle are run. Afterwards the `maven-surefire-plugin` is run but no test classes are found in this project. Finally the `maven-jar-plugin` is executed.

The Reactor Summary shows that all projects have been successfully built. A listing of the target directories in the two subprojects should now resemble Listing 2.7.

Listing 2.7 Build Artifacts

```
Directory of netty-in-action\chapter2\Client\target

06/08/2014  05:39 PM    <DIR>          .
06/08/2014  05:39 PM    <DIR>          ..
06/08/2014  05:39 PM    <DIR>          classes
06/08/2014  05:39 PM             5,619 echo-client-1.0-SNAPSHOT.jar
06/08/2014  05:39 PM    <DIR>          generated-sources
06/08/2014  05:39 PM    <DIR>          maven-archiver
06/08/2014  05:39 PM    <DIR>          maven-status

Directory of netty-in-action/chapter2/Server/target
```

```
06/08/2014  05:39 PM    <DIR>          .
06/08/2014  05:39 PM    <DIR>          ..
06/08/2014  05:39 PM    <DIR>          classes
06/08/2014  05:39 PM             5,511 echo-server-1.0-SNAPSHOT.jar
06/08/2014  05:39 PM    <DIR>          generated-sources
06/08/2014  05:39 PM    <DIR>          maven-archiver
06/08/2014  05:39 PM    <DIR>          maven-status
```

2.5.2 Running the Echo Server and Client

To run the application components we could certainly use the Java command directly. But in our POM files we have configured the `exec-maven-plugin` to do this for us (see Appendix A for details).

Open two console windows side by side, one logged into the chapter2/Server directory, the other in chapter2/Client.

In the server's console, execute

```
mvn exec:java
```

You should see something like the following:

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Echo Server 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) @ echo-server >>>
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) @ echo-server <<<
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ echo-server ---
nettyinaction.echo.EchoServer started and listening for connections on
      /0:0:0:0:0:0:0:0:9999
```

The server is started and ready to accept connections. Now execute the same command in the client's console. You should see the following:

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Echo Client 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) @ echo-client >>>
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) @ echo-client <<<
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ echo-client ---
Client received: Netty rocks!
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3.907 s
[INFO] Finished at: 2014-06-08T18:26:14-05:00
```

```
[INFO] Final Memory: 8M/245M
[INFO] ------------------------------------------------------------------------
```

And in the server console:

```
Server received: Netty rocks!
```

What happened:

- After the client is connected, it sends its message: "Netty rocks!"
- The server reports the received message and echoes it back to the client.
- The client reports the returned message and exits.

Every time you run the client, you'll see one log statement in the server's console:

Server received: Netty rocks!

This works as expected. But now let's see how failures are handled. The server should still be running, so type Ctrl-C in the server console to stop the process. Once it has terminated, start the client again with

```
mvn exec:java
```

Listing 2.10 shows the output you should see from the client when it is unable to connect to the server.

Listing 2.8 Exception Handling in Client

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Echo Client 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) @ echo-client >>>
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) @ echo-client <<<
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ echo-client ---
[WARNING]
java.lang.reflect.InvocationTargetException
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke
                (NativeMethodAccessorImpl.java:57)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke
                (DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:606)
        at org.codehaus.mojo.exec.ExecJavaMojo$1.run(ExecJavaMojo.java:297)
        at java.lang.Thread.run(Thread.java:744)
Caused by: java.net.ConnectException: Connection refused:
          no further information: localhost/127.0.0.1:9999
        at sun.nio.ch.SocketChannelImpl.checkConnect(Native Method)
        at sun.nio.ch.SocketChannelImpl.finishConnect
                (SocketChannelImpl.java:739)
        at io.netty.channel.socket.nio.NioSocketChannel
                .doFinishConnect(NioSocketChannel.java:191)
        at io.netty.channel.nio.
```

```
                AbstractNioChannel$AbstractNioUnsafe.finishConnect(
                AbstractNioChannel.java:279)
        at io.netty.channel.nio.NioEventLoop
                .processSelectedKey(NioEventLoop.java:511)
        at io.netty.channel.nio.NioEventLoop
                .processSelectedKeysOptimized(NioEventLoop.java:461)
        at io.netty.channel.nio.NioEventLoop
                .processSelectedKeys(NioEventLoop.java:378)
        at io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:350)
        at io.netty.util.concurrent
                .SingleThreadEventExecutor$2.run
                (SingleThreadEventExecutor.java:101)
        ... 1 more
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3.728 s
[INFO] Finished at: 2014-06-08T18:49:13-05:00
[INFO] Final Memory: 8M/245M
[INFO] ------------------------------------------------------------------------
[ERROR] Failed to execute goal org.codehaus.mojo:exec-maven-plugin:1.2.1:java
      (default-cli) on project echo-client: An exception occured while executing the
      Java class. null: InvocationTargetException: Connection refused: no further
      information:
      localhost/127.0.0.1:9999 -> [Help 1]
```

What happened? The client tried to connect to the server, which it expected to find running on `localhost:9999`. This failed (as expected) because the server had been stopped previously, causing a `java.net.ConnectException` in the client. This exception triggered the `exceptionCaught()` method of the `EchoClientHandler`, which prints out the stack trace and closes the channel (please see Listing 2.4.)

2.6 Summary

In this chapter you built and ran your first Netty client and server. While this is a simple application, it will scale to several thousand concurrent connections - many more messages per second than a "plain vanilla" socket-based Java application would be able to handle.

In the following chapters, we'll see many more examples of how Netty simplifies scalability and multithreading. We'll also go deeper into Netty's support for the architectural concept of separation of concerns; by providing the right abstractions for decoupling business logic from networking logic, Netty makes it easy to keep pace with rapidly evolving requirements without jeopardizing system stability.

In the next chapter we will provide an overview of Netty's architecture. This will provide the context for the in-depth and comprehensive study of Netty's internals that will follow in subsequent chapters.

3

Netty Overview

In this chapter we'll focus on Netty's architectural model. We'll study the functionality of its primary components taken singly and in collaboration. These include

- `Bootstrap and ServerBootstrap`
- `Channel`
- `ChannelHandler`
- `ChannelPipeline`
- `EventLoop`
- `ChannelFuture`

This goal is to provide a context for the in-depth study we will be undertaking in subsequent chapters. It's much easier to understand a framework if you have a good grasp on its organizing principles; this helps you to avoid losing your way when you get into the details of its implementation.

3.1 Netty Crash Course

We'll start by enumerating the basic building blocks of all Netty applications, both clients and servers.

BOOTSTRAP

A Netty application begins by setting up one of the bootstrap classes, which provide a container for the configuration of the application's network layer.

CHANNEL

To be somewhat formal about it, the underlying network transport API must provide applications with a construct that implements I/O operations: read, write, connect, bind and so forth. For us, this construct is pretty much always going to be a "socket". Netty's interface Channel defines the semantics for interacting with sockets by way of a rich set of operations: bind, close, config, connect, isActive, isOpen, isWritable, read, write and others. Netty provides numerous Channel implementations for specialized use. These include AbstractChannel, AbstractNioByteChannel, AbstractNioChannel, EmbeddedChannel, LocalServerChannel, NioSocketChannel and many more.

CHANNELHANDLER

`ChannelHandlers support` a variety of protocols and provide containers for data-processing logic. We have already seen that a `ChannelHandler` is triggered by a specific event or set of events. Note that the use of the generic term "event" is intentional, since a `ChannelHandler` can be dedicated to almost any kind of action - converting an object to bytes (or the reverse), or handling exceptions thrown during processing.

One interface you'll be encountering (and implementing) frequently is `ChannelInboundHandler`. This type receives inbound events (including received data) that will be handled by your application's business logic. You can also flush data from a `ChannelInboundHandler` when you have to provide a response. In short, the business logic of your application typically lives in one or more `ChannelInboundHandlers`.

CHANNELPIPELINE

A `ChannelPipeline` provides a container for a chain of `ChannelHandler`s and presents an API for managing the flow of inbound and outbound events along the chain. Each `Channel` has its own `ChannelPipeline`, created automatically when the `Channel` is created.

How do `ChannelHandler`s get installed in the `ChannelPipeline`? This is the role of `abstract ChannelInitializer, which implements ChannelHandler.` A subclass of `ChannelInitializer is registered with a ServerBootstrap. When its method initChannel()` is called, this object will install a custom set of `ChannelHandler`s in the pipeline. When this operation is completed, the `ChannelInitializer subclass` then automatically removes itself from the `ChannelPipeline`.

EVENTLOOP

An `EventLoop` processes I/O operations for a `Channel`. A single `EventLoop` will typically handle events for multiple `Channels`. An `EventLoopGroup` may contain more than one `EventLoop` and provides an iterator for retrieving the next one in the list.

CHANNELFUTURE

As we have already explained, all I/O operations in Netty are asynchronous. Since an operation may not return immediately we need to have a way to determine its result at a later time. For this purpose Netty provides `interface ChannelFuture`, whose `addListener` method registers a `ChannelFutureListener` to be notified when an operation has completed (whether successfully or not).

More on `ChannelFuture`

Think of a `ChannelFuture` as a placeholder for the result of an operation that is to be executed in the future. *When* it will executed may depend on several factors and thus impossible to predict with precision. But we can be certain that it *will* be executed. Furthermore, all operations that return a `ChannelFuture` and belong to the same `Channel` will be executed in the correct order - that in which they are invoked.

The rest of this chapter is devoted to exploring each of these core components and its functionality in more detail.

3.2 Channels, Events and I/O

We have stated that "Netty is a non-blocking, event-driven networking framework." More simply, "Netty uses `Threads` to process I/O events." If you are familiar with the requirements of multi-threaded programming, you may be concerned about whether you will need to synchronize your code. You won't, as long as you don't share `ChannelHandler` instances among `Channels`, and Figure 3.1 shows why.

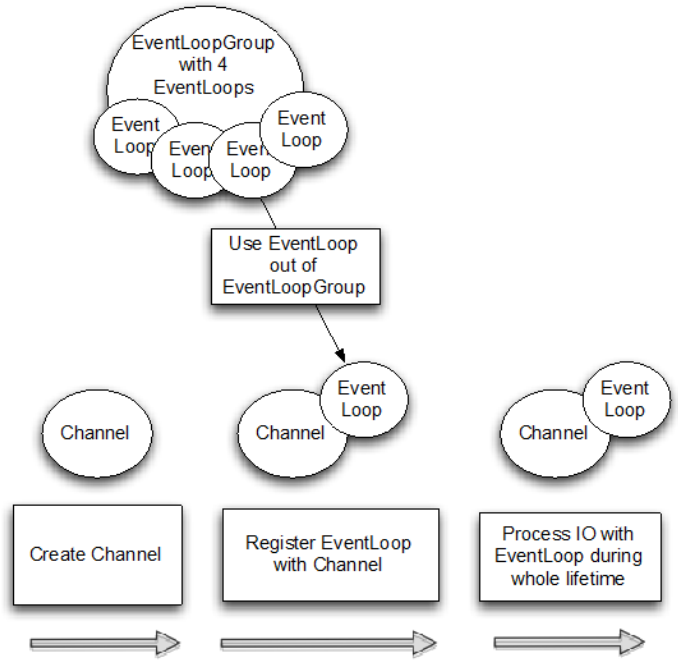


Figure 3.1

The figure shows that an `EventLoopGroup` has one or more `EventLoops`. Think of an `EventLoop` as a `Thread` that performs the actual work for a `Channel`. (In fact, an `EventLoop` is bound to a single `Thread` for its lifetime.)

When a `Channel` is created, Netty registers that `Channel` with a single `EventLoop` instance (and so to a single `Thread`) for the lifetime of the `Channel`. This is why your application doesn't need to synchronize on Netty I/O operations; all the I/O for a given `Channel` will always be performed by the same `Thread`.

We'll discuss `EventLoop` and `EventLoopGroup` further in Chapter 15.

3.3 The What and Why of Bootstrapping

Bootstrapping is the process of configuring your application for its networking function. That is, you perform bootstrapping to bind a process to a given port or to connect one process to another at a specified host and port.

In general, we refer to the former use case as a "server" and the latter as a "client". While this is simple and convenient terminology, it obscures a fundamental difference that is of interest to us; namely, that a "server" listens for incoming connections while a "client" establishes connections with one or more processes.

Accordingly, there are two types of bootstraps, one intended for clients (simply called `Bootstrap`), the other (`ServerBootstrap`) for servers. Regardless of which protocol or protocols your application uses or what type of data processing it performs, the only thing that determines which bootstrap it uses is its function as a "client" or "server".

Connection-oriented vs. Connectionless

Keep in mind that this discussion applies to the TCP protocol, which is "connection-oriented". Such protocols guarantee the ordered delivery of messages between the endpoints of the connection. Connectionless protocols send their messages without any guarantee of ordered, or even successful, delivery.

There are several similarities between the two types of bootstraps; in fact, there are more similarities than there are differences. Table 3.1 shows some of the key similarities and differences.

Table 3.1 Comparison of Bootstrap classes

Category	Bootstrap	ServerBootstrap
Networking function	Connects to a remote host and port	Binds to a local port
Number of `EventLoopGroups`	1	2

Groups, transports and handlers are covered separately later in this chapter, so for now we'll examine only the key differences between the two types of bootstrap classes. The first difference is obvious; as we stated above, a `ServerBootstrap` binds to a port, since servers must listen for connections, while a `Bootstrap` is used in client applications that want to connect to a remote peer.

The second difference is perhaps more significant. Bootstrapping a client requires only a single `EventLoopGroup` while a `ServerBootstrap` requires two (which, however, can be the same instance). Why?

A server actually needs two distinct sets of `Channel`s. The first set will contain a single `ServerChannel` representing the server's own listening socket, bound to a local port. The second set will contain all of the `Channels that have been created to handle` incoming client connections, one for each connection the server has accepted. Figure 3.2 illustrates this model, and shows why two distinct `EventLoopGroup`s are required.

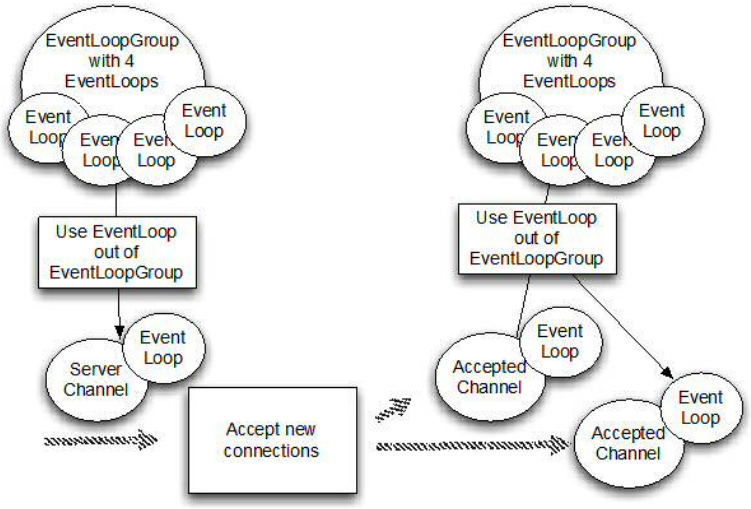


Figure 3.2 Server with two EventLoopGroups

The `EventLoopGroup` associated with the `ServerChannel` assigns an `EventLoop` that is responsible for creating `Channel`s for incoming connection requests. Once a connection has been accepted the second `EventLoopGroup` assigns an `EventLoop` to its `Channel`.

3.4 ChannelHandler and ChannelPipeline

Let's take a look at what happens to data when you send or receive it. Recall our earlier discussion of `ChannelPipeline`s as containers for chains of `ChannelHandler`s whose execution order they also prescribe. In this section we'll go a bit deeper into the symbiotic relationship between these two classes.

In many ways `ChannelHandler` is at the core of your application, even though at times it may not be apparent. `ChannelHandler` has been designed specifically to support a broad range of uses, making it hard to define narrowly. So perhaps it is best thought of as a generic container for any code that processes events (including data) coming and going through the `ChannelPipeline`. This is illustrated in Figure 3.3, which shows the derivation of `ChannelInboundHandler` and `ChannelOutboundHandler` from the parent `interface ChannelHandler`.

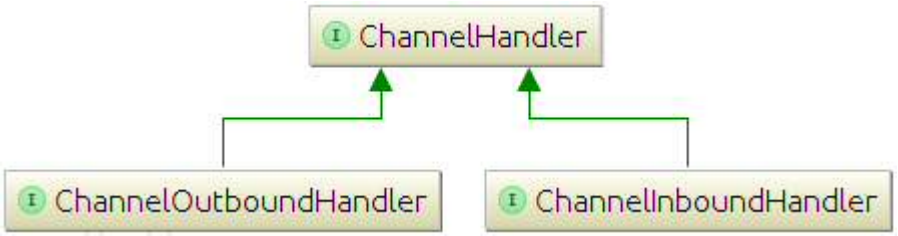


Figure 3.3 ChannelHandler class hierarchy

While it is natural to explain the function of the `ChannelHandler` in terms of data flow it should be noted that the examples used in this discussion are limited in scope. As you will see later on, `ChannelHandlers` can be applied in many other ways as well.

Figure 3.4 illustrates the distinction between inbound and outbound data flow in a Netty application. Events are said to be "outbound" if the movement is from the client application to the server and "inbound" in the opposite case.

The movement of an event through the pipeline is the work of the `ChannelHandler`s that have been installed during the bootstrapping phase. These objects receive the event, execute the processing logic for which they have been implemented and pass the data to the next handler in the chain. The order in which they are executed is determined by the order in which they were added In effect, this ordered arrangement of `ChannelHandler`s is what we refer to as the `ChannelPipeline`.

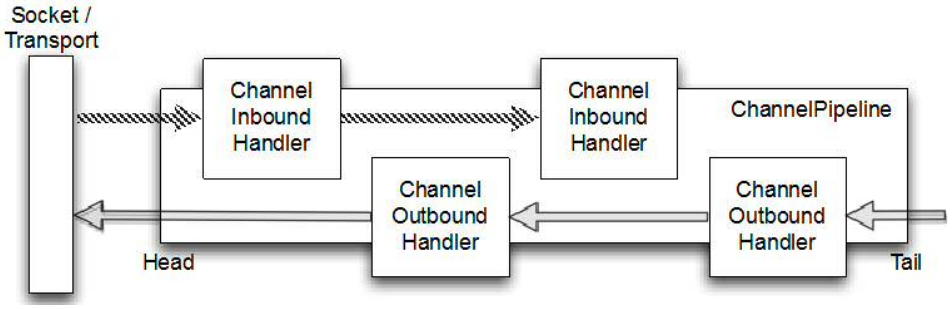


Figure 3.4 ChannelPipeline with inbound and outbound ChannelHandlers

Figure 3.4 also shows that both inbound and outbound handlers can be installed in the same pipeline. In this example, if a message or any other inbound event is read it will start from the head of the pipeline and be passed to the first `ChannelInboundHandler`. This handler may or may not actually modify the data, depending on its specific functionality, after which the data will be passed to the next `ChannelInboundHandler` in the chain. Finally the data will reach the tail of the pipeline, at which point all processing is terminated.

The outbound movement of data (that is, data being "written") is identical in concept. In this case data flows from the tail through the chain of `ChannelOutboundHandler`s until it reaches the head. Beyond this point, outbound data will reach the network transport, shown here as a socket. Typically, this will trigger a write operation.

More on Inbound and Outbound Handlers

An event can be forwarded to the next `handler` in the current chain by using the `ChanneHandlerContext` that is passed in to each method. Because this is what you usually want for events that you are not interested in Netty provides the abstract base classes `ChannelInboundHandlerAdapter` and `ChannelOutboundHandlerAdapter`. Each of these provides an implementation for each method and simply passes the event to the next handler by calling the corresponding method on the `ChannelHandlerContext`. You can then override the method in question to actually do the required processing.

So if outbound and inbound operations are distinct, what happens when handlers are mixed in the same `ChannelPipeline`? Although inbound and outbound handlers both extend `ChannelHandler`, Netty distinguishes between implementations of `ChannelInboundHandler` and `ChannelOutboundHandler`, thus guaranteeing that data is passed only to from one handler to the next handler of the correct type.

When a `ChannelHandler` is added to a `ChannelPipeline` it gets a `ChannelHandlerContext`, which represents the "binding" between a `ChannelHandler` and the `ChannelPipeline`. It is generally safe to hold a reference to this object, except when the protocol in use is not connection-oriented (e.g., UDP). While this object can be used to obtain the underlying `Channel`, it is mostly utilized to write outbound data.

There are, in fact, two ways of sending messages in Netty. You can write directly to the `Channel` or write to the `ChannelHandlerContext` object. The main difference is that the former approach causes the message to start from the tail of the `ChannelPipeline`, while the latter causes the message to start from the next handler in the `ChannelPipeline`.

3.5 A Closer Look at ChannelHandlers

As we said before, there are many different types of `ChannelHandlers`. What each one does depends on its superclass. Netty provides a number of default handler implementations in the form of "adapter" classes. These are intended to simplify the development of your processing logic. We have seen that each `ChannelHandler` in a pipeline is responsible for forwarding events on to the next handler in the chain. These adapter classes (and their subclasses) do this for you automatically, so you need only implement the methods and events that have to be specialized.

Why adapters ?

There are a few adapter classes that really reduce the effort of writing custom `ChannelHandlers` to the bare minimum, since they provide default implementations of all the methods defined in the corresponding interface. (There are also similar adapters for creating coders and decoders, which we'll discuss shortly.)

These are the adapters you'll call most often when creating your custom handlers:

- `ChannelHandlerAdapter`
- `ChannelInboundHandlerAdapter`
- `ChannelOutboundHandlerAdapter`
- `ChannelDuplexHandlerAdapter`

Next we'll examin at three `ChannelHandler` subtypes: encoders, decoders and `SimpleChannelInboundHandler<T>`, a sub-class of `ChannelInboundHandlerAdapter`).

3.5.1 EncoderS AND decoders

When you send or receive a message with Netty a data conversion takes place. An inbound message will be converted from bytes to a Java object; that is, "decoded". If the message is outbound the reverse will happen: "encoding" from a Java object to bytes. The reason is simple: network data is a series of bytes, hence the need to convert to and from that type.

Various types of abstract classes are provided for encoders and decoders, depending on the task at hand. For example, your application may use Netty in a way that doesn't require the message to be converted to bytes immediately. Instead, the message is to be converted to some other format. An encoder will still be used but it will derive from a different superclass To determine the appropriate superclass you can apply a simple naming convention.

In general, base classes will have a name resembling `ByteToMessageDecoder` or `MessageToByteEncoder`. In the case of a specialized type you may find something like `ProtobufEncoder` and `ProtobufDecoder`, used to support Google's protocol buffers.

Strictly speaking, other handlers could do what encoders and decoders do. But just as there are adapter classes to simplify the creation of channel handlers, all of the encoder/decoder adapter classes provided by Netty implement either `ChannelInboundHandler` or `ChannelOutboundHandler`.

For inbound data he `channelRead` method/event is overridden. This method is called by each message that is read from the inbound `Channel`. This method will then call the "decode" method of the specific decoder and forward the decoded message to the next `ChannelInboundHandler in the pipeline`.

The pattern for outbound messages is similar. In this case an encoder converts the message to bytes and forwards them to the next `ChannelOutboundHandler`.

3.5.2 SimpleChannelHandler

Perhaps the most common handler your application will employ is one that receives a decoded message and applies some business logic to the data. To create such a `ChannelHandler`, you need only to extend the base class `SimpleChannelInboundHandler<T>`, where T is the type of message you want to process. In this `handler` you will override one or more methods of the base class and obtain a reference to the `ChannelHandlerContext` which is passed as an input argument to all the methods.

The most important method in a handler of this type is `channelRead0(ChannelHandlerContext,T)`. In this call, T is the message to be processed. How you do that is entirely your concern. Keep in mind, though, that even though you must not block the I/O thread as this could be detrimental to performance in high- throughput environments.

Blocking operations

While the I/O thread must not be blocked at all, thus prohibiting any direct blocking operations within your `ChannelHandler`, there is a way to implement this requirement. You can specify an `EventExecutorGroup` when adding `ChannelHandlers` to the `ChannelPipeline`. This `EventExecutorGroup` will then be used to obtain an `EventExecutor`, which will execute all the methods of the `ChannelHandler`. This `EventExecutor` will use a different thread from the I/O thread, thus freeing up the `EventLoop`.

3.6 Summary

In this chapter we presented an overview of the key components and concepts of Netty and how they fit together. Many of the following chapters are devoted to in-depth study of individual components and this overview should help you to keep the big picture in focus.

The next chapter will explore the different transports provided by Netty and how to choose the transport best suited to your application.

4

Transports

This chapter covers

- Transports
- NIO, OIO, Local, Embedded
- Use-cases
- APIs

One of the most important tasks of a network application is transferring data. This can be done differently depending on the kind of transport used, but what gets transferred is always the same: bytes over the wire. Transports help abstract how the data is transferred. All you need to know is that you have bytes to send and receive. Nothing more, nothing less…

If you've ever worked with a Java-provided way of network programming you've maybe come across a situation when you wanted to switch from blocking transports to non-blocking transports because you need to handle more concurrent connections as you expected first. This switch isn't easily possible, because the network API exposed by Java itself uses different interfaces and classes to handle blocking versus non-blocking.

Netty offers a unified API on top of its transport implementation, which makes the switch easier compared to switch when using the network API shipped with Java. You'll be able to keep your code as generic as possible, and not depend on some implementation-dependent API. There won't be a need to refactor your whole code base when you need to move from one transport to another. If you ever needed to switch from blocking to non-blocking while using the plain network API that comes with the JDK, you already know how massive such changes can be. Don't waste your time with this boring stuff; spend it on something more productive.

This chapter shows you what the unified API of Netty looks like and how to use it. It compares it to the API that comes with the JDK and show you why Netty's API is easier to use. It will also explain the various transport implementations that are bundled with Netty and which is preferred for each use case. After gathering this information, you'll be able to choose the best option for your specific application, giving you the best result for your use case.

In ordert o understand what we discuss in the chapter, you only need experience with Java. Having experience with network frameworks or network programming can help but isn't needed.

Let's see how transports work in a real-world situation.

4.1 Case study: transport migration

To give you an idea how transports work, I'll start with a simple application which does nothing but accept client connections and write "Hi!" to the client. After that's done, it disconnects the client.

4.1.1 Using I/O and NIO without Netty

To start this case study I'll implement the application without Netty. Listing 4.1 shows the code using a blocking input/output (I/O). If you ever had the joy to do any network programming with Java the code will look very familiar to you.

Listing 4.1 Blocking networking without Netty

```
public class PlainOioServer {

    public void serve(int port) throws IOException {
        final ServerSocket socket = new ServerSocket(port);              #1
        try {
            while (true) {
                final Socket clientSocket = socket.accept();             #2
                System.out.println("Accepted connection from " + clientSocket);

                new Thread(new Runnable() {                              #3
                    @Override
                    public void run() {
                        OutputStream out;
                        try {
                            out = clientSocket.getOutputStream();
                            out.write("Hi!\r\n".getBytes(Charset.forName("UTF-8")));
      #4
                            out.flush();
                            clientSocket.close();                        #5

                        } catch (IOException e) {
                            e.printStackTrace();
                            try {
                                clientSocket.close();
                            } catch (IOException ex) {
                                // ignore on close
                            }
                        }
                    }
                }).start();                                              #6
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

#1 Bind server to port
#2 Accept connection
#3 Create new thread to handle connection
#4 Write message to connected client
#5 Close connection once message written and flushed
#6 Start thread to begin handling

This code works fine and writes "Hi!" to the remote peer, but after some time you may notice that the blocking handling doesn't scale enough for your use case as you expect 10th of thousands of concurrent connections. You want to use asynchronous networking to handle all the concurrent connections, but the problem is that the API is completely different. So you are

forced to rewrite your application completely. The code for non-blocking api is shown in Listing 4.2.

Listing 4.2 Asynchronous networking without Netty

```
public class PlainNioServer {

    public void serve(int port) throws IOException {
        System.out.println("Listening for connections on port " + port);
        ServerSocketChannel serverChannel;
        Selector selector;

        serverChannel = ServerSocketChannel.open();
        ServerSocket ss = serverChannel.socket();
        InetSocketAddress address = new InetSocketAddress(port);
        ss.bind(address);                                                    #1
        serverChannel.configureBlocking(false);
        selector = Selector.open();                                          #2
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);            #3
        final ByteBuffer msg = ByteBuffer.wrap("Hi!\r\n".getBytes());

        while (true) {

            try {
                selector.select();                                           #4
            } catch (IOException ex) {
                ex.printStackTrace();
                // handle in a proper way
                break;
            }

            Set<SelectedKey> readyKeys = selector.selectedKeys();            #5
            Iterator<SelectedKey> iterator = readyKeys.iterator();
            while (iterator.hasNext()) {
                SelectionKey key = iterator.next();
                iterator.remove();
                try {
                    if (key.isAcceptable()) {                                #6
                        ServerSocketChannel server = (ServerSocketChannel)
    key.channel();
                        SocketChannel client = server.accept();
                        System.out.println("Accepted connection from " + client);
                        client.configureBlocking(false);
                        client.register(selector, SelectionKey.OP_WRITE |
    SelectionKey.OP_READ, msg.duplicate());                                  #7
                    }
                    if (key.isWritable()) {                                  #8
                        SocketChannel client = (SocketChannel) key.channel();
                        ByteBuffer buffer = (ByteBuffer) key.attachment();
                        while (buffer.hasRemaining()) {
                            if (client.write(buffer) == 0) {                 #9
                                break;
                            }
                        }
                        client.close();                                     #10
                    }
                } catch (IOException ex) {
                    key.cancel();
```

```
                        try {
                            key.channel().close();
                        } catch (IOException cex) {
                        }
                    }
                }
            }
        }
    }
```

#1 Bind server to port
#2 Open selector that handles channels
#3 Register erverSocket to selector and specify that it is interested in new accepted clients
#4 Wait for new events that are ready for process. This will block until something happens
#5 Obtain all SelectionKey instances that received events
#6 Check if event was because new client ready to get accepted
#7 Accept client and register it to selector
#8 Check if event was because socket is ready to write data
#9 Write data to connected client. This may not write all the data if the network is saturated. If so it will pick up the not-written data and write it once the network is writable again.
#10 Close connection

As you can see, the code is completely different, even if it does exactly the same as the code shown in Listing 4.1. Even for this simple application the code looks completely different. Consider the work it would take to port something more complex.

With this in mind, let's implement the same application with Netty.

4.1.2 Using I/O and NIO with Netty

Start by writing a blocking version of the application, but this time use Netty as a network framework, as shown in the following listing.

Listing 4.3 Blocking networking with Netty

```
public class NettyOioServer {

    public void server(int port) throws Exception {
        final ByteBuf buf = Unpooled.unreleaseableBuffer(
           Unpooled.copiedBuffer("Hi!\r\n", Charset.forName("UTF-8")));
        EventLoopGroup group = new OioEventLoopGroup();
        try {
            ServerBootstrap b = new ServerBootstrap();                    #1
            b.group(group)                                                #2
             .channel(OioServerSocketChannel.class)
             .localAddress(new InetSocketAddress(port))
             .childHandler(new ChannelInitializer<SocketChannel>() {      #3
                 @Override
                 public void initChannel(SocketChannel ch)
                     throws Exception {
                     ch.pipeline().addLast(
                             new ChannelInboundHandlerAdapter() {         #4
                         @Override
                         public void channelActive(
                             ChannelHandlerContext ctx) throws Exception {
                             ctx.writeAndFlush(buf.duplicate())
                               .addListener(ChannelFutureListener.CLOSE);#5
```

```
                        }
                    });
                }
            });
            ChannelFuture f = b.bind().sync();                          #6
            f.channel().closeFuture().sync();
        } finally {
            group.shutdownGracefully().sync();                          #7
        }
    }
}
```

#1 Create ServerBootstrap to allow bootstrap to server instance
#2 Use OioEventLoopGroup Ito allow blocking mode (Old-IO)
#3 Specify ChannelInitializer that will be called for each accepted connection
#4 Add ChannelHandler to intercept events and allow to react on them
#5 Write message to client and add ChannelFutureListener to close connection once message written
#6 Bind server to accept connections
#7 Release all resources

You may notice the code shown in Listing 4.3 is a lot shorter and compact compared to the code shown in Listing 4.1, while it does exactly the same.

Let's move on and implement the same logic but use non-blocking networking.

4.1.3 Using non-blocking networking

You'll see that the code in the following listing looks much the same as listing 4.3. A change of two lines of code is all that's needed to switch from blocking to asynchronous mode. You swap out the OIO transport for the NIO.

OIO and NIO

OIO stands for Old-IO and is blocking while NIO stands for New-IO and is non-blocking.

The following listing shows the changed lines in bold.

Listing 4.4 Asynchronous networking with Netty

```
public class NettyNioServer {

    public void server(int port) throws Exception {
        final ByteBuf buf = Unpooled.copiedBuffer("Hi!\r\n", Charset.forName("UTF-
      8"));
        EventLoopGroup group = new NioEventLoopGroup();
        try {
            ServerBootstrap b = new ServerBootstrap();                  #1
            b.group(group)
             .channel(NioServerSocketChannel.class)
             .localAddress(new InetSocketAddress(port))
             .childHandler(new ChannelInitializer<SocketChannel>() {    #3
                 @Override
                 public void initChannel(SocketChannel ch)
                     throws Exception {
```

```
                    ch.pipeline().addLast(
                            new ChannelStateHandlerAdapter() {          #4
                        @Override
                        public void channelActive(
                            ChannelHandlerContext ctx) throws Exception {
                            ctx.writeAndFlush(buf.duplicate())
                              .addListener(ChannelFutureListener.CLOSE);#5
                        }

                        @Override
                        public void inboundBufferUpdated(
                            ChannelHandHandlerContext ctx)
                            Throws Exception {
                            czx.fireInboundBufferUpdated();
                        }

                    });
                }
             });
            ChannelFuture f = b.bind().sync();                          #6
            f.channel().closeFuture().sync();
        } finally {
            group.shutdownGracefully().sync();                          #7
        }
    }
}
```

#1 Create ServerBootstrap to allow bootstrap to server
#2 Use NioEventLoopGroup forl nonblocking mode
#3 Specify ChannelInitializer called for each accepted connection
#4 Add ChannelHandler to intercept events and allow to react on them
#5 Write message to client and add ChannelFutureListener to close connection once message written
#6 Bind server to accept connections
#7 Release all resources

Because Netty exposes the same API for every transport implementation, it doesn't matter what implementation you use. Netty exposes its operations through the `Channel` interface and its `ChannelPipeline` and `ChannelHandler`.

Now that you've seen Netty in action, let's take a deeper look at the transport API.

4.2 Transport API

At the heart of the transport API is the `Channel` interface, which is used for all of the outbound operations.

See the hierarchy of the `Channel` interface as shown in figure 4.1.

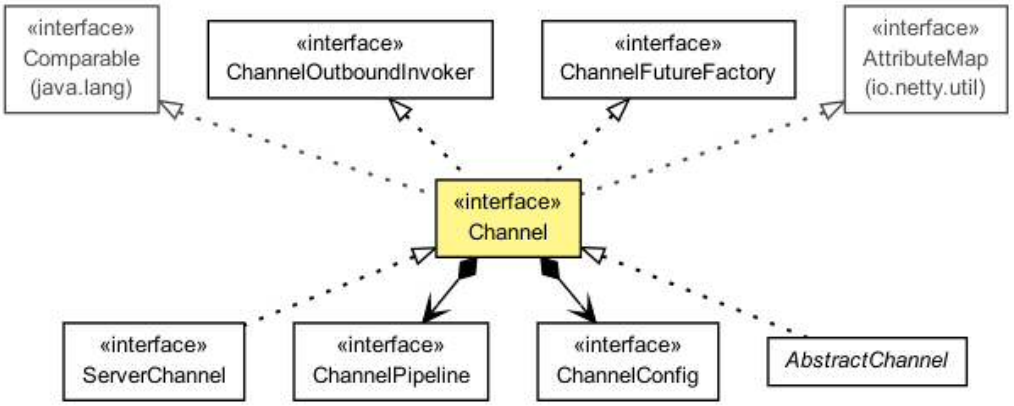


Figure 4.1 Channel interface hierarchy

As you can see in figure 4.1, a `Channel` has a `ChannelPipeline` and a `ChannelConfig` assigned to it.

The `ChannelConfig` has the entire configuration settings stored for the `Channel` and allows for updating them on the fly. Often a transport has specific configuration settings that are possible only on the transport and not on other implementations. For this purpose the transport may expose a subtype of `ChannelConfig`. For more information, refer to the javadocs of the specific `ChannelConfig` implementation.

The `ChannelPipeline` holds all of the `ChannelHandler` instances that should be used for the inbound and outbound data that is passed through the `ChannelPipeline`. These `ChannelHandler` implementations allow you to react to state changes or transform data.

Chapter 6 covers the details of ChannelHandlers, as these are one of the key concepts of Netty.

For now, I'll note that you can use `ChannelHandler` for these tasks:

- Transforming data from one format to another.
- Notifying you of exceptions.
- Notifying you when a `Channel` becomes active or inactive.
- Notifying you once a `Channel` is registered/deregistered from an `EventLoop`.
- Notifying you about user-specific events.

These `ChannelHandler` instances are placed in the `ChannelPipeline`, where they execute one after the other.

ChannelPipeline

The `ChannelPipeline` implements the Intercepting Filter Pattern, which means you can chain different `ChannelHandlers` and intercept the data or events which go through the ChannelPipeline.

Think of it like UNIX pipes, which allows to chain different commands (where the `ChannelHandler` would be the command here).

You can also modify the `ChannelPipeline` on the fly, which allows you to add/remove `ChannelHandler` instances whenever needed. This can be used to build highly flexible applications with Netty. For example you could make use of this to support STARTTLS, which means you would add a `ChannelHandler` (the `SslHandler`) into the `ChannelPipeline` once STARTTLS was requested and so support secure communication.

In addition to accessing the assigned `ChannelPipeline` and `ChannelConfig`, you can also make use of the `Channel` itself. The `Channel` provides various methods, but the most important ones are listed in Table 4.1.

Table 4.1 Most important channel methods

Method name	Description
`eventLoop()`	Returns the `EventLoop` that is assigned to the `Channel`
`pipeline()`	Returns the `ChannelPipeline` that is assigned to the `Channel`
`isActive()`	Returns if the `Channel` is active. What active means here depends on the underlying transport. For example a Socket transport would be active once connected to the remove peer. While a Datagram transport would be active once it is open.
`localAddress()`	Returns the `SocketAddress` that is bound local
`remoteAddress()`	Returns the `SocketAddress` that is bound remote
`write(…)`	Writes data to the remote peer. This data is passed though the `ChannelPipeline` and queued until it is flushed.
`flush()`	Flush the previous written data to the underlying transport, which for example could be a socket.
`writeAndFlush(…)`	Shortcut for first call `write(…)` and then `flush()`

You'll learn more later about how you can use all of these features. Remember for now that you'll always operate on the same interfaces, which gives you a high degree of flexibility and guards you from big refactoring once you want to try out a different transport implementation.

A very common task is writing data and flushing it to the remote peer. This means you transmit data. For this you'd call `Channel.writeAndFlush(...)` as shown in the following listing.

Listing 4.5 Writing to a channel

```
Channel channel = ...
ByteBuf buf = Unpooled.copiedBuffer(„your data", CharsetUtil.UTF_8);     #1
ChannelFuture cf = channel.writeAndFlush(buf);                           #2

cf.addListener(new ChannelFutureListener() {                             #3

    @Override
    public void operationComplete(ChannelFuture future) {
        if (future.isSuccess()) {                                        #4
            System.out.println("Write successful");
        } else {
            System.err.println("Write error");                           #5
            future.cause().printStacktrace();
        }
    }
});
```

#1 Create ByteBuf that holds data to write
#2 Write data and flush it
#3 Add ChannelFutureListener to get notified after write completes
#4 Write operation completes without error
#5 Write operation completed but because of error

Please note that `Channel` is thread-safe, which means it is safe to operate on it from different `Threads`. All it's methods are safe to use in a multi-thread enviroment. Because of this it's safe to store a reference to the `Channel` in your application and use it once the need arises to write something to the remote peer, even when using many threads. The following listing shows a simple example of writing with multiple threads.

Listing 4.6 Using the channel from many threads

```
final Channel channel = ...
final ByteBuf buf = Unpooled.copiedBuffer(„your data",
        CharsetUtil.UTF_8);                                              #1
Runnable writer = new Runnable() {                                       #2
    @Override
    public void run() {
        channel.write(buf.duplicate());
    }
};
Executor executor = Executors.newChachedThreadPool();                    #3

// write in one thread
   executor.execute(writer);                                             #4

// write in another thread
   executor.execute(writer);                                             #5
     …
```

#1 Create ByteBuf that holds data to write
#2 Create Runnable which writes data to channel
#3 Obtain reference to the Executor which uses threads to execute tasks
#4 Hand over write task to executor for execution in thread
#5 Hand over another write task to executor for execution in thread

Also, this method guarantees that the messages are written in the same order as you passed them to the write method. For a complete reference of all methods, refer to the provided API documentation (javadocs).

Knowing the interfaces used is important, but it's also quite helpful to know what different transport implementations ship with Netty. Chances are good that everything is already provided for you.

In the next section I'll look at what implementations are provided and what their behaviors are.

4.3 Included transports

Netty already comes with a handful of transports that you can use. Not all of them support all protocols, which means the transport you want to use also depends on the underlying protocol that your application depends on. You'll learn more about which transport supports which protocol in this section.

Table 4.1 shows all of the transports that are included by default in Netty.

Table 4.1 Provided transports

Name	Package	Description
NIO	`io.netty.channel.socket.nio`	Uses the `java.nio.channels` package as a foundation and so uses a selector-based approach.
OIO	`io.netty.channel.socket.oio`	Uses the `java.net package` as a foundation and so uses blocking streams.
Local	`io.netty.channel.local`	A local transport that can be used to communicate in the VM via pipes.
Embedded	`io.netty.channel.embedded`	Embedded transport, which allows using `ChannelHandlers` without a real network based Transport. This can be quite useful for testing your `ChannelHandler` implementations.

Now let's go into more detail by first looking into the most-used transport implementation, the NIO transport.

4.3.1 NIO – Nonblocking I/O

The NIO transport is currently the most used. It provides a full asynchronous implementation of all I/O operations by using the selector-based approach that's included in Java since Java 1.4 and the NIO subsystem.

The idea is that a user can register to get notified once a `Channels` state changes. The possible changes are:

- A new `Channel` was accepted and is ready.
- A `Channel` connection was completed.
- A `Channel` has data received that is ready to be read.
- A `Channel` is able to send more data on the channel.

The implementation is then responsible for reacting to these state changes to reset them and to be notified once a state changes again. This is done with a thread that checks for updates and, if there are any, dispatches them accordingly.

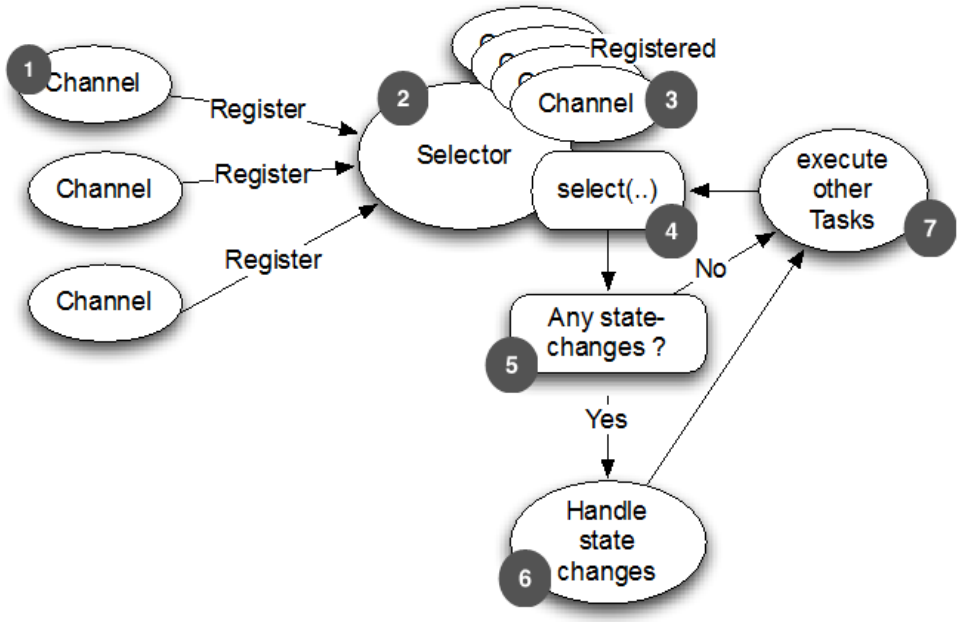
Here it's possible to register to be notified for only one of the events and ignore the rest.

The exact bit-sets that are supported by the underlying `Selector` are shown in Table 4.2. These are defined in the `SelectionKey` class.

Table 4.2 Selection operation bit-set

Name	Description
`OP_ACCEPT`	Get notified once a new connection is accepted and a `Channel` is created.
`OP_CONNECT`	Get notified once a connection attempt finishes.
`OP_READ`	Get notified once data is ready to be read out of the `Channel`.
`OP_WRITE`	Get notified once it's possible to write more data to the `Channel`. Most of the time this is possible, but it may not be because the OS socket buffer is completely filled. This usually happens when you write faster then the "remote peer" can handle it.

Netty's NIO transport uses this model internally to receive and send data, but exposes its own API to the user, which completely hides the internal implementation. As mentioned previously, that helps to expose only one unified API to the user, while hiding all of the internals. Figure 4.2 shows the process flow.



#1 New created channel that registers to selector
#2 Selector to handle state changes
#3 Already registered channels
#4 The Selector.select() method which blocks until new state changes received or given timeout elapsed
#5 Check if there were state changes
#6 Handle all state changes
#7 Execute other tasks in same thread in which selector operates

Figure 4.2 Selecting Events and process them

One feature that only the NIO transport offers at the moment is called "zero-file-copy". This feature allows you to quickly and efficiently transfer content from your file system. The feature provides a way to transfer the bytes from the file system to the network stack without copying the bytes from the kernel space to the user space.

Be aware that not all operation systems support „zero-file-copy". Please refer to operating system's documentation to find out if it's supported. Also, be aware that you'll only be able to benefit from this if you don't use any encryption/compression of the data. Otherwise it will need to copy the bytes first to the user space to do the actual work, so only transferring the raw content of a file makes use of this feature. What actually would work is to „pre-encrypt" a file before transfering it.

One application that can really make use of this is an FTP or HTTP server that downloads big files.

The next transport I'll discuss is the OIO transport, which provides a blocking transport.

4.3.2 OIO – Old blocking I/O

The OIO transport is a compromise in Netty. It builds on the known unified API but isn't asynchronous by nature because it uses the blocking `java.net` implementations under the covers. At first glance, this transport may not look useful to you, but it has its use cases.

Suppose you need to port some legacy code that uses many libraries that do blocking calls (such as database calls via JDBC[6]). It may not be feasible to port the logic to not block. Instead, you could use the OIO transport in the short term and port it later to one of the pure asynchronous transports. Let's focus on how it works.

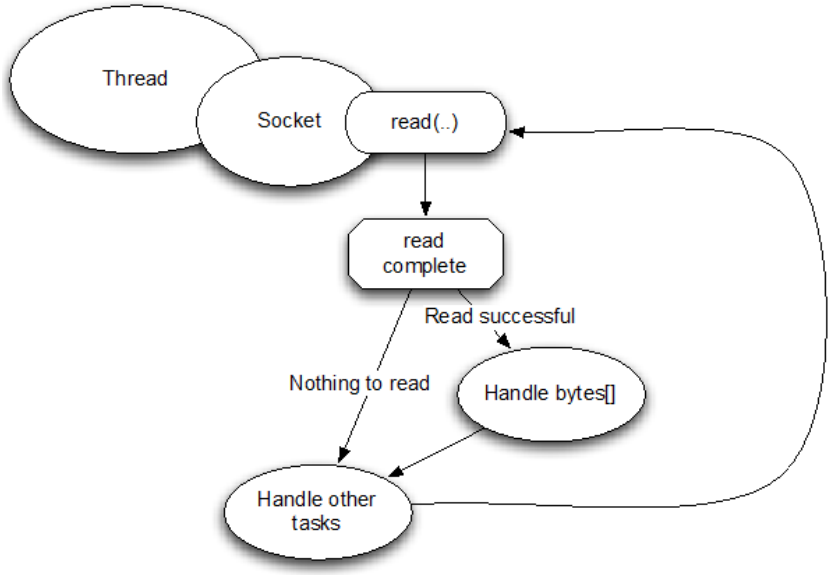
Because the OIO transport uses the `java.net` classes internally, it also uses the same logic that you may already be familiar with if you previously written network applications.

When using these classes, you usually have one thread that handles the acceptance of new sockets (server-side) and then creates a new thread for each accepted connection to serve the traffic over the socket. This is needed as every I/O operation on the socket may block at any time. If you share the same thread over more than one connection (socket), this could lead to a situation where blocking an operation could block all other sockets from doing their work.

Knowing that operations may block, you may start to wonder how Netty uses it while still providing the same way of building APIs. Here Netty makes use of the `SO_TIMEOUT` that you can set on a socket. This timeout specifies the maximum number of milliseconds to wait for an I/O operation to complete. If the operation doesn't complete within the specified timeout, a `SocketTimeoutException` is thrown. Netty catches this `SocketTimeoutException` and moves on with its work. Then on the next `EventLoop` run, it tries again. Unfortunately, this is the only way to do this and still confirm the inner working of Netty. The problem with this approach is that firing the `SocketTimeoutException` isn't free, as it needs to fill in the `StrackTrace` on every `SocketTimeoutException`.

Figure 4.3 shows the logic.

[6] http://www.oracle.com/technetwork/java/javase/jdbc/index.html



#1 Thread allocated to socket
#2 Socket connected to remote peer
#3 Read operation that may block
#4 Read complete
#5 Read completed and able to read bytes so handled them
#6 Execute other tasks submitted belonging to socket
#7 Try to read again

Figure 4.3 OIO-Processing logic

Now you know the two most used transports in Netty, but there are still others which we will cover in the next sections.

4.3.3 Local – In VM transport

Netty contains the so-called local transport. This transport implementation can be used to communicate within a VM and still use the same API you're used to. The transport is fully asynchronous and so not blocks at all.

Every `Channel` uses and unique `SocketAddress` that is stored in a registry. This `SocketAddress` can then be used to connect to it via the client. It will be registered as long as the server is running. Once the `Channel` is closed, it will automatically deregister it and so the clients won't be able to access it anymore.

The behavior of connecting to a local transport server is nearly the same as with other transport implementations. One important thing to note is that you can only make use of them on the server and client side at the same time. It's not possible to use the local transport on the server side but some other on the client side. This may seem like a limitation, but once

you think about it, it makes sense. As the local transport does not bound a socket and so does not accept "real" network traffic, it's not possible to work with any other transport implementation.

4.3.4 Embedded transport

Netty also includes the so called embedded transport. This isn't a real transport when you compare it with the others listed previously, but it's included to complete this section. When we say "not a real transport" we just coin it like this because you can not use it to transfer data of the network. But If it's not a real transport, what can it be used for?

The embedded transport allows you to interact with your different `ChannelHandler` implementation more easily. It's also easy to embed `ChannelHandler` instances in other `ChannelHandlers` and use them like a helper class.

This is often used in test cases to test a specific `ChannelHandler` implementation, but can also be used to re-use some `ChannelHandler` in your own `ChannelHandler` without extend it. For this purpose, it comes with a concrete `Channel` implementation that is called `EmbeddedChannel`.

Chapter 10 gives more information how the `EmbeddedChannel` can be used within unit-tests to test a `ChannelHandler` implementation.

4.4 When to use each type of transport

Now that you've learned about all the transports in detail, you may ask when you should choose one over the other. As mentioned previously, not all transports support all core protocols. This can also limit the transports you can choose from. Table 4.3 lists the protocols that each transport supports.

Table 4.3 Transport support by network protocol

Transport	TCP	UDP	SCTP*	UDT
NIO	X	X	X	X
OIO	X	X	X	X

* Only supported on Linux at the moment. This may change in the future..The content of the table reflects what is supported at the time of publication.

Enabling SCTP on Linux

Be aware that for SCTP, you'll need to have the user space libraries installed as well as a kernel that supports it.

For Ubuntu, use the following command:

```
# sudo apt-get install libsctp1
```

For Fedora, you use yum:

```
# sudo yum install kernel-modules-extra.x86_64 lksctp-tools.x86_64
```

Please refer to the documentation of each Linux distribution for more information about how to enable SCTP.

Other than the SCTP[7] items, there aren't any hard rules, but because of the nature of the transports there are some suggestions. It's also likely that you have to handle more concurrent connections when you implement a server than if you implement a client.

Let's take a look at the use-cases that you're likely to experience.

NON-BLOCKING CODE-BASE

If you don't have blocking calls in your code-base or can limit them to some extend it is always a good idea to use NIO. While NIO is mostly "designed" to handle a lot of concurrent connections it also works out quite well with a small number of concurrent-connections. The NIO transport allows handling a massive amount of concurrent connections with a small number of Threads as they don't use one Thread per connection, but use a few threads and share them across the connections.

BLOCKING CODE BASE

If you're converting an old code base, which was heavily based on blocking networking and application design, to Netty, then you're likely to have several operations that would block the I/O thread for too long to scale efficiently with an asynchronous transport such as NIO. You could fix this by rewriting your entire stack, but this is may not be doable in the target timeframe. In that case, start with OIO and when you think you need more scale, move over to NIO once you have rewritten your code-base.

COMMUNICATE WITHIN THE SAME JVM

If you only need to communicate within the same VM and have no need to expose the service over the network, then you have the perfect use case for the local transport. This is because you'll remove all the overhead of real network operations, but still be able to reuse your Netty code base.

This also makes it easy later to expose the service over the network if the need arises. The only thing that needs to be done in that case is to replace the transport with NIO or

[7] http://www.ietf.org/rfc/rfc2960.txt

OIO, and maybe add an extra encoder/decoder that converts Java objects to `ByteBuf` and vice versa.

TESTING YOUR CHANNELHANDLER IMPLEMENTATIONS

If you want to write tests for your `ChannelHandler` implementations that aren't integration tests, give the embedded transport a spin. It makes it easy to test `ChannelHandlers` without the need for creating many mocks, while still conforming to the event flow that is the same in all transports. This guarantees that the `ChannelHandler` will also work once you put it to use with some real transports.

Chapter 7 gives you more details about testing `ChannelHandlers`.

As you now know, it's important to understand what kind of use case/nature your application has, and always choose the transport that will give you the best result. Table 4.4 summarizes the common use cases.

Table 4.4 Optimal transport for an application

Application needs	Recommended transport
Non-Blocking code base or general starting point	NIO
Blocking code base	OIO
Communication within the same JVM	Local
Testing your `ChannelHandler` implementations	Embedded

4.5 Summary

In this chapter you learned one of the fundamentals of Netty, and how it's provided to the user. You learned what a transport is and what it's used for. Also, I explained the API and gave some examples of its uses.

I highlighted the shipped transports of Netty and explained their behavior. You learned what minimum requirements the transports have, as not all transports work with the same Java version or may only work on specific operating systems.

You also learned which transport should be used for which use case, as not every transport is optimal for a specific use case.

The next chapter will focus on `ByteBuf` and `ByteBufHolder`, which are "data containers" used within Netty to transfer data. You'll learn how to use it and how you can create the best performance from it.

5

Buffers

This chapter covers

- `ByteBuf`
- `ByteBufHolder`
- `ByteBufAllocator`
- Allocating and performing operations on these interfaces

Whenever you need to transmit data it must involve bytes. Java's NIO API comes with its own `ByteBuffer` class which acts as container for bytes,. Working with the JDK's `ByteBuffer` is often cumbersome and more complex than needed.

Luckily, Netty comes with a powerful buffer implementation that's used to represent a sequence of bytes. The new buffer type, the `ByteBuf`, is effectively Netty's equivalent to the JDK's `ByteBuffer`. The `ByteBuf`'s purpose is to pass data through the Netty pipeline. It was designed from the ground up to address problems with the JDK's `ByteBuffer` and to meet the daily needs of networking application developers, making them more productive. This approach has significant advantages over using the JDK's `ByteBuffer`.

In this chapter, you'll learn about Netty's buffer API, how it's superior to what the JDK provides out of the box, what makes it so powerful, and why it's more flexible than the JDK's buffer API. You'll gain a deeper understanding of how to access data that's exchanged in the Netty framework and how you can work with the data. This chapter also builds the groundwork for later chapters, as the buffer API is used nearly everywhere in Netty.

Because the `ByteBuf` is passed through Netty's `ChannelPipeline` and `ChannelHandler` implementations, the `ByteBuf` is prevalent in the day-to-day development of Netty applications. `ChannelHandler` and `ChannelPipeline` will be discussed in detail in chapter 6.

5.1 Buffer API

Netty's buffer API is exposed through:

- `ByteBuf`
- `ByteBufHolder`

Netty uses reference-counting (more on this in section 5.6) to know when it's safe (like the `ByteBuf` is not used anymore) to release a `ByteBuf and ByteBufHolder` and its claimed resources. This allows Netty to use pooling and other tricks to speed things up and keep the memory utilization at a sane level. You aren't required to do anything to make this happen, but when developing a Netty application, you should try to process your data and release pooled resources as soon as possible. This is a general advice, but is especially important for `ByteBuf` and `ByteBufHolder` as these may be pooled.

Netty's buffer API offers several more advantages:

- You can define your own buffer type, if necessary.
- Transparent zero copy is achieved by a built-in composite buffer type.

- Capacity is expanded on demand, such as with `StringBuilder`.
- No need to call `flip()` to switch between reader/writer mode.
- Separate reader and writer index.
- Method chaining.
- Reference counting.
- Pooling.

We'll have a deeper look at some of these, including pooling, in later sections of this chapter. But let us start with looking at `ByteBuf` itself before we move on to `ByteBufHolder`.

5.2 ByteBuf—The byte data container

Whenever you need to interact with a remote peer such as a database, the communication needs to be done in bytes. For this and other reasons an efficient, convenient, and easy-to-use data structure is required, and Netty's ByteBuf implementation meets these requirements and more, making it an ideal data container, optimized for holding and interacting with bytes.

`ByteBuf` is a data container that allows you to add/get bytes from it in an efficient way. To make it easier to operate, it uses two indices: one for reading and one for writing. This allows you to read data out of them in a sequential way and "jump" back to read it again. All you need to do is adjusting the reader index and start the read operation again or use one of the various get operation that allow to specify an index.

5.2.1 How it works

After something is written to the `ByteBuf`, its `writerIndex` is increased by the amount of bytes written. After you start to read bytes, its `readerIndex` is increased. You can read bytes until the `writerIndex` and `readerIndex` are at the same position. The `ByteBuf` then becomes unreadable, so the next read request triggers an `IndexOutOfBoundsException` similar to what you've seen when trying to read beyond the capacity of an array.

Calling any of the `ByteBuf` methods beginning with "`read`" or "`write`" automatically advances the reader and writer indexes or you. There are also relative operations to "`set`" and "`get`" bytes. These don't move the indexes but operate on the relative index that was given.

A `ByteBuf` may have a maximum capacity to set an upper limit to the maximum data it can hold, trying to move the writer index beyond this capacity will result in an exception. The default limit is `Integer.MAX_VALUE`.

Figure 5.2 shows how a `ByteBuf` is laid out.

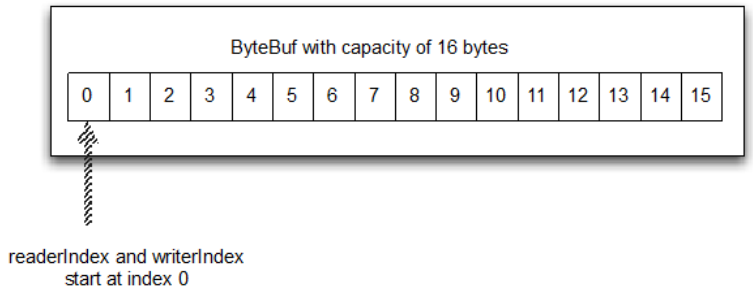


Figure 5.1 A 16-byte ByteBuf, initialized with the read and write indices set to 0

As Figure 5.1 shows, a `ByteBuf` is similar to a byte array, the most notable difference being the addition of the read and write indices which can be used to control access to the buffer's data.

You'll learn more about the operations that can be performed on a `ByteBuf` in a later section. For now, keep in mind that a ByteBuf is like an array, and let's review the different types of `ByteBuf` that you'll most likely use.

5.2.2 Different types of ByteBuf

There are three different types of `ByteBuf` you'll encounter when using Netty (there are more, but these are used internally). You may end up implementing your own, but this is out of scope here. Let's look at the provided types that you are most likely interested in.

HEAP BUFFERS

The most used type is the `ByteBuf` that stores its data in the heap space of the JVM. This is done by storing it in a backing array. This type is fast to allocate and also de-allocate when you're not using a pool. It also offers a way to directly access the backing array, which may make it easier to interact with "legacy code".

Listing 5.1 Access backing array

```
ByteBuf heapBuf = ...;
if (heapBuf.hasArray()) {                                          #1
    byte[] array = heapBuf.array();                                #2
    int offset = heapBuf.arrayOffset() + heapBuf.position();       #3
    int length = heapBuf.readableBytes();                          #4

    YourImpl.method(array, offset, length);                        #5

}
```

#1 Check if ByteBuf is backed by array
#2 Get reference to array
#3 Calculate offset of first byte in it
#4 Get amount of readable bytes
#5 Call method using array, offset, length as parameter

Accessing the array from a "nonheap" `ByteBuf` will result in an `UnsupportedOperationException`. Because of this it's always a good idea to check if the `ByteBuf` is backed by an array with `hasArray()` as shown in listing 5.1.

This pattern might be familiar if you've worked with the JDK's `ByteBuffer` before, if you didn't work with `ByteBuffer` before don't worry, as you will learn everything needed through this chapter.

DIRECT BUFFERS

Another `ByteBuf` implementation is the "direct" one. Direct means that it allocates the memory directly, which is outside the "heap". You won't see its memory usage in your heap space. You must take this into account when calculating the maximum amount of memory your application will use and how to limit it, as the max heap size won't be enough. Direct buffers on the other side are optimal when it's time to transfer data over a socket. In fact, if you use a nondirect buffer (heap buffer), the JVM will make a copy of your buffer to a direct buffer internally before sending it over the socket.

The down side of direct buffers is that they're more expensive to allocate and de-allocate compared to heap buffers. This is one of the reasons why Netty supports pooling, which makes this problem disappear. Another possible down side can be that you're no longer able to access the data via the backing array, so you'll need to make a copy of the data if it needs to work with legacy code that requires this.

The following listing shows how you can get the data in an array and call your method even without the ability to access the backing array directly.

Listing 5.2 Access data

```
ByteBuf directBuf = ...;
if (!directBuf.hasArray()) {                                    #1
    int length = directBuf.readableBytes();                     #2
       byte[] array = new byte[length];                         #3
    directBuf.getBytes(array);                                  #4
    YourImpl.method(array, 0, array.length);                    #5

}
```

#1 Check if ByteBuf not backed by array which will be false for direct buffer
#2 Get number of readable bytes
#3 Allocate new array with length of readable bytes
#4 Read bytes into array
#5 Call method that takes array, offset, length as parameter

As you can see it's a bit more work and involves a copy operation. If you expect to access the data and need to have it in an array, you may be better off using a heap buffer.

COMPOSITE BUFFERS

The last `ByteBuf` implementation you may be confronted with is the `CompositeByteBuf`. This does exactly what its name says; it allows you to compose different `ByteBuf` instances and provides a view over them. The good thing is you can also add and remove `ByteBufs` on-the-

fly. If you've ever worked with the JDK's `ByteBuffer` you've most likely missed such a feature there as it not exist. As the `CompositeByteBuf` is just a "view" over other `ByteBuf`, the `hasArray()` method will return false because it may contain several `ByteBuf` instances of both direct and nondirect types.

For example, a message could be composed of two parts: header and body. In a modularized application, the two parts could be produced by different modules and assembled later when the message is sent out. Also, you may use the same body all the time and just change the header. So it would make sense here to not allocate a new buffer every time.

This would be a perfect fit for a `CompositeByteBuf` as no memory copy will be needed and the same API could be used as with non-composite buffers.

Figure 5.2 shows how a `CompositeByteBuf` would be used to compose the header and body.

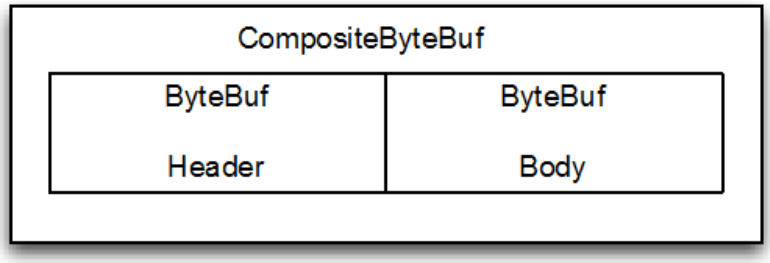


Figure 5.2 CompositeBuf that holds a header and body.

In contrast if you'd used the JDK's `ByteBuffer` this would be impossible. The only way in the JDK to compose two `ByteBuffers` is to create an array that holds them or create a new `ByteBuffer` and copy the contents of both of them to the newly created one. The following listing shows these.

Listing 5.3 Compose legacy JDK `ByteBuffer`

```
// Use an array to composite them
 ByteBuffer[] message = new ByteBuffer[] { header, body };

 // Use copy to merge both
 ByteBuffer message2 = ByteBuffer.allocate(
                          header.remaining()+ body.remaining();
 message2.put(header);
 message2.put(body);
 message2.flip();
```

Both approaches shown in figure 5.3 have disadvantages: having to deal with an array won't allow you to keep the API simple if you want to support both. And, of course, there's a performance cost associated with this copying; it's simply not optimal.

But let's see the `CompositeByteBuf` in action in Listing 5.4.

Listing 5.4 CompositeByteBuf in action

```
CompositeByteBuf compBuf = ...;
ByteBuf heapBuf = ...;
ByteBuf directBuf = ...;

compBuf.addComponent(heapBuf, directBuf);                    #1
.....
compBuf.removeComponent(0);                                  #2

for (ByteBuf buf: compBuf) {                                 #3
    System.out.println(buf.toString());
}
```

#1 Append ByteBuf instances to the composite
#2 Remove ByteBuf on index 0 (heapBuf here)
#3 Loop over all the composed ByteBuf

There are more methods in there, but I think you get the idea. The Netty API is clearly documented so as you use other methods not shown, it'll be easy to understand what they do by referring to the API docs.

Also, because of the nature of a `CompositeBytebuf`, you won't be able to access a backing array. It looks similar to what you saw for the direct ByteBuf and in the following listing.

Listing 5.5 Access data

```
CompositeBuf compBuf = ...;
if (!compBuf.hasArray()) {                                   #1
    int length = compBuf.readableBytes();                    #2
       byte[] array = new byte[length];                      #3
    compBuf.getBytes(array);                                 #4
    YourImpl.method(array, 0, array.length);                 #5

}
```

#1 Check if ByteBuf not backed by array which will be false for a composite buffer
#2 Get amount of readable bytes
#3 Allocate new array with length of readable bytes
#4 Read bytes into array
#5 Call method that takes array, offset, length as parameter

As `CompositeByteBuf` is a sub-type of `ByteBuf` (like Heap and Direct buffers), you're able to operate on the buffer as usual but with the possibility for some extra operations.

You may also like that Netty will optimize read and write operations on the socket whenever possible when using a `CompositeByteBuf`. This means that using gathering and scattering doesn't incur performance penalties when reading or writing to a socket or suffer from the memory leak issues in the JDK's implementation. All of this is done in the core of Netty itself so you don't need to worry about it too much, but it can't hurt to know that some optimization is done under-the-hood.

A class such as the `CompositeByteBuf` doesn't exist when using `ByteBuffer`. This is just one thing that makes the buffer API more feature-rich than the buffer API provided by the JDK as part of the `java.nio` package.

5.3 ByteBuf's byte operations

The `ByteBuf` offers many operations that allow modifying the content or just reading it. You'll learn quickly that it's much like the JDK's `ByteBuffer`, but on steroids, as it offers a much better user experience. While the `ByteBuf` interface offer many methods the next sections wll focus on the most importants.

5.3.1 Random access indexing

Like an ordinary primitive byte array, `ByteBuf` uses zero-based-indexing. This means the index of the first byte is always 0 and the index of the last byte is always capacity – 1. For example, I can iterate all the bytes of a buffer (see the following listing), regardless of its internal implementation.

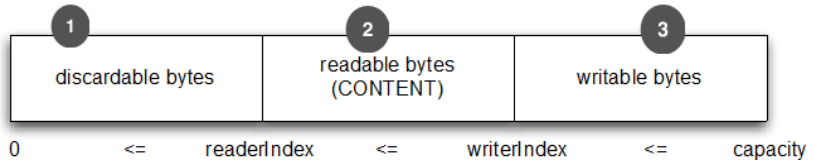
Listing 5.7 Access data

```
ByteBuf buffer = ...;
 for (int i = 0; i < buffer.capacity(); i ++) {
     byte b = buffer.getByte(i);
     System.out.println((char) b);
}
```

Be aware that index pased access will not advance the `readerIndex` or `writerIndex`. You can advance it by hand by call `readerIndex(index)` and `writerIndex(index)` if needed.

5.3.2 Sequential access indexing

`ByteBuf` provides two pointer variables to support sequential read and write operations—`readerIndex` for a read operation and `writerIndex` for a write operation, respectively. Again, this is different from the JDK's `ByteBuffer` that has only one, so you need to `flip()` to switch between read and write mode. Figure 5.3 shows how a buffer is segmented into three areas by the two pointers.



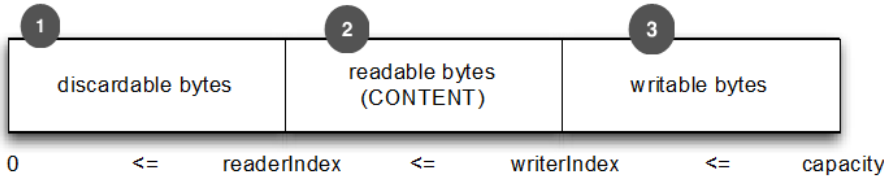
#1 Segment that holds bytes that can be discarded as they were read before
#2 Segment that holds the actual readable content that was not read yet
#3 Segment that contains the left space of the buffer and so to which more bytes can be written

Figure 5.3 Different ByteBuf areas which are part of the ByteBuf.

5.3.3 Discardable bytes

The discardable bytes segment contains the bytes that were already read by a read operation and so may be discarded. Initially, the size of this segment is 0, but its size increases up to the `writerIndex` as read operations are executed. This only includes "read" operations; "get" operations do not move the `readerIndex`. The read bytes can be discarded by calling `discardReadBytes()` to reclaim unused space.

Figure 5.4 shows what the segments of a `ByteBuf` look like before `discardReadBytes()` is called.

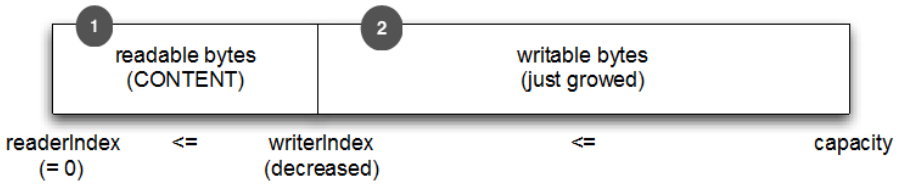


#1 Segment that holds bytes that can be discarded as they was read before
#2 Segment that holds the actual readable content that was not read yet
#3 Segment that contains the left space of the buffer and so to which more bytes can be written

Figure 5.4 Before discardReadBytes is called.

As you can see, the discardable bytes segment contains some space that is ready for reuse. This can be achieved by calling `discardReadBytes()`.

Figure 5.5 shows how the call of `discardReadBytes()` will affect the segments.



#1 Segment that holds the actual readable content that was not read yet. This starts now on index 0
#2 Segment that contains the left space of the buffer and so to which more bytes can be written. This is now bigger as it growed by the space that was hold by the discardable bytes before

Figure 5.5 After discardReadBytes is called

Note that there's no guarantee about the content of writable bytes after calling `discardReadBytes()`. The writable bytes won't be moved in most cases and could even be filled with completely different data depending on the underlying buffer implementation.

Also, you may be tempted to frequently call `discardReadBytes()` to provide the `ByteBuf` with more writable space again. Be aware that `discardReadBytes()` will most likely involve a memory copy as it needs to move the readable bytes (content) to the start of the `ByteBuf`.

Such an operation isn't free and may affect performance, so only use it if you need it and will benefit from it. Thus would be for example if you need to free up memory as soon as possible.

5.3.4 Readable bytes (the actual content)

This segment is where the actual data is stored. Any operation whose name starts with read or skip will get or skip the data at the current `readerIndex` and increase it by the number of read bytes. If the argument of the read operation is also a `ByteBuf` and no destination index is specified, the specified destination buffer's `writerIndex` is increased together.

If there's not enough content left, `IndexOutOfBoundException` is raised. The default value of newly allocated, wrapped, or copied buffer's `readerIndex` is 0.

The following listing shows how to read all readable data.

Listing 5.8 Read data

```
// Iterates the readable bytes of a buffer.
ByteBuf buffer = ...;
while (buffer.readable()) {
    System.out.println(buffer.readByte());
}
```

5.3.5 Writable bytes

This segment is an undefined space which needs to be filled. Any operation whose name starts with write will write the data at the current `writerIndex` and increase it by the number of written bytes. If the argument of the write operation is also a `ByteBuf` and no source index is specified, the specified buffer's `readerIndex` is increased together.

If there's not enough writable bytes left, `IndexOutOfBoundException` is raised. The default value of newly allocated buffer's `writerIndex` is 0.

The following listing shows an example that fills the buffer with random `int` values until it runs out of space. This shows how you would use the `writableBytes()` method to see if there is enough room in the `ByteBuf` left to write the needed data.

Listing 5.9 Write data

```
// Fills the writable bytes of a buffer with random integers.
ByteBuf buffer = ...;
while (buffer.writableBytes() >= 4) {
    buffer.writeInt(random.nextInt());
}
```

5.3.6 Clearing the buffer indexes

You can set both `readerIndex` and `writerIndex` to 0 by calling `clear()`. It doesn't clear the buffer's content (for example, filling with 0) but clears the two pointers. Please note that the semantics of this operation are different from the JDK's `ByteBuffer.clear()`.

Let's look at its functionality. You may recall our ByteBuf with the three different segements, Figure 5.6 shows it again.

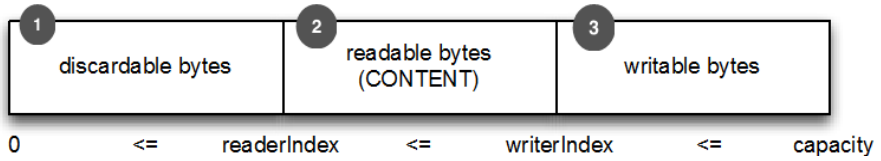
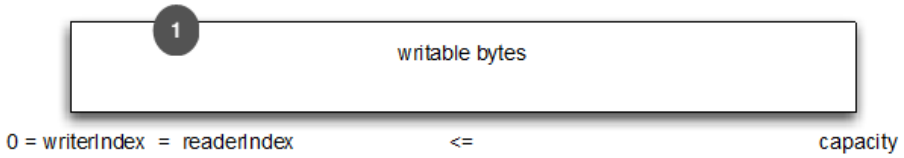


Figure 5.6 Before clear() is called.

As before, it contains three segments. You'll see this change once `clear()` is called. Figure 5.7 shows the `ByteBuf` after `clear()` is used.



#1 Segment is now as big as the capacity of the ByteBuf, so everything is writable

Figure 5.7 After clear() is called

Compared to `discardReadBytes()`, the `clear()` operation is cheap, because it just adjust the indexes (`readerIndex` and `writerIndex`) and doesn't need to copy any memory.

5.3.7 Search operations

Various `indexOf()` methods help you locate an index of a value which meets a certain criteria. Complicated dynamic sequential search can be done with `ByteBufProcessor` implementations as well as simple static single-byte search.

If you're decoding variable length data such as NULL-terminated string, you'll find the `bytesBefore(byte)` method useful. Let's imagine you've written an application, which has to integrate with flash sockets, which uses NULL-terminated content. Using the `bytesBefore()` method, you can easily consume data from Flash without manually readying every byte in the data to check for NULL bytes. Without the `ByteBufProcessor` you would need to do all this work by yourself. Also it is more efficient as it needs less "bound checks" during processing.

5.3.8 Mark and reset

As stated before, there are two marker indexes in every buffer. One is for storing `readerIndex` and the other is for storing `writerIndex`. You can always reposition one of the two indexes by calling a reset method. It works in a similar fashion to the mark and reset methods in an `InputStream` except that there are no `read limits`.

Also, you can move them to an "exact" index by calling `readerIndex(int)` or `writerIndex(int)`. Be aware that trying to set the `readerIndex` or `writerIndex` to an invalid position will cause an `IndexOutOfBoundException`.

5.3.9 Derived buffers

To create a view of an existing buffer, call `duplicate()`, `slice()`, `slice(int, int)`, `readOnly()`, or `order(ByteOrder)`. A derived buffer has an independent `readerIndex`, `writerIndex`, and marker indexes, but it shares other internal data representation the way a NIO `ByteBuffer` does. Because it shares the internal data representation, it's cheap to create and is the preferred way if, for example, you need a "slice" of a `ByteBuf` in an operation.

If a fresh copy of an existing buffer is required, use the `copy()` or `copy(int, int)` method instead. The following listing shows how to work with a slice of a `ByteBuf`.

Listing 5.10 Slice a ByteBuf

```
Charset utf8 = Charset.forName("UTF-8");
ByteBuf buf = Unpooled.copiedBuffer("Netty in Action rocks!", utf8);     #1

ByteBuf sliced = buf.slice(0, 14);                                       #2
System.out.println(sliced.toString(utf8);                                #3

buf.setByte(0, (byte) 'J');                                              #4
assert buf.get(0) == sliced.get(0);                                      #5
```

#1 Create ByteBuf which holds bytes for given string
#2 Create new slice of ByteBuf which starts at index 0 and ends at index 14
#3 Contains "Netty in Action"
#4 Update byte on index 0
#5 Won't fail as both ByteBuf share the same content and so modifications to one of them are visible on the other too

Now let's look at how to create a copy of a `ByteBuf` and how that differs from a slice. The following listing shows how to work with a copy of a `ByteBuf`.

Listing 5.11 Copying a ByteBuf

```
Charset utf8 = Charset.forName("UTF-8");
ByteBuf buf = Unpooled.copiedBuffer("Netty in Action rocks!", utf8);     #1

ByteBuf copy = buf.copy(0, 14);                                          #2
System.out.println(copy.toString(utf8);                                  #3

buf.setByte(0, (byte) 'J');                                              #4
assert buf.get(0) != copy.get(0);                                        #5
```

#1 Create ByteBuf which holds bytes for given string
#2 Create copy of ByteBuf which starts at index 0 and ends at index 14
#3 Contains "Netty in Action"
#4 Update byte on index 0
#5 Won't fail as both ByteBuf does not share the same content and so modifications to one of them are not shared

The API is the same, but how a modification affects the derived `ByteBuf` is different.

Use a slice whenever possible, and use copy only as needed. Creating a copy of the `ByteBuf` is more expensive because it needs to do a memory copy.

5.3.10 Read/write operations

There are two main types of read/write operations:

- Index based get/set operations that set or get bytes on a given index.
- Read/write operations that either read bytes from the current index and increase them or write to the current index and increase it.

Let's review the relative operations first; I'll mention only the most popular for now. For a complete overview, refer to the API docs.

Table 5.1 shows the most interesting get operations that are used to access data on a given index.

Table 5.1 Relative get operations

Name	Description
`getBoolean(int)`	Return the Boolean value on the given index.
`getByte(int)` `getUnsignedByte(int)`	Return the (unsigned) byte value on the given index.
`getMedium(int)` `getUnsignedMedium(int)`	Return the (unsigned) medium value on the given index.
`getInt(int)` `getUnsignedInt(int)`	Return the (unsigned) int value on the given index.
`getLong(int)` `getUnsignedLong(int)`	Return the (unsigned) int value on the given index.
`getShort(int)` `getUnsignedShort(int)`	Return the (unsigned) int value on the given index.
`getBytes(int, ...)`	Return the (unsigned) int value on the given index.

For most of these get operations there is a similar set operation. Table 5.2 shows these.

Table 5.2 Relative set operations

Name	Description
`setBoolean(int, boolean)`	Set the Boolean value on the given index.
`setByte(int, int)`	Set byte value on the given index.
`setMedium(int, int)`	Set the medium value on the given index.

`setInt(int, int)`	Set the int value on the given index.
`setLong(int, long)`	Set the long value on the given index.
`setShort(int, int)`	Set the short value on the given index.
`setBytes(int,...)`	Transfer the bytes to the given index to/from given resources.

You may have noticed that there are no unsigned versions of these methods. This is because there isn't a notion of it when setting values. Now that you know there are relative operations, let's see them in practice, as shown in the following listing.

Listing 5.12 Relative operations on the ByteBuf

```
Charset utf8 = Charset.forName("UTF-8");
ByteBuf buf = Unpooled.copiedBuffer("Netty in Action rocks!", utf8);     #1
System.out.println((char) buf.getByte(0));                               #2

//                                                                       #3
int readerIndex = buf.readerIndex();
int writerIndex = buf.writerIndex();

buf.setByte(0, (byte) 'B');                                              #4
System.out.println((char) buf.getByte(0));                               #5

//                                                                       #6
assert readerIndex = buf.readerIndex();
assert writerIndex = buf.writerIndex();
```

#1 Create a new ByteBuf which holds the bytes for the given String
#2 Prints out the first char which is "N"
#3 Store the current readerIndex and writerIndex
#4 Update the byte on index 0 with the char 'B'
#5 Prints out the first char which is "B" now as I updated it before
#6 Check that the readerIndex and writerIndex did not change which is true as relative operations never modify the indexes.

In addition to the relative operations, there are also operations that act on the current `readerIndex` or `writerIndex`. These operations are the ones that you mostly use to read from the `ByteBuf` as if it were a stream. The same is true for the write operations that are used to "append" to a `ByteBuf`.

Table 5.3 shows the most-used read operations.

Table 5.3 Read operations

Name	Description
`readBoolean()`	Reads the Boolean value on the current `readerIndex` and increases the `readerIndex` by 1.
`readByte()`	Reads the (unsigned) byte value on the current `readerIndex` and

`readUnsignedByte()`	increases the `readerIndex` by 1.
`readMedium()` `readUnsignedMedium()`	Reads the (unsigned) medium value on the current `readerIndex` and increases the `readerIndex` by 3.
`readInt()` `readUnsignedInt()`	Reads the (unsigned) int value on the current `readerIndex` and increases the `readerIndex` by 4.
`readLong()` `readUnsignedLong()`	Reads the (unsigned) int value on the current `readerIndex` and increases the `readerIndex` by 8.
`readShort()` `readUnsignedShort()`	Reads the (unsigned) int value on the current `readerIndex` and increases the `readerIndex` by 2.
`readBytes(int,int, ...)`	Reads the value on the current `readerIndex` for the given length into the given object. Also increases the `readerIndex` by the length.

There is a write method for almost every read method. Table 5.4 shows these.

Table 5.4 Write operations

Name	Description
`writeBoolean(boolean)`	Writes the Boolean value on the current `writerIndex` and increases the `writerIndex` by 1.
`writeByte(int)`	Writes the byte value on the current `writerIndex` and increases the `writerIndex` by 1.
`writeMedium(int)`	Writes the medium value on the current `writerIndex` and increases the `writerIndex` by 3.
`writeInt(int)`	Writes the int value on the current `writerIndex` and increases the `writerIndex` by 4.
`writeLong(long)`	Writes the long value on the current `writerIndex` and increases the `writerIndex` by 8.
`writeShort(int)`	Writes the short value on the current `writerIndex` and increases the `writerIndex` by 2.
`writeBytes(int,…)`	Transfers the bytes on the current `writerIndex` from given resources.

Let's see them in practice, as shown in the following listing.

Listing 5.13 Read/write operations on the ByteBuf

```
Charset utf8 = Charset.forName("UTF-8");
ByteBuf buf = Unpooled.copiedBuffer("Netty in Action rocks!", utf8);     #1
System.out.println((char) buf.readByte());                               #2

//
int readerIndex = buf.readerIndex();                                     #3
int writerIndex = buf.writerIndex();                                     #4

buf.writeByte( (byte) '?');                                              #5
```

```
//                                                                    #6
assert readerIndex = buf.readerIndex();
assert writerIndex != buf.writerIndex();
```

#1 Create ByteBuf which holds bytes for given string
#2 Prints first char "N"
#3 Store current readerIndex and writerIndex
#4 Update byte on index 0 with char "B"
#5 Prints first char "B" that I updated
#6 Check readerIndex and writerIndex didn't change

On #6, note that relative operations never modify the indexes. With a broad overview of the methods that write or read values from a `ByteBuf`, under your belt, let's take a look at a few others.

5.3.11 Other useful operations

There are other useful operations that I haven't mentioned yet, but they often come in handy, depending on your use case. Table 5.5 gives an overview of them and explains what they do.

Table 5.5 Other useful operations

Name	Description
`isReadable()`	Returns true if at least one byte can be read.
`isWritable()`	Returns true if at least one byte can be written.
`readableBytes()`	Returns the number of bytes that can be read.
`writablesBytes()`	Returns the number of bytes that can be written.
`capacity()`	Returns the number of bytes that the `ByteBuf` can hold. After this it will try to expand again until `maxCapacity()` is reached.
`maxCapacity()`	Returns the maximal number of bytes the `ByteBuf` can hold.
`hasArray()`	Returns true if the `ByteBuf` is backed by a byte array.
`array()`	Returns the byte array if the `ByteBuf` is backed by a byte array, otherwise throws an `UnsupportedOperationException`.

During the last sections you learned how you can operate on `ByteBuf` and make the best use of it. This included the concept of `ByteBuf` itself and the different provided types. You learned how to write into a `ByteBuf` and how to read data out of it. Beside this you received some

introduction about how the `ByteBuf` uses `readerIndex` and `writerIndex` to allow navigation within the `ByteBuf` itself.

In the next section you will learn about `ByteBufHolder` that is also part of the Buffer API of Netty and can be used by messages that use `ByteBuf` to hold it's content or payload.

5.4 ByteBufHolder

Often, you have an object that needs to hold bytes as the actual payload and also other properties. For example, an object that represents an HTTP response is exactly like this. You have properties, such as the status code, cookies, and so on, but also the actual content/payload that's represented as bytes.

As this situation is common, Netty provides an extra "abstraction" for it called `ByteBufHolder`. The good thing is that this enables Netty to also make use of advanced features such as buffer pooling as the `ByteBuf` that holds the data can also get allocated out of a pool while still enabling Netty to release it automatically.

`ByteBufHolder`, in fact, has only a handful of methods that allows access to the data that it holds and also makes use of reference counting. Table 5.7 shows the methods that it provides (ignoring the ones that are declared in its super-type `ReferenceCounted`).

Table 5.7 ByteBufHolder operations

Name	Description
`data()`	Return the `ByteBuf` that holds the data.
`copy()`	Make a copy of the `ByteBufHolder` that does not share its data (so the data is also copied).

If you want to implement a "message object" that stores its "payload/data" in a `ByteBuf`, it's always a good idea to make use of `ByteBufHolder`.

5.5 ByteBuf allocation

There are different ways to allocate a `ByteBuf` and make use of it when using Netty. The next sections will give you some insight how you allocate `ByteBuf's` and so be able to make use of them.

5.5.1 ByteBufAllocator—Allocate ByteBuf when needed

As mentioned before, Netty supports pooling for the various `ByteBuf` implementations, this greatly eliminate the overhead of allocating and deallocating memory. To make this possible it provides an abstraction called `ByteBufAllocator`. As the name implies it's responsible for allocating `ByteBuf` instances of the previously explained types. Whether these are pooled or not is specific to the implementation but doesn't change the way you operate on it.

Let's first look at the various operations `ByteBufAllocator` provides. Table 5.8 gives a brief overview here.

Table 5.8 ByteBufAllocator methods

Name	Description

`buffer()`
`buffer(int);`
`buffer(int, int);`
	Return a `ByteBuf` that may be of type heap or direct depend on the implementation.

`heapBuffer()`
`heapBuffer(int)`
`heapBuffer(int, int)`
	Return a `ByteBuf` of type heap.

`directBuffer()`
`directBuffer(int)`
`directBuffer(int, int)`
	Return a `ByteBuf` of type direct.

`compositeBuffer()` `compositeBuffer(int);`	Return a `CompositeByteBuf` that may expand internally if needed using a heap or direct buffer.

`heapCompositeBuffer ()`
`heapCompositeBuffer(int);`
`directCompositeBuffer ()`
`directCompositeBuffer(int);`	

`ioBuffer()`	Return a `ByteBuf` that will be used for I/O operations which is reading from the socket.

As you can see, these methods take some extra arguments which allow the user to specify the initial capacity of the `ByteBuf` and the maximum capacity. You may remember that `ByteBuf` is allowed to expand as needed. This is true until the maximum capacity is reached.

Getting a reference to the `ByteBufAllocator` is easy. You can obtain it either through the `Channel` (in theory, each `Channel` can have a different `ByteBufAllocator`) or through the `ChannelHandlerContext` that is bound to the `ChannelHandler` from which you execute your code. More on `ChannelHandler` and `ChannelHandlerContext` can be found in chapter 6.

The following listing 5.14 shows both of the ways of obtaining a byte buffer allocator.

Listing 5.14 Obtain `ByteBufAllocator` reference

```
Channel channel = ...;
ByteBufAllocator allocator = channel.alloc();                    #1
....

ChannelHandlerContext ctx = ...;
ByteBufAllocator allocator2 = ctx.alloc();                       #2
...
```

#1 Get ByteBufAllocator from a channel
#2 Get ByteBufAllocator from a ChannelHandlerContext

Netty comes with two different implementations of `ByteBufAllocator`. One implementation pools `ByteBuf` instances to minimize the allocation/de-allocation costs and keeps memory

fragmentation to a minimum. How exactly these are implemented is outside the scope of this book, but let me note it's based on the "jemalloc" paper and so uses the same algorithm as many operating systems use to efficiently allocate memory.

The other implementation does not pool `ByteBuf` instances at all and returns a new instance every time. Netty uses the `PooledByteBufAllocator` (which is the pooled implementation of `ByteBufAllocator`) by default but this can be changed easily by either changing it through the `ChannelConfig` or specifying a different one when bootstrapping the server. More details can be found in chapter 9 "Bootstrap your application".

5.5.2 Unpooled—Buffer creation made easy

There may be situations where you can't access the previously explained `ByteBuf` because you don't have a reference to the `ByteBufAllocator`. For this use case Netty provides a utility class called `Unpooled`. This class contains static helper methods to create unpooled `ByteBuf` instances.

Let's have a quick peek at the provided methods. Explaining all of them would be too much for this section, so please refer to the API docs for an overview on all of them.

Table 5.9 shows the most used methods of `Unpooled`.

Table 5.9 Unpooled helper class

Name	Description

`buffer()`
`buffer(int)`
`buffer(int, int)`
	Returns an unpooled `ByteBuf` of type `heap`.

`directBuffer()`
`directBuffer(int)`
`directBuffer(int, int)`
	Returns an unpooled `ByteBuf` of type `direct`.

`wrappedBuffer()`	Returns a `ByteBuf`, which wraps the given data.
`copiedBuffer()`	Returns a `ByteBuf`, which copies the given data and uses it .

This `Unpooled` class also makes it easier to use the Netty's buffer API outside Netty, which you may find useful for a project that could benefit from a high-performing extensible buffer API but that does not need other parts of Netty.

5.5.3 ByteBufUtil—Small but useful

Another useful class is the `ByteBufUtil` class. This class offers static helper methods, which are helpful when operating on a `ByteBuf`. One of the main reasons to have these outside the `Unpooled` class mentioned before is that these methods are generic and aren't dependent on a `ByteBuf` being pooled or not.

Perhaps the most valuable is the `hexdump()` method which is provided as a static method like the others in this class. What it does is print the contents of the `ByteBuf` in a hex presentation. This can be useful in a variety of situations. One is to "log" the contents of the `ByteBuf` for debugging purposes. The hex value can easily be converted back to the actual byte representation. You may wonder why not print the bytes directly. The problem here is that this may result in some difficult-to-read log entries. A hex string is much more user friendly.

In addition to this the class also provides methods to check equality of `ByteBuf` implementations and others that may be of use when you start to implement your own `ByteBuf`.

5.6 Reference counting

Netty 4 introduced reference-counting which is used for `ByteBuf` and `ByteBufHolder` (both implement the `ReferenceCounted` interface). Using reference-counting allows to eliminate the overhead of allocating new memory as `ByteBuf` instances can be pooled in the `ByteBufAllocator` implementation.

There isn't much to say about reference-counting as it is slightly simple. This section will give you all the informations that you need to make proper use of it.

Every Object that implements `ReferenceCounted` should start with a reference count of 1. Thus means one reference to this Object is to be considered active.

Listing 5.14 Reference count

```
Channel channel = ...;
ByteBufAllocator allocator = channel.alloc();                    #1
....

ByteBuf buffer = allocator.directBuffer();                       #2
assert buffer.refCnt() == 1;                                     #3
...
```

#1 Get ByteBufAllocator from a channel
#2 Allocate a ByteBuf from the ByteBufAllocator
#3 Check for expected reference count of 1.

As long as the reference count is > 0 it is guaranteered that the reference counted object is not released. What releasing means in the context is specific to the implementation, but in general it is disallowed to use this object anymore.

Listing 5.15 Release reference counted object

```
ByteBuf buffer = ...;
boolean released = buffer.release();                             #1
...
```

#1 Try to release the buffer which will decrement the reference count by 1. Once the reference count reaches 0 it is released and true is returned.

Trying to access a released reference counted object will result in a `IllegalReferenceCountException`.

Who is responsible for release ?

This actually quite simple... The party who access it last is responsible for release it. How this fits in with Netty's concept of `ChannelHandler` and `ChannelPipeline` is explained in Chapter 6.

5.7 Summary

In this chapter you learned about the data containers that are used inside of Netty and why these are more flexible in their usage than what you would find in the JDK.

The chapter also highlighted how you can use the different data containers and what operations are possible, and explained what operations are more "expensive" then others and what methods to use for some use cases.

In the next chapter I'll focus on `ChannelHandler`, which lets you write your own logic to process data. This `ChannelHandler` will make heavy use of the data containers described in this chapter. This will help you better understand the use cases and also show why these are important.

6

ChannelHandler and ChannelPipeline

This chapter covers

- `ChannelPipeline`
- `ChannelHandlerContext`
- `ChannelHandler`
- `Inbound` versus `outbound`

Accepting connections or creating them is only one part of your application. While it's true that these tasks are important, there's another aspect that's often more complex and needs more code to write. This is the processing of incoming data and outgoing data.

Netty provides you with a powerful approach to achieve exactly this. It allows the user to hook in `ChannelHandler` implementations that process the data. What makes `ChannelHandler` even more powerful is that you can "chain" `ChannelHandlers` and so each `ChannelHandler` implementation can fulfill small tasks. This helps you write clean and reusable implementations.

But processing data is only one thing you can do with `ChannelHandler`. You can also suppress I/O operations, which would be for example a write request (you wil see more examples later in this chapter). All of this can be done on-the-fly which makes it even more powerful. Beside this there are many different use cases, you will learn more about these in the later chapters.

All these `ChannelHandlers` can be chained together in the so called `ChannelPipeline`, which acts like a linked-lised of `ChannelHandlers` itself.

In this chapter you will learn everything you need about so called `ChannelHandlers` and how they can be used within your application to build a powerful processing logic. This includes also details about the `ChannelPipeline` and the `ChannelHandlerContext`.

6.1 ChannelPipeline

A `ChannelPipeline` is a list of `ChannelHandler` instances that handle or intercept inbound and outbound events of a `Channel`. `ChannelPipeline` offers an advanced form of the interception filter pattern[8], giving a user full control over how an event is handled and how the `ChannelHandlers` in the `ChannelPipeline` interact with each other.

For each new `Channel`, a new `ChannelPipeline` is created and attached to the `Channel`. Once attached, the coupling between the `Channel` and the `ChannelPipeline` is permanent; the `Channel` cannot attach another `ChannelPipeline` to it or detach the current `ChannelPipeline` from it. All of this is handled for you as part of the lifeycle; you don't need to take care of this.

[8] http://www.oracle.com/technetwork/java/interceptingfilter-142169.html

Figure 6.1 describes how `ChannelHandlers` in a `ChannelPipeline` typically process I/O. An I/O event can be handled by either a `ChannelInboundHandler` or a `ChannelOutboundHandler` and be forwarded to the closest handler by calling either one of the methods defined in the `ChannelHandlerContext` interface for inbound and outbound I/O.

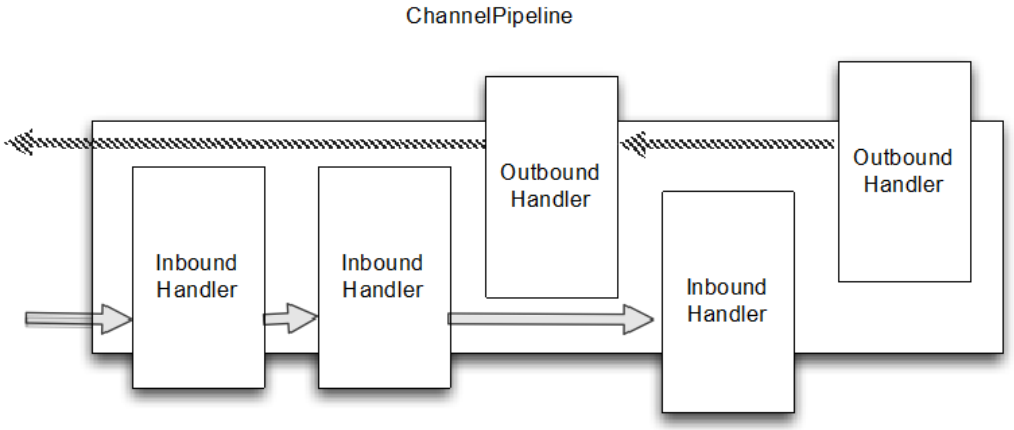


Figure 6.1 ChannelPipeline and how events flow through the handlers

As shown in figure 6.1 a `ChannelPipeline` is mainly a list of `ChannelHandlers`, but it also provide various methods to trigger events through the `ChannelPipeline` itself. If an inbound I/O event is triggered it's passed from the beginning to the end of the `ChannelPipeline`. For outbound I/O events it begins at the end of the `ChannelPipeline` and process to the start. The `ChannelPipeline` itself knows if a `ChannelHandler` can handle the event by checking its type. If it can't handle it, it skips the `ChannelHandler` and forward it to the next `ChannelHandler`.

6.1.1 Modify the ChannelPipeline

Modifications on the `ChannelPipeline` can be done on-the-fly, which means you can add/remove/replace `ChannelHandler` even from within another `ChannelHandler` or have it remove itself. This allows writing flexible logic, such as multiplexer, but we willl go into more detail later in this chapter.

For now, let's look at how you can modify a `ChannelPipeline` as this is one of the most important things to do with the ChannelPipeline. This is because you will usually add `ChannelHandlers` tot he `ChannelPipeline` to fulfill the work later.

Table 6.1 Methods to modify a ChannelPipeline

Name	Description
`addFirst(…)` `addBefore(…)` `addAfter(…)` `addLast(…)`	Add a `ChannelHandler` to the `ChannelPipeline`.
`remove(…)`	Remove a `ChannelHandler` from the `ChannelPipeline`.
`replace(…)`	Replace a `ChannelHandler` in the `ChannelPipeline` with another `ChannelHandler`.

The following listing shows how you can use these methods to modify the `ChannelPipeline`.

Listing 6.1 Modify the `ChannelPipeline`

```
ChannelPipeline pipeline = ..;
FirstHandler firstHandler = new FirstHandler();                    #1
pipeline.addLast("handler1", firstHandler);                        #2
pipeline.addFirst("handler2", new SecondHandler());                #3
pipeline.addLast("handler3", new ThirdHandler());                  #4

pipeline.remove("handler3");                                       #5
pipeline.remove(firstHandler);                                     #6

pipeline.replace("handler2", "handler4", new FourthHandler());     #7
```

#1 Create instance of FirstHandler
#2 Add FirstHandler to ChannelPipeline
#3 Add SecondHandler instance to ChannelPipeline using first position. This means it will be before the already existing FirstHandler
#4 Add ThirdHandler to ChannelPipeline on the last position
#5 Remove ThirdHandler by using name it was added with
#6 Remove FirstHandler by using reference to instance
#7 Replace SecondHandler which was added with handler2 as name by FourthHandler and add it with name handler4

As you can see, modifying the `ChannelPipeline` is easy and allows you to add, remove, replace `ChannelHandler` on demand.

ChannelHandler execution and blocking

Normally each `ChannelHandler` that is added to the `ChannelPipeline` will process the event that is passed through it in the IO-Thread, which means you MUST NOT block as otherwise you block the IO-Thread and so affect the overall handling of IO.

Sometimes it's needed to block as you may need to use legacy API's which only offer a blocking API. For example this is true for JDBC. For exactly this use-case Netty allows to pass an `EventExecutorGroup` to each of the `ChannelPipeline.add*` methods. If a custom `EventExecutorGroup` is passed in the event it will be handled by one of the `EventExecutor` contained in this `EventExecutorGroup` and so „moved". A default implementation which is called `DefaultEventExecutorGroup` comes as part of Netty.

In addition to the operations that allow you to modify the `ChannelPipeline` there are also operations that let you access the `ChannelHandler` implementations that were added as well as check if a specific `ChannelHandler` is present in the `ChannelPipeline`.

Table 6.2 Get operations on ChannelPipeline

Name	Description
`get(…)`	There are a few `get(…)` operations provided by the `ChannelPipeline`. These allow you to retrieve either the `ChannelHandler` or the `ChannelHandlerContext`, which was created and assigned to the `ChannelHandler`.
`context(…)`	Return the `ChannelHandlerContext bound to the ChannelHandler`.
`names()` `iterator()`	Return the names or a reference to all the added `ChannelHander` of the `ChannelPipeline`.

6.1.2 Fire events via the ChannelPipeline

As `ChannelPipeline` exposes additional methods for invoking inbound and outbound operations.

Table 6.3 lists all inbound operations that are exposed, as defined in the `ChannelPipeline` interface.

Table 6.3 Inbound operations on ChannelPipeline

Name	Description
`fireChannelRegistered()`	This results in having the `channelRegistered(ChannelHandlerContext)` method called of the next `ChannelInboundHandler` in the `ChannelPipeline`.
`fireChannelUnregistered()`	This results in having the `channelUnregistered(ChannelHandlerContext)` method called of the next `ChannelInboundHandler` in the `ChannelPipeline`.
`fireChannelActive()`	This results in having the `channelActive(ChannelHandlerContext)` method called of the next `ChannelInboundHandler` in the `ChannelPipeline`.
`fireChannelInactive()`	This results in having the `channelInactive(ChannelHandlerContext)` method called of the next `ChannelInboundHandler` in the `ChannelPipeline`.
`fireExceptionCaught(…)`	This results in having the `exceptionCaught(ChannelHandlerContext, Throwable)` method called of the next `ChannelHandler` in the `ChannelPipeline`.
`fireUserEventTriggered(…)`	This results in having the `userEventTriggered(ChannelHandlerContext, Object)` method call the next `ChannelInboundHandler` contained in the `ChannelPipeline`.
`fireChannelRead(….)`	This results in having the `channelRead(ChannelHandlerContext, Object msg)` method called of the next `ChannelInboundHandler` in the `ChannelPipeline`.
`fireChannelReadComplete()`	This results in having the `channelReadComplete(ChannelHandlerContext)` method called of the next `ChannelStateHandler` in the `ChannelPipeline`.

These operations are mainly useful for notifying the `ChannelInboundHandlers` in the `ChannelPipeline`. These `ChannelInboundHandlers` can then intercept these events and act on them. Like for example transform data or log the events and so on.

But handling inbound events is only half of the story. You also need to trigger and handle outbound events, which will cause some action on the underlying socket.

Table 6.4 lists all outbound operations that are exposed via `ChannelPipeline`.

Table 6.4 Outbound operations on ChannelPipeline

Method name	Description
`bind(…)`	Requests to bind the `Channel` to a local address. This will call the `bind(ChannelHandlerContext, SocketAddress, ChannelPromise)` method of the next `ChannelOutboundHandler` in the `ChannelPipeline`.
`connect(…)`	Requests to connects the `Channel` to a remote address. This will call the `connect(ChannelHandlerContext, SocketAddress, ChannelPromise)` method of the next `ChannelOutboundHandler` in the `ChannelPipeline`.
`disconnect(…)`	Requests to disconnect the `Channel`. This will call the `disconnect(ChannelHandlerContext, ChannelPromise)` method of the next `ChannelOutboundHandler` in the `ChannelPipeline`.
`close(…)`	Requests to close the `Channel`. This will call the `close(ChannelHandlerContext, ChannelPromise)` method of the next `ChannelOutboundHandler` in the `ChannelPipeline`.
`deregister(…)`	Requests to deregister the `Channel` its EventLoop. This will call the `deregister(ChannelHandlerContext, ChannelPromise)` method of the next `ChannelOutboundHandler` in the `ChannelPipeline`.
`flush(…)`	Requests to flush all pending writes of the `Channel`. This will call the `flush(ChannelHandlerContext)` method of the next `ChannelOutboundHandler` in the `ChannelPipeline`.
`write()`	Requests to write the given message to the `Channel`. This will call the `write(ChannelHandlerContext, Object msg, ChannelPromise)` method of the next `ChannelOutboundHandler` in the `ChannelPipeline`. Be aware this will not write the message to the underlying Socket, but only queue it. For have it written to the actual Socket yet need to call `flush(..)` or use `writeAndFlush(…)`.
`writeAndFlush(…)`	Shortcut for calling `write(…)` and `flush(…)`.
`read()`	Requests to read more data from the `Channel`. This will call the `read(ChannelHandlerContext)` method of the next `ChannelOutboundHandler` in the `ChannelPipeline`.

In this section you learnt about the `ChannelPipeline` and that it is used to hold the different `ChannelHandlers` for a Channel. You learnt about how you can modify it on the fly and so add and remove `ChannelHandlers` when the need arise. Beside this you also learned about how you can "trigger" different evetns on the `ChannelPipeline` and so have them flow through all the `ChannelHandlers` in the `ChannelPipeline`.

In the next section we will look at the `ChannelHandlerContext` which acts as „binding" between the `ChannelPipeline` and the `ChannelHandler` itself.

6.2 ChannelHandlerContext

Each time a `ChannelHandler` is added to a `ChannelPipeline`, a new `ChannelHandlerContext` is created and assigned. The `ChannelHandlerContext` allows the `ChannelHandler` to interact with other `ChannelHandler` implementations, which are part of the same `ChannelPipeline`.

The `ChannelHandlerContext` never changes for an added `ChannelHandler` so it's safe to get cached.

The `ChannelHandlerContext` has many methods that are also present on the `Channel` or the `ChannelPipeline` itself. The difference is that if you call them on the `Channel` or `ChannelPipeline` they always flow through the complete `ChannelPipeline`. In contrast, if you call a method on the `ChannelHandlerContext`, it starts at the current position and notify the closes `ChannelHandler` in the `ChannelPipeline` that can handle the event.

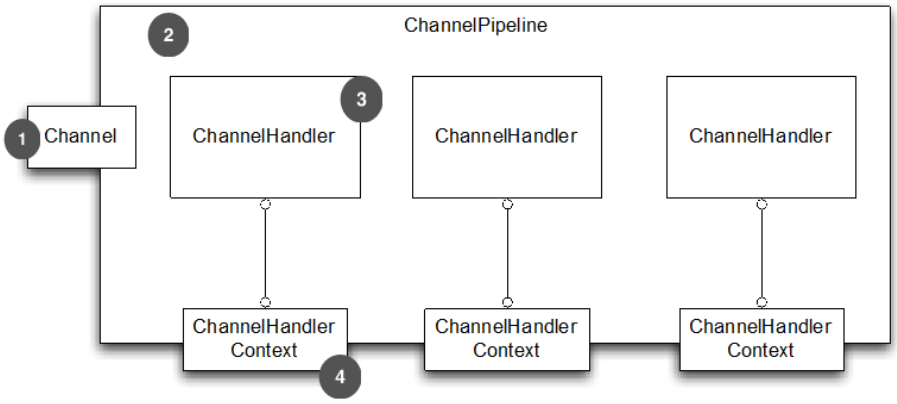
Normally you would use the methods on ChannelHandlerContext when trigger events from inside your ChannelHandler as this will give you the shortest event flow and so the most out of performance.

6.2.1 Notify the next ChannelHandler

You can notify the closest handler in the same `ChannelPipline` by calling one of the various methods listed in `ChannelHandlerContext`. This is one of the core principials of the Interceptor pattern. If you are done with the event and want to have it handled further you just forward it to the next `ChannelHandler` in the `ChannelPipeline`.

Where the notification starts depends on how you set up the notification.

Figure 6.2 shows how the `ChannelHandlerContext` belongs to the `ChannelHandler` and binds it to the `ChannelPipeline`.



#1 Channel that ChannelPipeline is bound to
#2 ChannelPipeline bound to channel and holds added ChannelHandler instances
#3 ChannelHandler that is part of ChannelPipeline
#4 ChannelHandlerContext created while adding ChannelHandler to ChannelPipeline

Figure 6.2 ChannelPipeline, ChannelHandlerContext and Channel

Now if you'd like to have the event flow through the whole `ChannelPipeline`, there are two different ways of doing so:

- Invoke methods on the `Channel`.
- Invoke methods on the `ChannelPipeline`.

Both methods let the event flow through the whole `ChannelPipeline`. Whether it begins at the start or at the end mainly depends on the nature of the event. If it's an inbound event, it begins at the start, and if it's an outbound event it begins at the end.

The following listing shows how you can pass a write event through the `ChannelPipeline` starting at the end (as it's an outbound operation).

Listing 6.2 Events via Channel

```
ChannelHandlerContext ctx = ..;
Channel channel = ctx.channel();                                    #A
channel.write(Unpooled.copiedBuffer("Netty in Action",
        CharsetUtil.UTF_8));                                        #B
```

#A Get reference of channel that belongs to ChannelHandlerContext
#B Write buffer via channel

The message flows through the entire `ChannelPipeline`. As mentioned before, you can also do the same via the `ChannelPipeline`. Listing 6.3 shows an example.

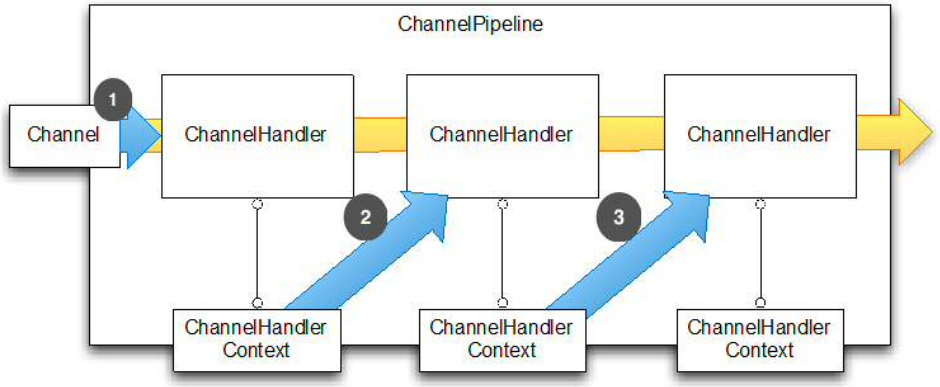
Listing 6.3 Events via `ChannelPipeline`

```
ChannelHandlerContext ctx = ..;
ChannelPipeline pipeline = ctx.pipeline();                          #A
pipeline.write(Unpooled.copiedBuffer("Netty in Action",
        CharsetUtil.UTF_8));                                        #B
```

#A Get reference of ChannelPipeline that belongs to ChannelHandlerContext
#B Write buffer via ChannelPipeline

The message flows through the entire `ChannelPipeline`. Each operation (listings 6.2 and 6.3) is equal in terms of event flow. You should also notice that the `Channel` and the `ChannelPipeline` are accessible via the `ChannelHandlerContext`.

Figure 6.3 shows the flow of the event of a notification that was triggered by either the `Channel` or the `ChannelPipeline`.



#1 Event passed to first ChannelHandler in ChannelPipeline
#2 ChannelHandler passes event to next in ChannelPipeline using assigned ChannelHandlercontext
#3 ChannelHandler passes event to next in ChannelPipeline using assigned ChannelHandlercontext

Figure 6.3 Notify via the Channel or the ChannelPipeline

There may be situations where you want to start a specific position of the `ChannelPipeline` and don't want to have it flow through the whole `ChannelPipeline`, such as:

- To save the overhead of passing the event through extra `ChannelHandlers` that are not interested in it.
- To exclude some `ChannelHandlers`.

In this case you can notify using the `ChannelHandlerContext` of the `ChannelHandler` that's your preferred starting point. Be aware that it executes the next `ChannelHandler` to the used `ChannelHandlerContext` and not the one that belongs to the used `ChannelHandlerContext`.

Listing 6.4 shows how this can be done by directly using `ChannelHandlerContext` for the operation.

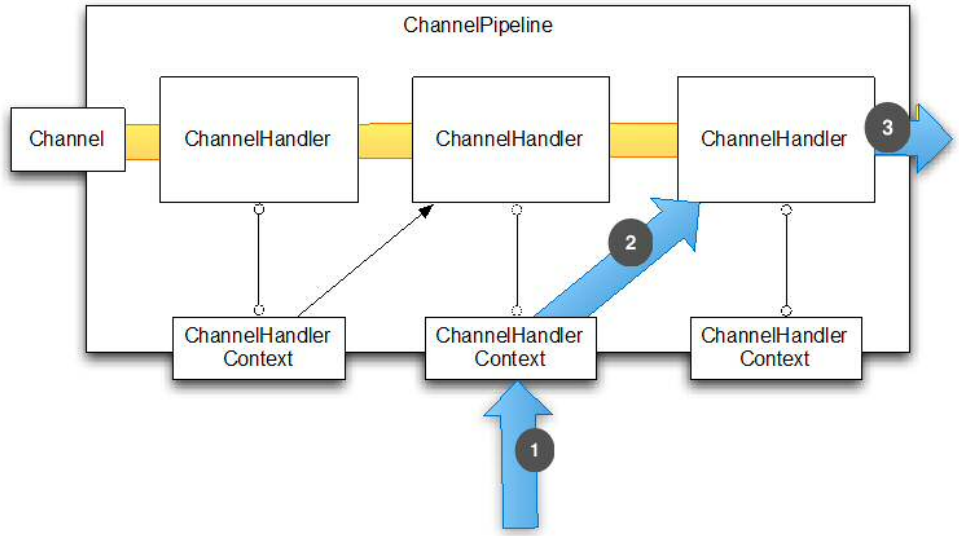
Listing 6.4 Events via ChannelPipeline

```
ChannelHandlerContext ctx = ..;                                    #1
ctx.write(Unpooled.copiedBuffer("Netty in Action", CharsetUtil.UTF_8));  #2
```

#1 Get reference of ChannelHandlerContext
#2 Write buffer via ChannelHandlerContext.

The message starts to flow through the `ChannelPipeline` by the next `ChannelHandler` to the `ChanneHandlerContext` In that case the event flow would start the next `ChannelHandler` to the used `ChannelHandlerContext`.

Figure 6.4 shows the event flow.



#1 Event passed to specific ChannelHandler usingChannelHandlerContext
#2 Event gets passed
#3 Move out of ChannelPipeline as no ChannelHandlers remain

Figure 6.3 Event flow for operations triggered via the ChannelHandlerContext

As you can see, it now starts at a specific `ChannelHandlerContext` and skips all `ChannelHandlers` before it. Using the `ChannelHandlerContext` for operations is a common pattern and the most used if you call operations from within a `ChannelHandler` implementation.

6.2.2 Acessing the ChannelPipeline

You can access the `ChannelPipeline` your `ChannelHandler` belongs to by calling the `pipeline()` method. A nontrivial application could insert, remove, or replace `ChannelHandler`s in the `ChannelPipeline` dynamically in runtime. This for example could be useful if you need to add a `ChannelHandler` later after switching to a different protocol.

> **NOTE** You can keep the `ChannelHandlerContext` for later use, such as triggering an event outside the `ChannelHandler` methods, even from a different Thread.

The following listing shows how you can store the `ChannelHandlerContext` for later use and then use it even from another thread.

Listing 6.5 `ChannelHandlerContext` usage

```
public class WriteHandler extends ChannelHandlerAdapter {

    private ChannelHandlerContext ctx;

    @Override
    public void handlerAdded(ChannelHandlerContext ctx) {
        this.ctx = ctx;                                          #A
    }

    public void send(String msg) {                               #B
        ctx.write(msg);
    }
}
```

#A Store reference to ChannelHandlerContext for later use
#B Send message using previously stored ChannelHandlerContext

Please note that a `ChannelHandler` instance can be added to more than one `ChannelPipeline` if it's annotated with the `@Sharable`.

If you try to add a `ChannelHandler` to more then one `ChannelPipeline` that is not annotated with `@Sharable` an exception is thrown. Be aware that once you annotate a `ChannelHandler` with `@Sharable` it must be safe to use from different threads and also be safe to use with different channels (connections) at the same time. Let's look at how it should be used. The following listing shows the correct use of the `@Sharable` annotation.

Listing 6.6 Valid usage of `@Sharable`

```
@Sharable                                                        #A
public class SharableHandler extends ChannelInboundHandlerAdapter {

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        System.out.println("Channel read message " + msg);       #B
        ctx.fireChannelRead(msg);
    }

}
```

#A Annotate with @Sharable
#B Log method call and forward to next ChannelHandler

The use of `@Sharable` is valid here, as the `ChannelHandler` doesn't use any fields to store data and so is "stateless".

There are also bad uses of `@Sharable`, as shown in the following listing.

Listing 6.7 Invalid usage of `@Sharable`

```
@Sharable                                                        #1
public class NotSharableHandler extends ChannelInboundHandlerAdapter {
    private int count;
```

```
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        count++;                                                          #2

        System.out.println("channelRead(...) called the "
                + count + " time");                                       #3
        ctx.fireChannelRead(msg();
    }

}
```

#1 Annotate with @Sharable
#2 Increment the count field
#3 Log method call and forward to next ChannelHandler

Why is the use of `@Sharable` wrong here? It's easy to guess once you look at the code. The problem is that we're using a field to hold the count of the method calls here. As soon as you add the same instance of the `NotSharableHandler` to the `ChannelPipeline` you get bad side effects, such as the count field being accessed and modified by different connections (and possible threads) now.

The rule of thumb is to use `@Sharable` only if you're sure that you can reuse the `ChannelHandler` on many different `Channels`.

Why share a ChannelHandler ?

You may wonder why you want to even try to share a `ChannelHandler` and so annotate it with `@Sharable`, but there are some situations where it can make sense.

First off if you can share it you will need to create less Object, which the Garbage Collector needs to recycle later. Another use case would be that you want to maintain some global statistics in the `ChannelHandler`, which are updated by all `Channels` (like concurrent connection count).

After we wrapped up on `ChannelPipeline` in this section we will move over to the `Channel` state model before moving on to the `ChannelHandlers` itself. The state model will be explained first as the states itself map to various methods oft he `ChannelHandlers` itself.

6.3 The state model

Netty has a simple but powerful state model that maps perfectly to the `ChannelInboundHandler` methods. We'll look at `ChannelInboundHandler` in section 6.4.2. So to make it easier to undersand the `ChannelInboundHandler` (and its parent called `ChannelHandler`) weill cover the states first.

There are four different states as shown in table 6.5.

Table 6.5 The lifecycle states of a channel

State	Description
`channelUnregistered`	The channel was created, but it isn't registered to its `EventLoop`.
`channelRegistered`	The channel is registered to its `EventLoop`.
`channelActive`	The channel is active, which means it's connected to its remote peer. It's now possible to receive and send data.
`channelInactive`	The channel isn't connected to the remote peer.

The state of the `Channel` changes during its lifetime, so state changes are triggered. Typically you see four state changes during the lifetime of the `Channel`, as shown in figure 6.5.

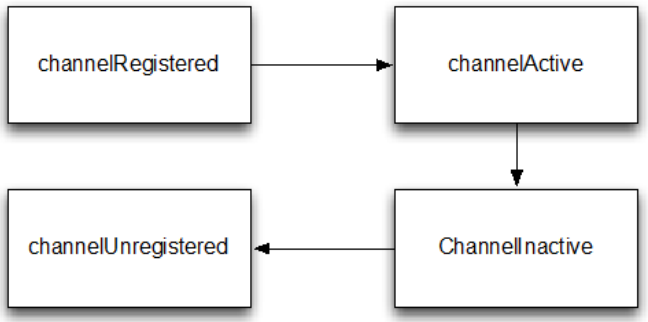


Figure 6.5 State model

Bye, bye channelUnregistered

One note to make about `channelUnregistered` is that it was removed from the code-base recently and so will be not present anymore in Netty 5. This is because unregistered a `Channel` is quite tricky as it is not clear anymore which Thread must be used to execute further actions.

This means the deregistration of a `Channel` from it's `EventLoop` is not possible by the user anymore. Please use the `channelInactive(...)` callback to replace any usage of `channelUnregistered(...)`.

You'll learn more about the operations that can be executed on a `ByteBuf` in a later section; for now, keep this in mind while we review the different types of `ByteBuf` that you'll most likely use.

6.4 ChannelHandlers and their types

Netty supports intercept operation or reacting on state changes via `ChannelHandler`. This makes it quite easy to write your custom processing logic in a reusable way.
There are two different types of `ChannelHandlers` that Netty supports, as shown in table 6.6.

Table 6.6 ChannelHandler types

Type	Description
Inbound Handler	`Processes` inbound data (received data) and state changes of all kinds
Outbound Handler	`Processes` outbound data (to-be-sent data) and allows to intercept all kind of operations

I'll discuss each type, but let's start with the base interface for all of them.

ChannelHandler types merge

Netty 5 will not have different types of `ChannelHandler` anymore; this means there is only `ChannelHandler` and not `ChannelInboundHandler` and `ChannelOutboundHandler`. The methods of `ChannelInboundHandler` and `ChannelOutboundHandler` were merged into `ChannelHandler` directly. This simplifies the hierarchy a lot for the user while still gives the same flexibility. If you are only interested in handling i.e. inbound events you could still just extend `ChannelInboundHandlerAdapter` and override only the inbound operations.

Just keep this in mind if you are planning to use Netty 5 once it is out.

6.4.1 ChannelHandler—the parent of all

Netty uses a well-defined type-hierarchy to represent the different handler types. The parent of all of them is the `ChannelHandler`. It provides lifecycle operations that are called after a `ChannelHandler` is added or removed from its `ChannelPipeline`.

Table 6.7 ChannelHandler methods

Type	Description
`handlerAdded(…)` `handlerRemoved(…)`	Lifecycle methods that get called during adding/removing the `ChannelHandler` from the `ChannelPipeline`.
`exceptionCaught(…)`	Called If an error happens during processing in the `ChannelPipeline`.

Each of the listed methods in table 6.7 takes a `ChannelHandlerContext` as a parameter when it's called.

Please refer to the section about `ChannelHandlerContext` for more information about it and what you can do with it.

Netty provides a skeleton implementation for `ChannelHandler` called `ChannelHandlerAdapter`. This provides base implementations for all of its methods so you can implement (override) only the methods you're interested in. Basically what it does is forward the "event" to the next `ChannelHandler` in the `ChannelPipeline` until the end is hit.

6.4.2 Inbound handlers

Inbound handlers handle inbound events and state changes. These are the right ones if you want to react to anything on inbound events. In this section, we'll explore the different `ChannelHandler` subtypes that allow you to hook in the inbound logic amd their methods.

CHANNELINBOUNDHANDLER

The `ChannelInboundHandler` provides methods that are called once the `Channel` state changes or data was recieved. These methods map to the channel state model explained in detail in section 6.3.

`ChannelInboundHandler` is a subtype of `ChannelHandler` and also exposes all of its methods. Table 6.8 lists the `ChannelInboundHandler` methods.

Table 6.8 ChannelInboundHandler methods

Type	Description
`channelRegistered(…)`	Invoked once a `Channel` was registered to its `EventLoop` and is now able to handle I/O.
`channelUnregistered(…)`	Invoked once a `Channel` was deregistered from its `EventLoop` and does not handle any I/O.
`channelActive(…)`	Invoked once a `Channel` is active which means the `Channel` is connected/bound and ready.
`channelInactive(…)`	Invoked once a `Channel` change from active mode and isn't connected to its remote peer anymore.
`channelReadComplete(…)`	Invoked once a read operation on the `Channel` completed.
`channelRead(…)`	Invoked if there is something read from the `Channel`.
`userEventTriggered(…)`	Invoked if the user triggered an event with some custom object.

Every one of these methods is a counterpart of a method that can be invoked on the `ChannelHandlerContext` and `ChannelPipeline`.

Like the mentioned skelton for `ChannelHandler` Netty provides also a skeleton implementation for `ChannelInboundHandler` implementations called

`ChannelInboundHandlerAdapter`. This provides base implementations for all of its methods and allows you to implement (override) only the methods you're interested in. All of these methods' implementations, by default, forward the event to the next `ChannelInboundHandler` in the `ChannelPipeline` by calling the same method on the `ChannelHandlerContext`.

Important to note is that the `ChannelInboundHandler` which handles received messages and so `@Override` the `channelRead(…)` method is responsible to release resources. This is especially important as Netty uses pooled resources for `ByteBuf` and so if you forgot to release the resource you will end up with a resource leak.

Listing 6.8 Handler to discard data

```
@Sharable
public class DiscardHandler extends ChannelInboundHandlerAdapter {   #1

    @Override
    public void channelRead(ChannelHandlerContext ctx,
            Object msg) {
        ReferenceCountUtil.release(msg);                              #2

    }
}
```

#1 Extend ChannelInboundHandlerAdapter
#2 Discard received message by pass it to ReferenceCountUtil.release(…)

Missing to pass the message to ReferenceCountUtil.release(…) will have the effect of creating a resource leak.

Fortunately Netty will log resources which was missed to be released with `WARN` loglevel, so it should be quite easy for you to figure out when you missed to do so.

As releasing the resources by hand can be cumbersome there is also `SimpleChannelInboundHandler`, which will take care of releasing the resource for you. This way you will not need to care about releasing at all. The only important thing here is to remember that if you use `SimpleChannelInboundHandler`, it will release the message once it was processed and so you **MUST NOT** store a reference to it for later usage.

So how does this change the previous example? Listing 6.9 shows the same implementation but with the use of `SimpleChannelInboundHandler`.

Listing 6.9 Handler to discard data

```
@Sharable
public class SimpleDiscardHandler
        extends SimpleChannelInboundHandler<Object> {                 #1

    @Override
    public void channelRead0(ChannelHandlerContext ctx,
            Object msg) {
        // No need to do anything special                             #2

    }
}
```

#1 Extend SimpleChannelInboundHandler
#2 Discard received message but no need for any release of resources

If you need to be notified about other state changes you could override one of the other methods that are part of the handler.

If you use ChannelInboundHandler, ChannelInboundHandlerAdapter or SimpleChannelInboundhandler depends on your needs. Most of the times you use SimpleChannelInboundHandler in the case of handling messages and ChannelInboundHandlerAdapter for other inbound events / state changes.

Message handling and reference counting

As you may remember from the previous chapter Netty uses reference counting to handle pooled `ByteBuf`'s. Thus it is important to make sure the reference count is adjusted after a `ByteBuf` is completely processed.

Because of this it is quite important to understand how `ChannelOutboundHandlerAdapter` differs from `SimpleChannelInboundHandler`. `ChannelInboundHandlerAdapter` does not call „release" on the message once it was passed to `channelRead(...)` and so its the responsibility of the user to do so if the message was consumed. `SimpleChannelInboundHandler` is different here as it automatically release a message after each `channelRead(...)` call, and thus expect your code to either consume the message or call `retain()` on it if you want to use it after the method returns.

Not correctly releasing a message will produce a memory leak, lucky enough Netty will by default log a message if this happens.

CHANNELINITIALIZER

There's one slightly modified `ChannelInboundHandler` that deserves more attention: `ChannelInitializer`. It does exactly what its name states. It allows you to init the `Channel` once it's registered with its `EventLoop` and ready to process I/O.

The `ChannelInitializer` is mainly used to set up the `ChannelPipeline` for each `Channel` that is created. This is done as part of the bootstrapping (see chapter 9 for a deeper look). For now, let's remember that there's an extra `ChannelInboundHandler` for that.

6.4.3 Outbound handlers

Now that you've had a look at how `ChannelHandler` allows you to hook into inbound operations and data, it's time to look at those `ChannelHandler` implementations that allow you the same for outbound operations and data. This section shows you all of them.

The `ChannelOutboundHandler` provides methods that are called when outbound operations are requested. These are the methods exposed by `Channel`, `ChannelPipeline`, and `ChannelHandlerContext`.

What makes it powerful is the ability to implement a `ChannelOutboundHandler` and defer an operation / event on request. This opens some powerful and flexible ways to handle requests. For example, you could defer flush operations once nothing should be written to the remote peer and pick them up later once it's allowed again.

Table 6.9 shows all the methods that are supported.

Table 6.9 ChannelOutboundHandler methods

Type	Description
`bind(…)`	Invoked once a request to bind the `Channel` to a local address is made.
`connect(…)`	Invoked once a request to connect the `Channel` to the remote peer is made.
`disconnect(…)`	Invoked once a request to disconnect the `Channel` from the remote peer is made.
`close(…)`	Invoked once a request to close the `Channel` is made
`deregister(…)`	Invoked once a request to deregister the `Channel` from its `EventLoop` is made.
`read(…)`	Invoked once a request to read more data from the `Channel` is made.
`flush(…)`	Invoked once a request to flush queued data to the remote peer through the `Channel` is made.
`write(…)`	Invoked once a `message` should be written through the Channel to the remote peer is made.

As shown in the hierarchy, `ChannelOutboundHandler` is a subtype of `ChannelHandler` and exposes all of its methods.

Allmost all of the methods take a `ChannelPromise` as an argument that MUST be notified once the request should stop to get forwarded through the `ChannelPipeline`.

Netty provides a skeleton implementation for `ChannelOutboundHandler` implementations called `ChannelOutboundHandlerAdapter`. This provides base implementations for all of its methods and allows you to implement (override) only the methods you're interested in. All these method implementations, by default, forward the event to the next `ChannelOutboundHandler` in the `ChannelPipeline` by calling the same method on the `ChannelHandlerContext`.

Here the same is true as for the `ChannelInboundHandler`. If you handle a write operation and discard a message it's you responsible to release it probably.

Now let's look at how you could make use of this in practice. The following listing shows an implementation that discarded all written data.

Listing 6.10 Handler to discard outbound data

```
@Sharable
public class DiscardOutboundHandler
        extends ChannelOutboundHandlerAdapter {                    #1

    @Override
    public void write(ChannelHandlerContext ctx,
            Object msg, ChannelPromise promise) {
        ReferenceCountUtil.release(msg);                            #2
        promise.setSuccess();                                       #3

    }

}
```

#1 Extend ChannelOutboundHandlerAdapter
#2 Release resource by using ReferenceCountUtil.release(…)
#3 Notify ChannelPromise that data handled

It's important to remember to release resources and notify the `ChannelPromise`. If the `ChannelPromise` is not notified it may lead to situations where a `ChannelFutureListener` is not notified about a handled message..

Outbound Message handling and reference counting

It's the responsibility of the user to call `ReferenceCountUtil.release(message)` if the message is consumed and not passed to the next `ChannelOutboundHandler` in the `ChannelPipeline`. Once the message is passed over to the actual transport it will be released automatically by it once the message was written or the `Channel` was closed.

6.5 Troubleshoot reference leaks

One of the tradeoffs of reference-counting is that the user have to be carefully when consume messages. While the JVM will still be able to GC such a message (as it is not aware of the reference-counting) this message will not be put back in the pool from which it may be obtained before. Thus chances are good that you will run out of resources at one point if you do not carefully release these messages.

To make it easier fort he user to find missing releases Netty contains a so called `ResourceLeakDetector` which will sample about 1% of buffer allocations to check if there is a leak in your application. Because of the 1% sampling, the overhead is quite small.

In case of a detected leak you will see a log message similar to the following.

```
LEAK: ByteBuf.release() was not called before it's garbage-collected. Enable advanced
      leak reporting to find out where the leak occurred. To enable advanced leak
      reporting, specify the JVM option '-Dio.netty.leakDetectionLevel=advanced' or
      call ResourceLeakDetector.setLevel()
Relaunch your application with the JVM option mentioned above, then you'll see the
      recent locations of your application where the leaked buffer was accessed. The
      following output shows a leak from our unit test
      (XmlFrameDecoderTest.testDecodeWithXml()):
Running io.netty.handler.codec.xml.XmlFrameDecoderTest
15:03:36.886 [main] ERROR io.netty.util.ResourceLeakDetector - LEAK:
      ByteBuf.release() was not called before it's garbage-collected.
Recent access records: 1
#1:
      io.netty.buffer.AdvancedLeakAwareByteBuf.toString(AdvancedLeakAwareByteBuf.java
      :697)
      io.netty.handler.codec.xml.XmlFrameDecoderTest.testDecodeWithXml(XmlFrameDecode
      rTest.java:157)
      io.netty.handler.codec.xml.XmlFrameDecoderTest.testDecodeWithTwoMessages(XmlFra
      meDecoderTest.java:133)
      ...

Created at:
      io.netty.util.ResourceLeakDetector.open(ResourceLeakDetector.java:169)
      io.netty.buffer.AbstractByteBufAllocator.toLeakAwareBuffer(AbstractByteBufAlloc
      ator.java:42)
      io.netty.buffer.UnpooledByteBufAllocator.newDirectBuffer(UnpooledByteBufAllocat
      or.java:55)
      io.netty.buffer.AbstractByteBufAllocator.directBuffer(AbstractByteBufAllocator.
      java:155)
      io.netty.buffer.UnpooledUnsafeDirectByteBuf.copy(UnpooledUnsafeDirectByteBuf.ja
      va:465)
      io.netty.buffer.WrappedByteBuf.copy(WrappedByteBuf.java:697)
      io.netty.buffer.AdvancedLeakAwareByteBuf.copy(AdvancedLeakAwareByteBuf.java:656
      )
      io.netty.handler.codec.xml.XmlFrameDecoder.extractFrame(XmlFrameDecoder.java:19
      8)
      io.netty.handler.codec.xml.XmlFrameDecoder.decode(XmlFrameDecoder.java:174)
      io.netty.handler.codec.ByteToMessageDecoder.callDecode(ByteToMessageDecoder.jav
      a:227)
      io.netty.handler.codec.ByteToMessageDecoder.channelRead(ByteToMessageDecoder.ja
      va:140)
      io.netty.channel.ChannelHandlerInvokerUtil.invokeChannelReadNow(
      ...
```

6.5.1 Leak detection Levels

As today there are 4 different leak detection levels that can be enabled when needed.

Table 6.9 gives you an overview over the different levels.

Table 6.9 Leak detection levels

Level	Description
DISABLED	Disables Leak detection completely. While this even eliminates the 1 % overhead you should only do this once you have tested your application with great care.

`SIMPLE`	Tells if a leak was found or not. This again uses the sampling rate of 1%. This is the default level and a good fit for most cases.
`ADVANCED`	Tells if a leak was found and where the message was accessed, using the sampling rate of 1%.
`PARANOID`	Same as level ADVANCED with the main difference that every access is sampled. Because of this it has a massive impact when it comes to performance. Only use this while debugging and not on a production system!

Changing the Leak detection Level is as easy as specifying the `io.netty.leakDetectionLevel` System property.

For example:

```
# java -Dio.netty.leakDetectionLevel=paranoid
```

6.6 Summary

In this chapter you got an in-depth look into how Netty allows hooking into data processing with its `ChannelHandler` implementation. The chapter shows how `ChannelHandlers` are chained and how the `ChannelPipeline` uses them.

It highlighted the differences between inbound and outbound handlers and the differences in operating on bytes or messages of various types.

The next chapter will focus on the codec abstraction of Netty, which makes writing codecs much easier than using the raw `ChannelHandler` interfaces.

7

The Codec framework

This chapter covers

- `Decoder`
- `Encoder`
- `Codec`

In the last chapter, you learned about the various ways to hook into the processing chain and how you can intercept operations or data. While `ChannelHandler` works to implement all kinds of logic, there's still room for improvements to make it easy to write codecs which often have logic in common.

To fulfill this, Netty offers a codec framework that makes writing custom codecs (which also includes decoders and encoders) for your protocol a piece of cake and allows for easy reuse and encapsulation.

In this chapter you will learn everything you need to know to implement all the decoding and encoding logic to built your own codec and so support any protocol that your application wants to "speak".

For example if you would like to build a Mail-Server on top of Netty you would implement the decoder and encoder for the various needed protocols like `POP3`[9], `IMAP`[10] and `SMTP`[11].

7.1 What is a Codec

Remember that everything is done by transferring bytes from one peer to another when transferring data over the network. Whenever you write a network-based program, you'll need to implement some kind of codec. The codec defines how the raw bytes need to be parsed and converted to some kind of logic unit that represents a custom message. The same is true for converting the message back to raw bytes that can be transmitted back over the network.

A codec is made up of two parts:

- Decoder
- Encoder

The decoder is responsible for decoding from bytes to the message or to another sequence of bytes. The encoder does the reverse; it encodes from a message back to bytes or from bytes to another sequence of bytes.

This should make it clear that the decoder is for inbound and the encoder is for outbound data.

[9] http://www.ietf.org/rfc/rfc1939.txt
[10] https://www.ietf.org/rfc/rfc2060.txt
[11] http://www.ietf.org/rfc/rfc2821.txt

Message handling

As mention in Chapter 5 and Chapter 6 you need to take special care about message because of reference counting which may be used. For Decoder and Encoder the contract is quite simple. Once a message is decoded or encoded it is automatically released via `ReferenceCountUtil.release(message)`. If you want to explicit not release the message (as you may need to keep a reference for later usage) you need to call `ReferenceCountUtil.retain(message)`. This will make sure the reference count is incremented and so the message not released.

Let's look at the various abstract base classes in Netty that allows implementing the decoder and encoder in an easy fashion.

7.2 Decoders

Netty provides a rich set of abstract base classes that help you easily write decoders. These are divided into two different types:

- Decoders that decode from bytes to message (`ByteToMessageDecoder` and `ReplayingDecoder`)
- Decoders that decode from message to message (`MessageToMessageDecoder`)

This section will give you an overview about the different abstract base classes you can use to implement your decoder and help you understand what a decoder is good for.

Before I dive into the actual abstract classes that Netty provides, let's define what the decoder's responsibility is. A decoder is responsible for decoding inbound data from one format to another one. Because a decoder handles inbound data, a decoder is an abstract implementation of `ChannelInboundHandler` (as learned in chapter 6).

You may ask yourself when you'll need a decoder in practice. It is quite simple; whenever you need to transform inbound data for the next `ChannelInboundHandler` in the `ChannelPipeline`.

It's even more flexible than you may expect, as you can put as many decoders in the `ChannelPipeline` as you need, thanks to the design of the `ChannelPipeline`, which allows you to assemble your logic of reusable components.

For more details on `ChannelInboundHandler` and `ChannelPipeline` please refer back to chapter 6.

7.2.1 ByteToMessageDecoder – Decode bytes to messages

Often you want to decode from bytes to messages or even from bytes to another sequence of bytes. This is such a common task that Netty ship an abstract base class that you can use to do this named `ByteToMessageDecoder`.

ByteToMessageDecoder will „buffer" received data until you can process them. This is needed as you can't expect to get all needed bytes at once. This is because you have no control over the remote peer and so if it will send all data at once.

Table 7.2 shows the two most interesting methods of `ByteToMessageDecoder` and explains how these can be used.

Table 7.2 Methods of ByteToMessageDecoder

Method name	Description
`decode()`	The `decode()` method is the only abstract method you need to implement. It's called with a `ByteBuf` that holds all the received bytes and a List into which decoded messages must be added. The `decode()` method is called as long as the List is not empty on return. All the messages that are added to the list are passed through the next handler in the pipeline.
`decodeLast()`	Default implementation delegates to `decode()`. This method is called once, which is when the `Channel` goes inactive. If you need special handling here you may override `decodeLast()` to implement it.

Let's imagine we have a stream of bytes written from a remote peer to us, and that it contains simple integers. We want to handle each integer separately later in the `ChannelPipeline`, so we want to read the integers from the inbound `ByteBuf` and pass each integer separately to the next `ChannelInboundHandler` in the `ChannelPipeline`. As we need to "decode" bytes into integers we will extend the ByteToMessageDecoder in our implementation, which we call `ToIntegerDecoder`. The needed logic is shown in Figure 7.2.

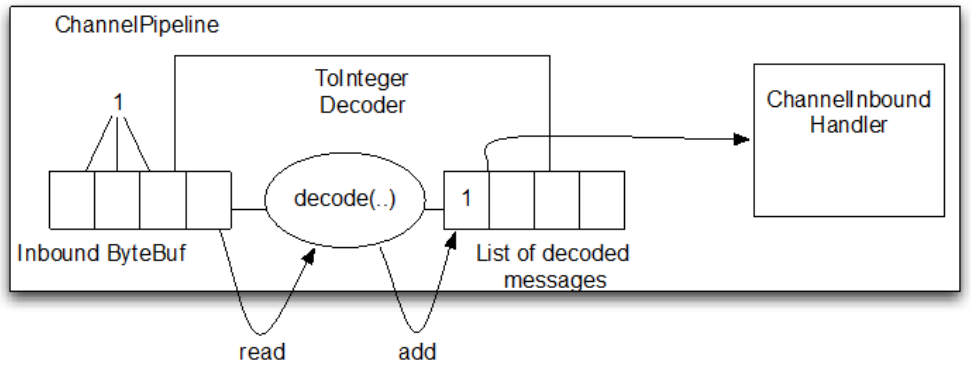


Figure 7.2 Logic of ToIntegerDecoder that decode bytes to integers

You see in figure 7.2 that it will read bytes from the inbound `ByteBuf` of the `ToIntegerDecoder`, decode them, and write the decoded messages (int this case `Integer`) to the next `ChannelInboundHandler` in the `ChannelPipeline`. The figure also shows each integer will take up four bytes in the `ByteBuf`.

The following listing shows how an implementation is done for this.

Listing 7.3 ByteToMessageDecoder that decodes to Integer

```
public class ToIntegerDecoder extends ByteToMessageDecoder {          #1

    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in,
            List<Object> out) throws Exception {
        if (in.readableBytes() >= 4) {                                 #2
            out.add(in.readInt());                                     #3
        }
    }
}
```

#1 Implementation extends ByteToMessageDecode to decode bytes to messages
#2 Check if there are at least 4 bytes readable (and int is 4 bytes long)
#3 Read integer from inbound ByteBuf, add to the List of decodec messages

Using `ByteToMessageDecoder` makes writing decoders that decode from byte to message easy as you saw in listing 7.2. But you may have noticed one little thing that is sometimes annoying. You need to check if there are enough bytes ready to read on the input `ByteBuf` before you do the actual read operation.

For a more complex example, please refer to the `LineBasedFrameDecoder`. This is part of Netty itself and can be found in the `io.netty.handler.codec` package. In addition to this, there are many other decoder implementations included, as decoding from bytes to messages is often useful.

Wouldn't it be preferable if you would not need to check if enough bytes are ready to read? Yes, it sometimes is and Netty addresses this with a special decoder called `ReplayingDecoder` that eliminate the need for this, buy adding a slightly overhead. The next section will look into `ReplayingDecoder`

7.2.2 ReplayingDecoder – Decode bytes to messages in a easy fashion

`ReplayingDecoder` is a special abstract base class for byte-to-message decoding that would be to hard to implement if you had to check if there's enough data in the `ByteBuf` all the time before calling operations on it. It does this by wrapping the input `ByteBuf` with a special implementation that checks if there's enough data ready and if not, throws a special `Signal` that it handles internally to detect it. Once such a signal is thrown, the `decode` loop stops.

Because of this wrapping, `ReplayingDecoder` comes with some limitations:

- Not all operations on the `ByteBuf` are supported, and if you call an unsupported operation, it will throw an `UnreplayableOperationException`.
- `ByteBuf.readableBytes()` won't return what you expect most of the time.

- It is slightly slower then using `ByteToMessageDecoder`

If you can live with the listed limitations, you may prefer the `ReplayingDecoder` over the `ByteToMessageDecoder`. The rule of thumb is, if you can use the `ByteToMessageDecoder` without introducing too much complexity, do it. If this isn't the case, use the `ReplayingDecoder`.

As `ReplayingDecoder` extends `ByteToMessageDecoder` the exposed methods are the same as those listed in Table 7.2 before.

So let us implement `ToIntegerDecoder` again but now with `ReplayingDecoder` in Listing 7.4.

Listing 7.4 `ReplayingDecoder` usage

```
public class ToIntegerDecoder2 extends ReplayingDecoder<Void> {        #1

    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in,
            List<Object> out) throws Exception {
        out.add(in.readInt());                                          #2
      }
}
```

#1 Implementation extends ReplayingDecoder to decode bytes to messages
#2 Read integer from inbound ByteBuf and add it to the List of decoded messages

When reading the integer from the inbound `ByteBuf`, if not enough bytes are readable, it will throw a `Signal` which will be catched by the `ReplayDecoder` itself. The `decode(…)` method will be called later, once more data is ready. Otherwise, add the decoded message to the `List`.

Compare listing 7.3 with listing 7.2 should make it easy to spot that the implementation is less complex then the previous, even with this very basic implementation.

Now imagine how to simplify the code if you need to implement something more complex. Again, using `ReplayingDecoder` or `ByteToMessageDecoder` is often a matter of taste. The important fact here is that Netty provides you with something you can easily use. Which one you choose is up to you.

For a more complex example of the usage of `ReplayingDecoder`, please refer to the `HttpObjectDecoder` that can be found in the `io.netty.handler.codec.http` package.

But what do you do if you want to decode from one message to another message (for example, POJO to POJO)? This can be done using `MessageToMessageDecoder`, which is explained in the next section.

7.2.3 MessageToMessageDecoder—Decode messages to messages

If you want to decode a message to another type of message `MessageToMessageDecoder` is the easiest way to go. The semantic of `MessageToMessageDecoder` is quite the same as of the previous explained `ByteToMessageDecoder` and `ReplayingDecoder` (explained in section 7.2.1 and 7.2.2).

Again let us have a look at the methods. Table 7.3 lists and explains them.

Table 7.3 Methods of MessageToMessageDecoder

Method name	Description
`decode()`	The `decode()` method is the only abstract method you need to implement. It's called for each inbound message of the decoder and lets you decode the message in an other message format . The decoded messages are then passed t o the next `ChannelInboundHandler` in the `ChannelPipeline`..
`decodeLast()`	Default implementation delegates to `decode()`. `decodeLast()` is only called one time, which is when the `Channel` goes inactive. If you need special handling here you may override `decodeLast()` to implement it.

To illustrate some uses let me give you an example. Imagine you have Integers and need to convert them to Strings. This should be done as part of the `ChannelPipeline` and implemented as a separate decoder to make it as flexible and reusable as possible.

As we need to decode from one message (Integer) to another message (String) a `MessageToMessageDecoder` is the right abstract base class to extend for our implementation that we cal `IntegerToStringDecoder`.

Figure 7.3 shows the actual logic of the class that needs to be implemented.

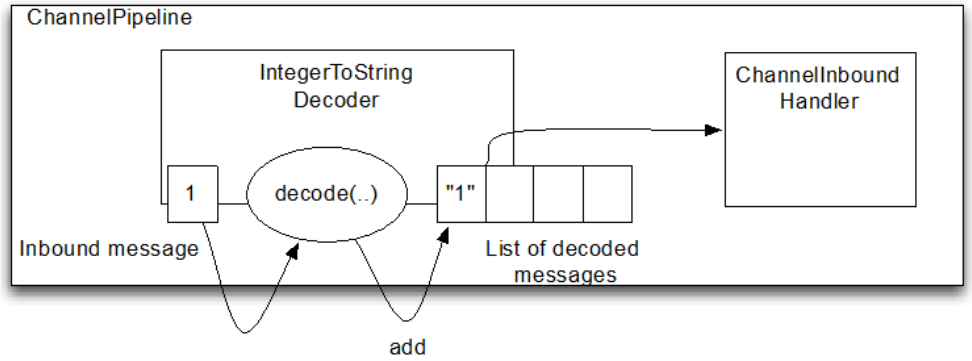


Figure 7.3 Logic of IntegerToStringDecoder

As it operates on messages and not bytes, the inbound message is directly passed in the `decode(…)` method and decoded messages will be added to the `List` of decoded messages and so forwarded to the next `ChannelInboundHandler` in the `ChannelPipeline`. This should sound familiar if you have read how `ByteToMessageDecoder` works.

Let's look at the concrete implementation of our needed logic in listing 7.5.

Listing 7.5 `MessageToMessageDecoder` **decodes integer to string**

```
public class IntegerToStringDecoder extends
        MessageToMessageDecoder<Integer> {                                  #1

    @Override
    public void decode(ChannelHandlerContext ctx, Integer msg
            List<Object> out) throws Exception {
        out.add(String.valueOf(msg));                                       #2
       }
}
```

#1 Implementation extends MessageToMessageDecoder
#2 Convert integer message string using String.valueOf()

In (#1), when the implementation extends the MessageToMessageDecoder to decode from one message type to another, the Type parameter (generic) is used to specify the input parameter. In our case, this is an integer type.

For a more complex example, please refer to the `HttpObjectAggregator` which can be found in the `io.netty.handler.codec.http` package.

7.2.4 Decoders summary

You should now have a good knowledge about the abstract base classes (`ByteToMessageDecoder`, `ReplayingDecoder` and `MessageToMessageDecoder`) Netty provides to build decoders and what decoders are good for in general. Decoders are only one part of the whole picture. What is missing is the counterpart that allows to transform outbound data. This is the job of an encoder, which is another part of the codec API.

The next sections will give you more insight into how you can write your own encoder.

7.3 Encoders

As a counterpart to the decoders Netty offers, base classes help you to write encoders in an easy way. Again, these are divided into different types similar to what you saw in section 7.2:

- Encoders that encode from message to bytes
- Encoders that encode from message to message

Before going deeper, let's define the responsibility of an encoder.

An encoder is responsible for encoding outbound data from one format to another. As an encoder handles outbound data, it implements `ChanneOutboundHandler`.

With this in mind, it's time to look at the provided abstract classes that helps you implement your encoder.

7.3.1 MessageToByteEncoder

As you learned in section 7.2.2 you often want to convert from bytes to messages. You already saw how to do this for inbound data via the `ByteToMessageDecoder`. But what do you do to move back from message to bytes?

`MessageToByteEncoder` is provided to serve as an abstract base class for your encoder implementations that need to transform messages back into bytes.

Table 7.5 shows the exact methods you need to implement for your encoder, which is named encode.

Table 7.5 Methods of MessageToByteEncoder

Method name	Description
`encode()`	The `encode()` method is the only abstract method you need to implement. It's called with the outbound message, which was received by this encoder and encodes it into a `ByteBuf`. The `ByteBuf` is then forwarded to the next `ChannelOutboundHandler` in the `ChannelPipeline`.

The reason why there is only one method compared to decoders where we have two messages is that decoders often need to to «produce» a last message once the `Channel` was closed. For this reason the `decodeLast()` method is used. This is not the case for an encoder where there is no need to produce any extra message when the connection was closed as there would be nothing you could do with the produced message, as writing to a closed `Channel` would fail anyway.

Let's see it in action to better understand its usage. I've written single short values and want to encode them into a `ByteBuf` to finally send them over the wire. `IntegerToByteEncoder` is the implementation that's used for this purpose.

Figure 7.5 shows the logic.

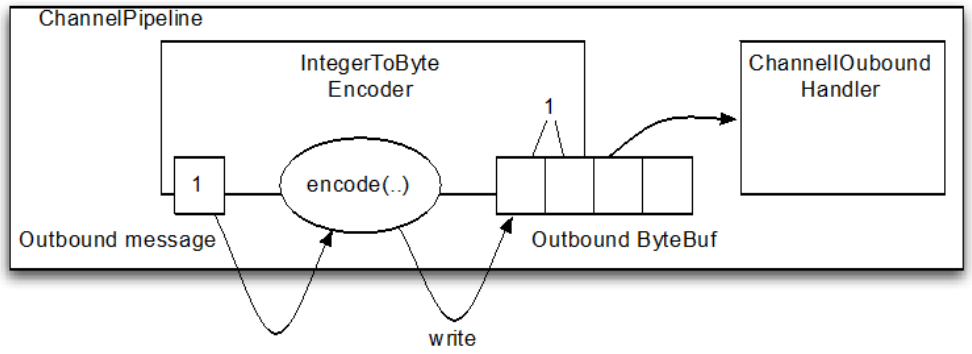


Figure 7.5 Logic of IntegerToByteEncoder

Figure 7.5 shows that it will "receive" short messages, encode, and write to a `ByteBuf`. This `ByteBuf` is then forwarded to the next `ChannelOutboundHandler` in the `ChannelPipeline`. As you can also see in figure 7.5, every short will take up 2 bytes in the `ByteBuf`.

The following listing shows how an implementation is done for this.

Listing 7.8 `IntegerToByteEncoder` encodes shorts into a `ByteBuf`

```
public class IntegerToByteEncoder extends
        MessageToByteEncoder<Short> {                                    #1

    @Override
    public void encode(ChannelHandlerContext ctx, Short msg, ByteBuf out)
            throws Exception {
        out.writeShort(msg);                                             #2
    }
}
```

#1 Implementation extends MessageToByteEncoder
#2 Write short into ByteBuf

Netty comes with several `MessageToByteEncoder` implementations which can be serve as examples to how to make use of the `MessageToByteEncoder` to create your own implementation. Please have a look at the `WebSocket08FrameEncoder` for a more real-world example. It can be found in the `io.netty.handler.codec.http.websocketx` package.

7.3.2 MessageToMessageEncoder

You've seen several decoder and encoder types. But one is still missing to complete the mix. Suppose you need a way to encode from one message to another, similar to what you did for inbound data with `MessageToMessageDecoder`.

`MessageToMessageEncoder` fills this gap. Table 7.6 shows the method you need to implement for your encoder, which is named `encode`.

Table 7.6 Methods of MessageToMessageEncoder

Name	Description
`encode()`	The `encode()` method is the only abstract method you need to implement. It's called for each message written with `write(…)` and encode the message to one or multiple new messages. The encoded messages are then forwarded to the next `ChannelOutboundHandler` in the `ChannelPipeline`.

The reason for only having one method to implement is the same as for `MessageToByteEncoder`, where there is no need to produce any special message once a `Channel` is closed.

Again let us look at an example. Imaging you need to encode Integer messages to String messages. You can do this easily with MessageToMessageEncoder.

Figure 7.6 shows the logic.

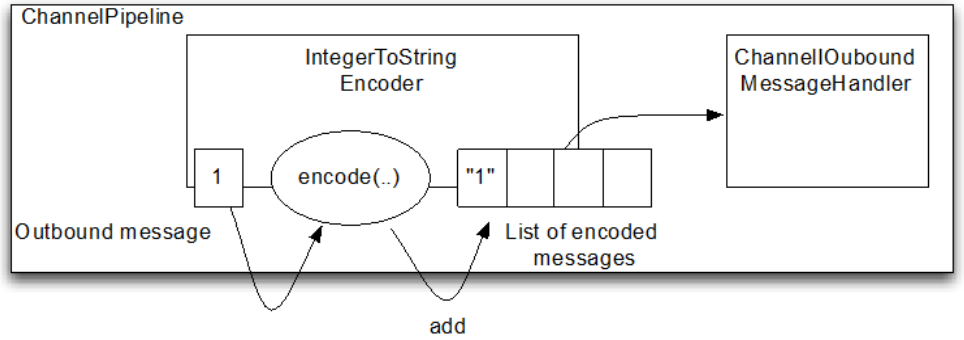


Figure 7.6 Logic of IntegerToStringEncoder

The encoder encodes Integer messages and forward the Integers to the next `ChannelOutboundHandler` in the `ChannelPipeline`.

The following listing 7.9 shows how an implementation for this is done.

Listing 7.9 `IntegerToStringEncoder` encodes integer to string

```
public class IntegerToStringEncoder extends
        MessageToMessageEncoder<Integer> {                          #1

    @Override
    public void encode(ChannelHandlerContext ctx, Integer msg
            List<Object> out) throws Exception {
        out.add(String.valueOf(msg));                               #2
    }
}
```

#1 Implementation extends MessageToMessageEncoder
#2 Convert integer to string and add to MessageBuf

Again there are more complex examples which use the `MessageToMessageEncoder`. One of those is the `ProtobufEncoder` which can be found in the `io.netty.handler.codec.protobuf` package.

7.4 Abstract Codec classes

Often you want to write an implementation, which handles decoding and encoding and is encapsulated in one class. Remember, decoding is for inbound data and encoding for outbound data. So instead of writing and using a separate decoder and encoder you can make use of the abstract Codec classs and provide an implementation which in fact is an decoder and encoder at the same time. This way all the logic for inbound and outbound is “bundled” in one class.

Now that you know what type of encoders and decoders are provided, you won't be surprised to learn that the codecs available handle the same type of decoding and encoding, again the difference is "only" that you can have the decoder and encoder in one implementation:

- byte-to-message decoding and encoding
- message-to-message decoding and encoding

If you're sure you'll need to have the encoder and decoder in the `ChannelPipeline` all the time and it won't work to only have one of the two, you may be better off using one of the various abstract codec classes which will be introduced in the following sub-sections.

You may ask yourself why I would not always use the codec classes but instead use a separate decoder an encoder? Doing so comes with the downside of lower chances of re-usibility. Having the decoder and encoder separate allows you for easier extending of one of both and also to only make use of one.

With this in mind, let's spend the next few sections taking a deeper look at the provided abstract codec classes to make it easier for you to compare them with the decoders and encoders.

7.4.1 ByteToMessageCodec—Decoding and encoding messages and bytes

Imagine you want to have a codec that decodes bytes to some kind of message (a POJO) and encodes the message back to bytes. For this case, a `ByteToMessageCodec` is a good choice.

Again, the codec is kind of a combination. So it's like having a `ByteToMessageDecoder` and the corresponding `MessageToByteDecoder` in the `ChannelPipeline`.

Let's look at the important methods in table 7.7.

Table 7.7 Methods of ByteToByteCodec

Method name	Description
`decode()`	The `decode()` method is called with the inbound `ByteBuf` of the codec and decode them to messages. Those messages are forwarded to the next ChannelInboundHandler in the `ChannelPipeline`. The `decode()` method will be called as long as bytes are consumed .
`decodeLast()`	Default implementation delegates to `decode()`. `decodeLast()` will only be called one time, which is when the `Channel` goes inactive. If you need special handling here you may override `decodeLast()` to implement it.
`encode()`	The `encode()` method is called for each message written through the `ChannelPipeline`. The encoded messages are written in a `ByteBuf` and the `ByteBuf` is forwarded to the next `ChannelOutboundHandler` in the `ChannelPipeline`.

Let's think about a good fit for such a `ByteToMessageCodec`.

Every codec that needs to decode from bytes to some custom message and back is a good fit. For example, I've used it in the past to build a SMTP codec that reads bytes and decodes them to a custom message type called `SmtpRequest`. The encode method then took `SmtpResponse` messages and encoded them to bytes.

You see it's especially useful for requests/responses combinations.

7.4.2 MessageToMessageCodec—Decoding and encoding messages

Sometimes you need to write a codec that's used to convert from one message to another message type and vice versa. `MessageToMessageCodec` makes this a piece of cake.

Before going into the details, let's look at the important methods in table 7.8.

Table 7.8 Methods of MessageToMessageCodec

Method name	Description
`decode()`	The `decode()` method is called with the inbound `message` of the codec and decode them to messages. Those messages are forwarded to the next ChannelInboundHandler in the `ChannelPipeline` Default implementation delegates to `decode()`.
`decodeLast()`	`decodeLast()` will only be called one time, which is when the `Channel` goes inactive. If you need special handling here you may override `decodeLast()` to implement it.
`encode()`	The `encode()` method is called for each message written through the `ChannelPipeline`. The encoded messages are forwarded to the next `ChannelOutboundHandler` in the `ChannelPipeline`.

But where can such a codec be useful?

There are many use cases, but one of the most common is when you need to convert a message from one API to another API on the fly. This is necessary for situations in which you need to talk with a custom or old API that uses another message type.

Listing 7.10 shows how such a conversation can take place using the `MessageToMessageCodec`. Here we convert from `WebSocketFrame` to `MyWebSocketFrame` and vise-versa.

Listing 7.10 Converting codec

```
public class WebSocketConvertHandler extends
        MessageToMessageCodec< WebSocketFrame,
        WebSocketConvertHandler.WebSocketFrame> {

    @Override
    protected void encode(ChannelHandlerContext ctx, MyWebSocketFrame msg
            List<Object> out) throws Exception {                          #1
        ByteBuf payload = msg.getData().duplicate().retain();
        switch (msg.getType()) {                                          #2
            case BINARY:
                out.add(new BinaryWebSocketFrame(payload));
                return;
            case TEXT:
                out.add(new TextWebSocketFrame(payload));
                return;
            case CLOSE:
                out.add(new CloseWebSocketFrame(true, 0, payload));
                return;
            case CONTINUATION:
                out.add(new ContinuationWebSocketFrame(payload));
                return;
            case PONG:
                out.add(new PongWebSocketFrame(payload));
                return;
            case PING:
                out.add(new PingWebSocketFrame(payload));
                return;
            default:
                throw new IllegalStateException(
                        "Unsupported websocket msg " + msg);
        }
    }

    @Override
    protected void decode(ChannelHandlerContext ctx, WebSocketFrame msg,
            List<Object> out) throws Exception {
        ByteBuf payload = msg.getData().duplicate().retain();
                                                                          #3
        if (msg instanceof BinaryWebSocketFrame) {
            out.add(new MyWebSocketFrame(
                    MyWebSocketFrame.FrameType.BINARY, payload));
            return;
        }
        if (msg instanceof CloseWebSocketFrame) {
            out.add(new MyWebSocketFrame (
                    MyWebSocketFrame.FrameType.CLOSE, payload));
            return;
        }
        if (msg instanceof PingWebSocketFrame) {
            out.add(new MyWebSocketFrame (
                    MyWebSocketFrame.FrameType.PING, payload));
            return;
        }
        if (msg instanceof PongWebSocketFrame) {
            out.add(new MyWebSocketFrame (
                    MyWebSocketFrame.FrameType.PONG, payload));
            return;
        }
```

```
            if (msg instanceof TextWebSocketFrame) {
                out.add(new MyWebSocketFrame (
                        MyWebSocketFrame.FrameType.TEXT, payload));
                return;
            }
            if (msg instanceof ContinuationWebSocketFrame) {
                out.add(new MyWebSocketFrame (
                        MyWebSocketFrame.FrameType.CONTINUATION, payload));
                return;
            }
            throw new IllegalStateException("Unsupported websocket msg " + msg);
        }

        public static final class MyWebSocketFrame {                          #4
            public enum FrameType {                                           #5
                BINARY,
                CLOSE,
                PING,
                PONG,
                TEXT,
                CONTINUATION
            }

            private final FrameType type;
            private final ByteBuf data;
            public WebSocketFrame(FrameType type, ByteBuf data) {
                this.type = type;
                this.data = data;
            }

            public FrameType getType() {
                return type;
            }

            public ByteBuf getData() {
                return data;
            }
        }
    }
```

#1 Encodes MyWebSocketFrame messages to WebSocketFrame messages
#2 Check the FrameType of the MyWebSocketFrame and create a new WebSocketFrame that matches the the FrameType.
#3 Decodes WebSocketFrame mssages to MyWebSocketFrame messages using the instanceof check to find the right FrameType.
#4 Our custom message from which we want to encode to WebSocketFrame messages and to which we want to decode from WebSocketFrame message.
#5 The type of the MyWebSocketFrame

After we have looked at the different abstract codec class we now move over to another way to combine decoders and encoders (this also allows to combine just `ChannelInboundHandler` and `ChannelOutboundHandler`). The difference is that you can just use pre-written decoder and encoder and combine them right away without the need to extend the abstract codec classes directly. This makes it quite more flexible and provides you with the most flexibility in this term.

7.4.3 CombinedChannelDuplexHandler—Combine your handlers

When you use codecs that act as a combination of a decoder and encoder, you lose the flexibility to use the decoder or encoder individually. You're forced to either have both or neither.

You may wonder if there's a way around this inflexibility problem that still allows you to have the decoder and encoder as a logic unit in the `ChannelPipeline`.

Fortunately, there is a solution for this called `CombinedChannelDuplexHandler`. Although this handler isn't part of the codec API itself, it's often used to build up a codec. .

CombinedChannelDuplexHandler in Netty 5

As part of merging ChannelInboundHandler and ChannelOutboundHandler into ChannelHandler in Netty 5 the CombinedChannelDuplexHandler was removed in Netty 5. This was mainly done as it is very hard to tell which ChannelHandler implements inbound and which outbound operations. Still if we would know this what should be done when one or both implement inbound and outbound operations?

Because of this problem the CombinedChannelDuplexHandler was removed. A user can now just either add both handlers to the ChannelPipeline or build its own combination.

To show how you can use `CombinedChannelDuplexHandler` to combine a decoder and encoder, let's create a decoder and encoder first. We'll use two simple examples to illustrate the use case.

Listing 7.11 shows a `ByteToCharDecoder`, which decodes bytes to chars and will be later combined.

Listing 7.11 `ByteToCharDecoder` decodes bytes into chars

```
public class ByteToCharDecoder extends
        ByteToMessageDecoder {                                    #1

    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in,
            List<Object> out) throws Exception {
        while (in.readableBytes() >= 2) {                         #2
            out.add(Character.valueOf(in.readChar()));
        }
    }
}
```

#1 Extends ByteToMessageDecoder
#2 Writes char into MessageBuf

Notice that the implementation extends `ByteToMessageDecoder` because it reads chars from a `ByteBuf`.

Now have a look at Listing 7.12, which represents the encoder that encodes chars back into bytes.

Listing 7.12 `CharToByteEncoder` encodes chars into bytes

```
public class CharToByteEncoder extends
        MessageToByteEncoder<Character> {                                  #1

    @Override
    public void encode(ChannelHandlerContext ctx, Character msg, ByteBuf out)
            throws Exception {
        out.writeChar(msg);                                                #2
    }
}
```

#1 Extends MessageToByteEncoder
#2 Writes char into ByteBuf

The implementation extends `MessageToByteEncoder` because it needs to encode char messages into a `ByteBuf`. This is done by writing the chars directly into the `ByteBuf`.

Now that we have a decoder and encoder, it's time to combine them to build up a codec.

Listing 7.13 shows how you would do this by make use of `CombinedChannelDuplexHandler`.

Listing 7.13 `CharToByteEncoder` encodes chars into bytes

```
public class CharCodec extends
    CombinedChannelDuplexHandler<ByteToCharDecoder, CharToByteEncoder> {  #1

    public CharCodec() {
        super(new ByteToCharDecoder(), new CharToByteEncoder());           #2
    }
}
```

#1 Handles inbound bytes and outbound messages
#2 Pass an instance of ByteToCharDecoder and CharToByteEncoder to the super constructor as it will delegate calls to them to combine them

This implementation can now be used to bundle the decoder and encoder. As you see, it's quite easy and often more flexible than using one of the *Codec classes.

What you choose to use is a matter of taste and style.

7.5 Summary

In this chapter, you learned how to use the provided codec API to write your own decoders and encoders. You also learned why using the codec API is the preferred way to write your decoders and encoders over using the plain `ChannelHandler` API.

In addition, you learned about the different abstract codec classes, which let you write a codec and let you handle decoding and encoding in one implementation.

If this isn't flexible enough, you also know how to combine a decoder and encoder to transform them to a logical unit without the need to extend any of the abstract codec classes.

In the next chapter, you'll get a brief overview about the provided `ChannelHandler` implementations and codecs that you can use out-of-the box to handle specific protocols and tasks, as these are part of Netty itself.

8

Provided ChannelHandlers and codecs

This chapter covers

- Securing Netty applications with SSL/TLS
- Building Netty HTTP/HTTPS applications
- Handling idle connections and timeouts
- Decoding delimiter- and length-based protocols
- Writing big data
- Serializing data

The previous chapter showed you how to create your own codec and also gave you a better understand of the inner workings of Netty's codec framework. With this new knowledge you can now write your own codec for your application. Nevertheless, wouldn't it be nice if Netty offered some standard `ChannelHandlers` and codecs?

Netty bundles many codecs and handlers for common protocol implementations (such as HTTP) so you don't have to reinvent the wheel. Those implementations can be reused out of the box, freeing you up to focus on more specific problems. This chapter will show you how to secure your Netty applications with SSL/TLS, how to write scalable HTTP servers, and how to use associated protocols such as WebSockets or Google's SPDY to get the best performance out of HTTP by using the Codecs / `ChannelHandlers` that Netty provides. All of these are common, and sometimes necessary, applications. This chapter will also introduce you to compression, for use when size really does matter.

8.1 Securing Netty applications with SSL/TLS

Communication of data over a network is insecure by default, as all the transmitted data is sent in plain text or binary protocols that are easy to decode. This becomes a problem once you want to transmit data that must be private.

Encryption comes into play in this situation SSL / TLS[12] are well-known standards and layered on top of many protocols to make sure data remains private. For example, if you use HTTPS or SMTPS, you're using encryption. Even if you're not aware of it, you've most likely already touched base with encryption at some point.

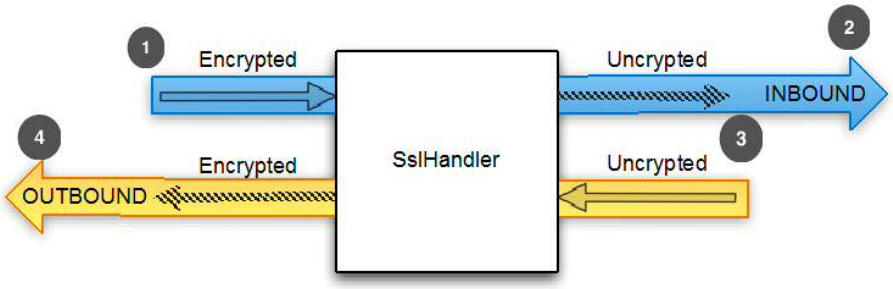
For SSL/TLS, Java ships an API which is exposed via `SslContext` and `SslEngine`. In fact, the `SslContext` can be used to obtain a new `SslEngine`, which can be used for decryption and encryption.

Netty extends Java's SSL engine to add functionality that makes it more suitable for Netty-based applications. It ships a `ChannelHandler` called `SslHandler` that wraps an `SslEngine` for

[12] http://tools.ietf.org/html/rfc5246

decryption and encryption of its network traffic. Which means the SslHandler internally uses an SslEngine to do the actual work but hide all the details from the user.

Figure 8.1 shows the data flow for the `SslHandler` implementation.



#1 Encrypted inbound data is intercepted by the SslHandler and gets decrypted
#2 The previous encrypted data was decrypted by the SslHandler
#3 Plain data is passed through the SslHandler
#4 The SslHandler encrypted the data and passed it outbound

Figure 8.1 Data flow through SslHandler for decryption and encryption

Listing 8.1 shows how you can use the `SslHandler` by adding it to the `ChannelPipeline` using a `ChannelInitializer` that you typically use to set up the `ChannelPipeline` once a `Channel` is registered.

Listing 8.1 Add SSL/TLS support

```
public class SslChannelInitializer extends
        ChannelInitializer<Channel>{

    private final SSLContext context;
    private final boolean client;
    private final boolean startTls;

    public SslChannelInitializer(SSLContext context,
        boolean client, boolean startTls) {                          #1
        this.context = context;
        this.client = client;
        this.startTls = startTls;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        SSLEngine engine = context.createSSLEngine();                #2
        engine.setUseClientMode(client);                             #3

        ch.pipeline().addFirst("ssl",
                new SslHandler(engine, startTls));                   #4
    }
}
```

#1 Use the constructor to pass the SSLContext to use and if it's a client and startTls should be used

#2 Obtain a new SslEngine from the SslContext. Use a new SslEngine for each SslHandler instance
#3 Set if the SslEngine is used in client or server mode
#4 Add the SslHandler in the pipeline as first handler

One important thing to note is that in almost all cases the `SslHandler` must be the first `ChannelHandler` in the `ChannelPipeline`. There may be some exceptions, but take this as rule of thumb. Recall that in chapter 6, we said the `ChannelPipeline` is like a LIFO[13] (last-in-first-out) queue for inbound messages and a FIFO[14] (first-in-first-out) queue for outbound messages. Adding the `SslHandler` first ensures that all other `ChannelHandlers` have applied their transformations/logic to the data before it's encrypted, thus ensuring that changes from all handlers are secured on a Netty server.

The `SslHandler` also has some useful methods, as shown in table 8.1, you can use to modify its behavior or get notified once the SSL/TLS handshake is complete (during the handshake the two peers validate each other and choose an encryption cipher that both support). Once the handshake completes successfully the data will be encrypted from this point of time. The SSL/TLS handshake will be executed automatically for you.

Table 8.1 Methods to modify a ChannelPipeline

Name	Description

`setHandshakeTimeout(...)`
`setHandshakeTimeoutMillis(...)`
`getHandshakeTimeoutMillis()`
	Set and get the timeout after which the handshake will fail and the handshake `ChannelFuture` is notified.

`setCloseNotifyTimeout(...)`
`setCloseNotifyTimeoutMillis(...)`
`getCloseNotifyTimeoutMillis()`
	Set and get the timeout after which the close notify will time out and the connection will close. This also results in having the close notify `ChannelFuture` fail.

`handshakeFuture()`	Returns a `ChannelFuture` that will get notified once the handshake is complete. If the handshake was done before it will return a `ChannelFuture` that contains the result of the previous handshake.
`close(...)`	Send the `close_notify` to request close and destroy the underlying `SslEngine`.

[13] https://en.wikipedia.org/wiki/LIFO_(computing)
[14] http://en.wikipedia.org/wiki/FIFO

8.2 Building Netty HTTP/HTTPS applications

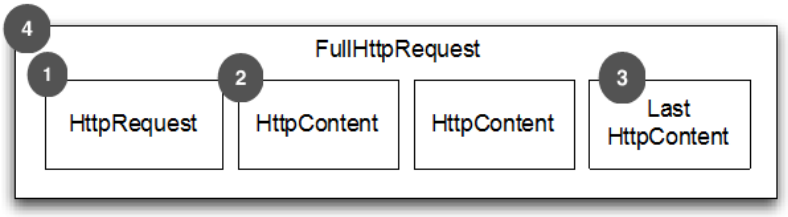
HTTP/HTTPS[15] is one of the most used protocols these days, and with the success of smartphones it gets more attention with each passing day. Almost every company has a homepage that you can access via HTTP or HTTPS, but this isn't the only use for it. Many services expose an API via HTTP(s) for easy use in a platform-independent manner.

Fortunately, Netty comes with `ChannelHandlers` that allow you to use HTTP(s) without requiring you to write your own codecs.

8.2.1 Netty's HTTP decoder, encoder, and codec

HTTP uses a Request–Response pattern, which means the client sends an HTTP request to the server and the server sends an HTTP response back. Netty makes it easy to work with HTTP by providing various encoders and decoders for handling HTTP protocols.

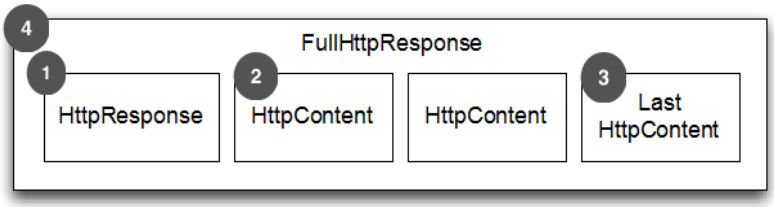
Figures 8.2 and 8.3 shows the commands producing and handling the HTTP-specific messages for requests and responses, respectively.



#1 First part of the HTTP-Request that contains headers and so on
#2 One more HttpContent part that contains chunks of data followed by another HttpContent part. There could be multiple or none of these.
#3 Special HttpContent subtype that marks the end of the HTTP request and may also contain trailing headers
#4 A full HTTP request with everything included

Figure 8.2 HTTP request parts

[15] http://www.w3.org/Protocols/rfc2616/rfc2616.html



#1 First part of the HTTP response that contains headers and so on
#2 One more HttpContent part that contains chunks of data followed by another HttpContent part. There could be multiple or none of these.
#3 Special HttpContent subtype that marks the end of the HTTP response and may also contain trailing headers
#4 A full HTTP response with everything included

Figure 8.3 HTTP response parts

As shown in figures 8.2 and 8.3, respectively, an HTTP request/response may consist of more than one message. Its end is always marked with the `LastHttpContent` message. The `FullHttpRequest` and `FullHttpResponse` message are a special subtype that represent a complete request and response. All types of HTTP messages (`FullHttpRequest`, `LastHttpContent`, and those shown in listing 8.2) implement the `HttpObject` interface as their parent.

Table 8.2 gives an overview of the HTTP decoders and encoders that handle and produce the messages.

Table 8.2 HTTP decoder and encoder

Name	Description
`HttpRequestEncoder`	Encodes `HttpRequest` , `HttpContent` and `LastHttpContent` messages to bytes.
`HttpResponseEncoder`	Encodes `HttpResponse`, `HttpContent` and `LastHttpContent` messages to bytes.
`HttpRequestDecoder`	Decodes bytes into `HttpRequest`, `HttpContent` and `LastHttpContent` messages.
`HttpResponseDecoder`	Decodes bytes into `HttpResponse`, `HttpContent` and `LastHttpContent` messages.

Listing 8.2 shows how easy it is to add support for HTTP to you application. All you need to do is adding the correct `ChannelHandlers` in the `ChannelPipeline`.

Listing 8.2 Add support for HTTP

```
public class HttpDecoderEncoderInitializer
        extends ChannelInitializer<Channel> {

    private final boolean client;

    public HttpDecoderEncoderInitializer(boolean client) {
        this.client = client;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        if (client) {
            pipeline.addLast("decoder", new HttpResponseDecoder());      #1
            pipeline.addLast("encoder", new HttpRequestEncoder());       #2
        } else {
            pipeline.addLast("decoder", new HttpRequestDecoder());       #3
            pipeline.addLast("encoder", new HttpResponseEncoder());      #4
        }
    }
}
```

#1 Add HttpResponseDecoder as decoder as a client will receive responses from the server
#2 Add HttpRequestEncoder as encoder as the client will send requests to the server
#3 Add HttpRequestDecoder as decoder as the server will receive request from the client
#4 Add HttpResponseEncoder as encoder as the server will send responses to the client

After you have the decoder, encoder in the `ChannelPipeline`, you'll be able to operate on the different `HttpObject` messages. But as an HTTP request and HTTP response can be made out of many "messages," you'll need to handle the different parts and may have to aggregate them to only work on "full" messages. This can be cumbersome. To solve this problem, Netty provides an aggregator, which will merge message parts in `FullHttpRequest` and `FullHttpResponse` messages, so you don't need to worry about receiving only "fragments" and always be sure you have the full message content at hand. The next section will explain message aggregation in more detail.

8.2.2 HTTP message aggregation

As explained in the previous section, when dealing with HTTP you may receive HTTP messages in fragments, because otherwise Netty would need to buffer the whole message until it's received.

Situations exist where you want to handle only "full" HTTP messages and be okay with some memory overhead (the memory overhead is caused be the fact that you will need to "buffer" the whole message in memory). For this purpose Netty bundles the `HttpObjectAggregator`.

With the `HttpObjectAggregator` in the `ChannelPipeline`, Netty will aggregate HTTP message parts for you and only forward `FullHttpResponse` and `FullHttpRequest` messages to the next `ChannelInboundHandler` in the `ChannelPipeline`. This eliminates the need to worry about fragmentation and ensures you act only on "complete" messages.

Automatic aggregation is as easy as adding another `ChannelHandler` in the `ChannelPipeline`. Listing 8.3 shows how to enable the aggregation.

Listing 8.3 Automatically aggregate HTTP message fragments

```
public class HttpAggregatorInitializer extends ChannelInitializer<Channel> {

    private final boolean client;

    public HttpAggregatorInitializer(boolean client) {
        this.client = client;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        if (client) {
            pipeline.addLast("codec", new HttpClientCodec());           #1
        } else {
            pipeline.addLast("codec", new HttpServerCodec());           #2
        }
        pipeline.addLast("aggegator",
            new HttpObjectAggregator(512 * 1024));                      #3
    }
}
```

#1 Add HttpClientCodec as we are in client mode
#2 Add HttpServerCodec as we are in server mode
#3 Add HttpObjectAggregator to the ChannelPipeline, using a max message size of 512kb. After the message is getting bigger a TooLongFrameException is thrown.

As you can see, it's easy to let Netty automatically aggregate the message parts for you. Be aware that to guard your server against DoS attacks you need to choose a sane limit for the maximum message size, that is configured by the constructor of `HttpObjectAggregator`. How big the maximum message size should be depends on your use case, concurrent requests/responses, and, of course, the amount of usable memory available.

8.2.3 HTTP compression

When using HTTP it's often advisable to use compression to minimize the data you need to transfer over the wire. The performance gain from this isn't free, as compression puts more load on the CPU. Although today's increased computing capabilities mean that this isn't as much of an issue as a few years ago, using compression is a good idea most of the time if you transfer content that benefits from compression like text.

Netty supports compression via "gzip" and "deflate" provided by two `ChannelHandler` implementations: one for compression and one for decompression. Compression and decompression are shown in listing 8.4.

Listing 8.4 Automatically compress HTTP messages

```
public class HttpAggregatorInitializer extends ChannelInitializer<Channel> {

    private final boolean client;

    public HttpAggregatorInitializer(boolean client) {
        this.client = client;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        if (client) {
            pipeline.addLast("codec", new HttpClientCodec());           #1
                pipeline.addLast("decompressor",
                        new HttpContentDecompressor());                 #2
        } else {
            pipeline.addLast("codec", new HttpServerCodec());           #3
                pipeline.addLast("decompressor",
                        new HttpContentDecompressor());                 #4
        }
    }
}
```

#1 Add HttpClientCodec as we are in client mode
#2 Add HttpContentDecompressor as the server may send us compressed content
#3 Add HttpServerCodec as we are in server mode
#4 Add HttpContentCompressor as the client may support compression and if so we want to compress it

Compression and dependencies

Be aware that if you NOT use Java 7 (or later) you will need to add an extra dependency to your classpath, as Java itself did not contain all needed classes for compression before Java 7. So if you use any Java version prior make sure you have jzlib[16] on your classpath.

8.2.4 Using HTTPS

As mentioned before in this chapter, you may want to protect your network traffic using encryption. You can do this via HTTPS, which is a piece of cake thanks to the "stackable" `ChannelHandlers` that come with Netty.

All you need to do is to add the `SslHandler` to the mix, as shown in listing 8.5.

[16] http://www.jcraft.com/jzlib/

Listing 8.5 Using HTTPS

```
public class HttpsCodecInitializer extends ChannelInitializer<Channel> {

    private final SSLContext context;
    private final boolean client;

    public HttpsCodecInitializer(SSLContext context, boolean client) {
        this.context = context;
        this.client = client;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        SSLEngine engine = context.createSSLEngine();
        engine.setUseClientMode(client);
        pipeline.addFirst("ssl", new SslHandler(engine));               #1

        if (client) {
            pipeline.addLast("codec", new HttpClientCodec());           #2
        } else {
            pipeline.addLast("codec", new HttpServerCodec());           #3
        }
    }
}
```

#1 Add SslHandler to use HTTPS
#2 Add HttpClientCodec as we are in client mode
#3 Add HttpServerCodec as we are in server mode

Listing 8.5 demonstrates how Netty's pipeline enables flexible applications. This is because to add a new "feature" you often just need to add another ChannelHandler into the ChannelPipeline, like in Listing 8.5 we added the `SslHandler` to support HTTPS while not touching any other code. This is a good example of how handlers enable a powerful approach to network programming.

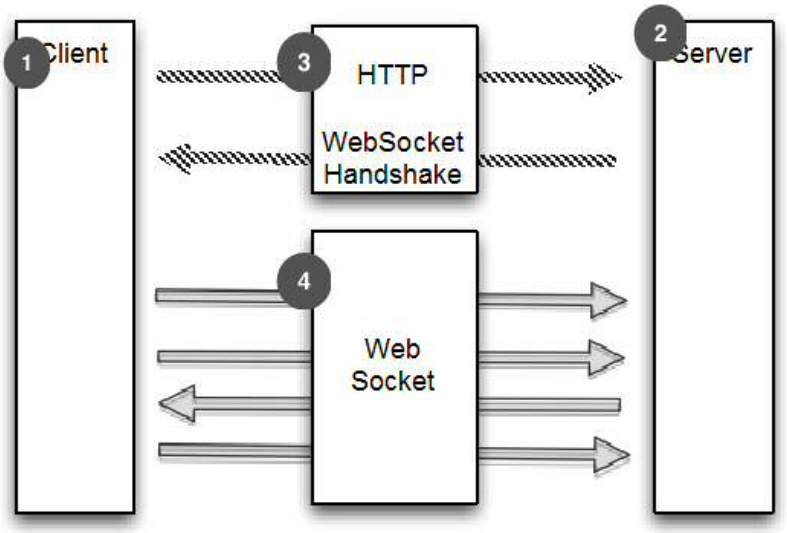
HTTP is one of the most widely used protocols on the World Wide Web and Netty comes equipped with all of the necessary tools to help you develop HTTP-based applications quickly and easily. In the next section we'll look at another popular extension to the HTTP protocol: WebSockets.

8.2.5 Using WebSockets

HTTP is nice, but what do you do if you need to publish information in real time? Previously, you might have used a workaround such as long-polling, but that kind of solution isn't optimal and does no scale well.

This was one of the reasons why the WebSockets[17] specification and implementations were created. WebSockets allow exchanging data in both directions (from client to server and from server to client) without the need of a request-response pattern. While in the early days of WebSockets it was only possible to send text data via WebSockets, this is no longer true. Now you can also send binary data, which makes it possible to use WebSockets more like a normal Socket.

We won't go into too much detail about how WebSockets works as it's out of scope for this book, but the general idea of what happens in a WebSocket communication is shown in figure 8.4.



#1 Client that communicates with server
#2 Server that communicates with client
#3 Client issues WebSocket handshake via HTTP(s) and waits for completion
#4 Connection upgraded to WebSockets

Figure 8.4 WebSocket

As you saw in figure 8.4 the connection starts as HTTP and then "upgrades" to WebSockets. Adding support for WebSockets in your application is just a matter of adding the needed `ChannelHandler` into the `ChannelPipeline`.

When using WebSockets there will be different message types (also called frames) you need to handle. Table 8.3 shows them.

[17] http://tools.ietf.org/html/rfc6455

Table 8.3 WebSocketFrame types

Name	Description
`BinaryWebSocketFrame`	`WebSocketFrame` that contains binary data.
`TextWebSocketFrame`	`WebSocketFrame` that contains text data.
`ContinuationWebSocketFrame`	`WebSocketFrame` that contains text or binary data that belongs to a previous `BinaryWebSocketFrame` or `TextWebSocketFrame`.
`CloseWebSocketFrame`	`WebSocketFrame` that represents a CLOSE request and contains close status code and a phrase.
`PingWebSocketFrame`	`WebSocketFrame` which request the send of a `PongWebSocketFrame`.
`PongWebSocketFrame`	`WebSocketFrame` which is sent as response to a `PingWebSocketFrame`.

As Netty is more often used to implement a Server we'll only have a look at what to use for a WebSocket server. More information for a client can be found in the examples[18] that ship with Netty source code.

Netty offers many ways to use WebSockets, but the easiest one, which works for most users, is using the `WebSocketServerProtocolHandler` when writing a WebSockets server, as shown in listing 8.6. This handles the handshake and also the `CloseWebSocketFrame`, `PingWebSocketFrame`, and `PongWebSocketFrame` for you.

Listing 8.6 Support WebSocket on the server

```
public class WebSocketServerInitializer extends ChannelInitializer<Channel>{
    @Override
    protected void initChannel(Channel ch) throws Exception {
        ch.pipeline().addLast(
                new HttpServerCodec(),
                new HttpObjectAggregator(65536),                        #1
                new WebSocketServerProtocolHandler("/websocket"),       #2
                new TextFrameHandler(),                                 #3
                new BinaryFrameHandler(),                               #4
                new ContinuationFrameHandler());                        #5
    }
```

[18] https://github.com/netty/netty/tree/4.0/example/src/main/java/io/netty/example/http/websocketx/client

```
    public static final class TextFrameHandler extends
            SimpleChannelInboundHandler<TextWebSocketFrame> {
        @Override
        public void channelRead0(ChannelHandlerContext ctx,
                TextWebSocketFrame msg) throws Exception {
            // Handle text frame
        }
    }

    public static final class BinaryFrameHandler extends
            SimpleChannelInboundHandler<BinaryWebSocketFrame> {
        @Override
        public void channelRead0(ChannelHandlerContext ctx,
                BinaryWebSocketFrame msg) throws Exception {
            // Handle binary frame
        }
    }

    public static final class ContinuationFrameHandler extends
            SimpleChannelInboundHandler<ContinuationWebSocketFrame> {
        @Override
        public void channelRead0(ChannelHandlerContext ctx,
                ContinuationWebSocketFrame msg) throws Exception {
            // Handle continuation frame
        }
    }
}
```

#1 Add HttpObjectAggregator as we need aggregated HttpRequests for the handshake
#2 Add WebSocketServerProtocolHandler will handle the upgrade if a request is send to /websocket and also handle the Ping, Pong and Close frames after the upgrade
#3 TextFrameHandler will handle TextWebSocketFrames
#4 BinaryFrameHandler will handle BinaryWebSocketFrames
#5 ContinuationFrameHandler will handle ContinuationWebSocketFrames

If you want to support Secure-WebSocket, it's as easy as adding the `SslHandler` as the first `ChannelHandler` in the `ChannelPipeline`.

For a more complete example of using WebSockets with Netty please see chapter 11 which covers using WebSockets with Netty in much greater detail. This includes building a „real" application that makes use of WebSockets and so the provided Codecs and `ChannelHandler`.

8.2.6 SPDY

SPDY[19] (SPeeDY) is one of the new movements when it comes to HTTP. Google invented SPDY, but you can find many of its "ideas" in the upcoming HTTP 2.0.

The idea of SPDY is to make transfer of content much faster. This is done by:

- Compressing the headers
- Encrypting everything

[19] http://www.chromium.org/spdy

- Multiplexing of connections
- Providing support for different transfer priorities

At the time of writing there are three versions of SPDY in the wild. Those versions are based on:

- Draft 1 – Initial version
- Draft 2 – Contains fixed and new features such as server-push
- Draft 3 – Contains fixes and features such as flow control

Many web browsers support SPDY at the time of writing, including Google Chrome, Firefox, and Opera.

Netty comes with support for Draft 2 and Draft 3 (including 3.1), which are the most used at the moment and should allow you to support most of your users.

For an example how to use SPDY iwith Netty see chapter 12, that provide a full chapter on this topic.

In the last sections of 8.2 you got some overview about the different Codecs and `ChannelHandlers` that Netty provides to support all kind of Protocols related to HTTP(s). This included Codecs for HTTP itself but also protocols like WebSockets and SPDY. In the next section we will look into how you can detect idle connections and handle them to free up resources as soon as possible.

8.3 Handling idle connections and timeouts

Often you need to handle idle connections and timeouts so you can free up resources in a timely manner once they are not needed anymore. Typically you'd send a message, also called a "heartbeat," to the remote peer if it idles too long in order to detect if it's still alive. Another approach is to disconnect the remote peer if it idles too long.

Dealing with idle connections is such a core part of many applications that Netty provides various `ChannelHandler` implementations to handle it. Table 8.4 gives an overview of them.

Table 8.4 ChannelHandlers for handling idle and timeouts

Name	Description
`IdleStateHandler`	`IdleStateHandler` fires an `IdleStateEvent` if the connection idles too long. You can then act on the `IdleStateEvent` by override the `userEventTriggered(…)` method in your `ChannelInboundHandler`.
`ReadTimeoutHandler`	`ReadTimeoutHandler` throws a `ReadTimeoutException` and closes the `Channel` when there is no inbound data received for the timeout. The `ReadTimeoutException` can be detected by overriding the `exceptionCaught(…)` method of your `ChannelHandler`.

`WriteTimeoutHandler`	`WriteTimeoutHandler` throws a `WriteTimeoutException` and closes the `Channel` when there is no inbound data received for the timeout. The `WriteTimeoutException` can be detected by overriding the `exceptionCaught(…)` method of your `ChannelHandler`.

The most used in practice is the `IdleStateHandler`, so let's focus on it.

Listing 8.9 shows how you can use the `IdleStateHandler` to get notified if you haven't received or sent data for 60 seconds. If this is the case, a heartbeat will be written to the remote peer, and if this fails the connection is closed.

Listing 8.9 Send heartbeats on idle

```
public class IdleStateHandlerInitializer extends ChannelInitializer<Channel> {


       @Override
       protected void initChannel(Channel ch) throws Exception {
           ChannelPipeline pipeline = ch.pipeline();
           pipeline.addLast(
                   new IdleStateHandler(0, 0, 60, TimeUnit.SECONDS));      #1
               pipeline.addLast(new HeartbeatHandler());
       }

       public static final class HeartbeatHandler extends ChannelStateHandlerAdapter
          {
           private static final ByteBuf HEARTBEAT_SEQUENCE =
                   Unpooled.unreleasableBuffer(Unpooled.copiedBuffer(
                       "HEARTBEAT", CharsetUtil.ISO_8859_1));              #2

           @Override
           public void userEventTriggered(ChannelHandlerContext ctx,
                   Object evt) throws Exception {
               if (evt instanceof IdleStateEvent) {
                    ctx.writeAndFlush(HEARTBEAT_SEQUENCE.duplicate())
                            .addListener(
                                 ChannelFutureListener.CLOSE_ON_FAILURE);  #3
               } else {
                   super.userEventTriggered(ctx, evt);                     #4
               }
           }
       }
   }
```

#1 Add IdleStateHandler which will fire an IdleStateEvent if the connection has not received or send data for 60 seconds
#2 The heartbeat to send to the remote peer
#3 Send the heartbeat and close the connection if the send operation fails
#4 Not of type IdleStateEvent pass it to the next handler in the ChannelPipeline

In this this section you saw yhow you can use the IdleStateHandler to automatically send a heartbeat message to the remote peer and so detect if it is still alive or not. This allows to free up the resources once the remote-peer is not alive anymore, which means closing the

connection in this case. In the next section we will look into how you can easily handle delimiter-based and length-based protocols.

8.4 Decoding delimiter- and length-based protocols

As you work with Netty, you'll come across delimiter-based and length-based protocols that need to be decoded. This section explains the implementations that come with Netty for the purpose of decoding these kind of protocols.

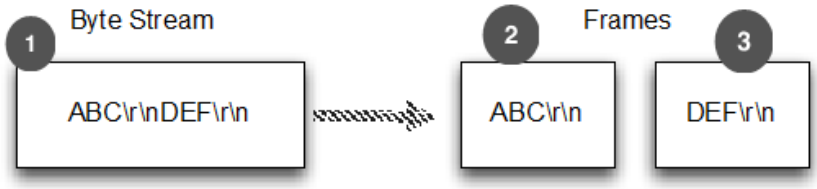
8.4.1 Delimiter-based protocols

A lot of protocols that are specified by RFC's are delimiter-based protocols or build on top of them. Some examples of popular delimiter-based protocols are SMTP[20], POP3[21], IMAP[22], and Telnet[23]. For those, Netty ships with special `ChannelHandlers` that make it easy to extract frames delimited by a sequence.

Table 8.5 ChannelHandler for handling idle and timeouts

Name	Description
`DelimiterBasedFameDecoder`	`Decoder` that extracts frames using a provided delimiter. This a generic decoder that can handle any delimiter.
`LineBasedFrameDecoder`	`Decoder` that extracts frames for the \r \n delimiter. This is faster than `DelimiterBasedFrameDecoder`.

So how does it work? Let's have a look at figure 8.5, which shows how frames are handled when delimited by a "\r\n" sequence (carriage return).



#1 Stream of bytes
#2 First frame which was extracted out of the stream
#3 Second frame which was extracted out of the stream

Figure 8.5 Delimiter terminated frame handling

[20] http://www.ietf.org/rfc/rfc2821.txt
[21] http://www.ietf.org/rfc/rfc1939.txt
[22] http://tools.ietf.org/html/rfc3501
[23] http://tools.ietf.org/search/rfc854

Listing 8.10 shows how you can use the `LineBasedFrameDecoder` to extract the "\r\n" delimited frames.

Listing 8.10 Handling \r\n delimited frames

```
public class LineBasedHandlerInitializer extends ChannelInitializer<Channel> {

       @Override
       protected void initChannel(Channel ch) throws Exception {
           ChannelPipeline pipeline = ch.pipeline();
           pipeline.addLast(new LineBasedFrameDecoder(65 * 1024));         #1
           pipeline.addLast(new FrameHandler());                           #2
       }

       public static final class FrameHandler
               extends SimpleChannelInboundHandler<ByteBuf> {
           @Override
           public void channelRead0(ChannelHandlerContext ctx,
                   ByteBuf msg) throws Exception {                         #3
               // Do something with the frame
           }
       }
   }
```

#1 Add LineBasedFrameDecoder which will extract the frames and forward to the next handler in the pipeline
#2 Add FrameHandler that will receive the frames
#3 Do something with the frame

If your frames are delimited by something other than line breaks, you can use the `DelimiterBasedFrameDecoder` in a similar fashion. You only need to pass the delimiter to the constructor.

Those decoders are also useful if you want to implement your own delimiter-based protocol. Imagine you have a protocol that handles only commands. Those commands are formed out of a name and arguments. The name and the arguments are separated by a whitespace.

Writing a special decoder for this protocol is straightforward if you extend the `LineBasedFrameDecoder`.

Listing 8.11 shows how to do it.

Listing 8.11 Decoder for the command and the handler

```
public class CmdHandlerInitializer extends ChannelInitializer<Channel> {

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(new CmdDecoder(65 * 1024));                    #1
        pipeline.addLast(new CmdHandler());                             #2
    }

    public static final class Cmd {                                     #3
        private final ByteBuf name;
```

```
        private final ByteBuf args;

        public Cmd(ByteBuf name, ByteBuf args) {
            this.name = name;
            this.args = args;
        }

        public ByteBuf name() {
            return name;
        }

        public ByteBuf args() {
            return args;
        }
    }

    public static final class CmdDecoder extends LineBasedFrameDecoder {
        public CmdDecoder(int maxLength) {
            super(maxLength);
        }

        @Override
        protected Object decode(ChannelHandlerContext ctx, ByteBuf buffer)
                throws Exception {
            ByteBuf frame =  (ByteBuf) super.decode(ctx, buffer);       #4
            if (frame == null) {
                return null;                                            #5
            }
            int index = frame.indexOf(frame.readerIndex(),
                    frame.writerIndex(), (byte) ' ');                   #6
            return new Cmd(frame.slice(frame.readerIndex(), index),
                    frame.slice(index +1, frame.writerIndex()));        #7
        }
    }

    public static final class CmdHandler
            extends SimpleChannelInboundHandler<Cmd> {
        @Override
        public void channelRead0(ChannelHandlerContext ctx, Cmd msg)
                throws Exception {
            // Do something with the command                            #8
        }
    }
}
```

#1 Add the CmdDecoder that will extract the command and forward to the next handler in the pipeline
#2 Add CmdHandler that will receive the commands
#3 The Pojo that represents the commands
#4 Extract the ByteBuf frame which was delimited by the \r\n
#5 There was no full frame ready so just return null
#6 Fix the first whitespace as it's the separator of the name and arguments
#7 Construct the command by passing in the name and the arguments using the index
#8 Handle the command

In this section you got a quick introduction into how you can use the DelimiterBasedFrameDecoder / LineBasedFrameDecoder to handle protocols that use any kind

of delimiter to signal a new frame. This kind of protocols are often text based and famous examples are SMTP/POP3/IMAP.

In the next section we will look into handling protocols that encode the actual frame length in the frame header itself.

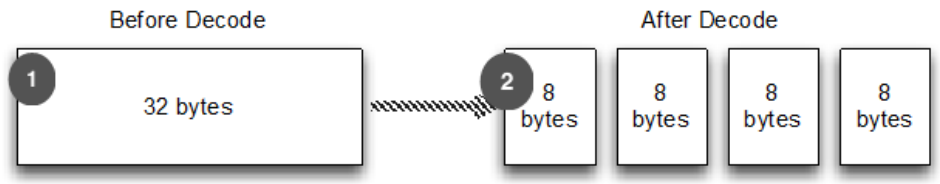
8.4.2 Length-based protocols

Often you find length-based protocols out in the wild. Those do not have a special delimiter to separate frames but use a length to signal how long a frame is. For this case, Netty provides two different decoders that will help you extract frames, as shown in table 8.6.

Table 8.6 Decoders that extract frames based on the length

Name	Description
`FixedLengthFrameDecoder`	`Decoder` that extracts extra frames of the same fixed size. The size is given when construct it and is the same for all frames.
`LengthFieldBasedFrameDecoder`	`Decoder` that extracts frames based on the size that's encoded in the header of the frame. Where exactly the length is contained in the header can be specified via the constructor.

To make it clearer, let's illustrate both of them with a figure. Figure 8.6 shows how the `FixedLengthFrameDecoder` works.



#1 Stream of bytes
#2 Extracted frames of 8 byte size

Figure 8.6 Message of fixed size of 8 bytes

As shown, the `FixedLengthFrameDecoder` extracts frames of a fixed length, which is 8 bytes in this case.

More often you find the case where the size of the frame is encoded in the header. For this purpose you can use the `LengthFieldBasedFrameDecoder`, which will read the length out of the header and extract the frame for the read length.

Figure 8.7 shows how it works.

#1 Length encoded in the header of the frame
#2 Content of the frame with the encoded length
#3 Extracted frame that has the length header stripped

Figure 8.7 Message that has fixed size encoded in the header

The `LengthFieldBasedFrameDecoder` lets you specify where exactly in the frame header the length field is contained and how long it is. It's very flexible, for more information, please refer to the API docs.

Because it's easier to use the `FixedLengthFrameDecoder` as you only need to give it the frame length in the constructor and nothing else, we'll focus on the `LengthFieldBasedFrameDecoder`.

Listing 8.12 shows how you can use the `LengthFieldBasedFrameDecoder` to extract frames whose length is encoded in the first 8 bytes.

Listing 8.12 Decoder for the command and the handler

```
public class LengthBasedInitializer extends ChannelInitializer<Channel> {

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(
                new LengthFieldBasedFrameDecoder(65 * 1024, 0, 8));     #1
        pipeline.addLast(new FrameHandler());                           #2
    }

    public static final class FrameHandler
            extends SimpleChannelInboundHandler<ByteBuf> {
        @Override
        public void channelRead0(ChannelHandlerContext ctx,
                ByteBuf msg) throws Exception {
            // Do something with the frame                              #3
        }
    }
}
```

#1 Add LengthFieldBasedFrameDecoder to extract the frames based on the encoded length in the header
#2 Add the FrameHandler to handle the frames
#3 Do something with the frames

In section 8.4 you learned how you can make use of Netty's provided codecs to support different protocols that either are delimiter based or length based. As many different protocols fit into one of these it may come in very handy at some point.

In the next section we look into another need that ofter arise; Transmitting files or other big chunks of data.

8.5 Writing big data

Writing big chunks of data in an efficient way is often a problem when it comes to asynchronous frameworks, because you need to stop writing if the network is saturated or you'd otherwise fill up the memory and have a good chance of getting an `OutOfMemoryError`. The cause of this is that writes are non-blocking which means the write call will not "block" at all and so directly return even if the data can not directly be written out and just notify the `ChannelFuture` once it is done. This means you need to be more carefully when write big masses of data as the remote peer may be on a slow connection and thus it may not be able to write off data fast enough and so free up the memory.

Netty allows you to write the contents of a file using a zero-memory-copy approach, which means that it handles all the shuffling from the Filesystem to the network stack in the kernel space, which eliminates copies and switching from one space to the other. This allows for maximum performance. All of this happens in the core of Netty, so you don't need to worry about it. All you need to do is using a `FileRegion` to write a files content.

Writing a `file's` contents via zero-memory-copy works by writing a `DefaultFileRegion` to the `Channel`, `ChannelHandlerContext`, or `ChannelPipeline`, as shown in listing 8.13.

Listing 8.13 Transfer file content with `FileRegion`

```
FileInputStream in = new FileInputStream(file);                          #1
FileRegion region = new DefaultFileRegion(
        in.getChannel(), 0, file.length());                              #2

channel.writeAndFlush(region)
        .addListener(new ChannelFutureListener() {                       #3
    @Override
    public void operationComplete(ChannelFuture future)
            throws Exception {
        if (!future.isSuccess()) {
            Throwable cause = future.cause();                            #4
            // Do something
        }
    }
});
```

#1 Get FileInputStream on file
#2 Create a new DefaultFileRegion for the file starting at offset 0 and ending at the end of the file
#3 Send the DefaultFileRegion and register a ChannelFutureListener
#4 Handle failure during send

This only works if the content of the Fille needs to transfer as it is (no in program modification); if not, Netty needs to copy the data into the user space to perform the operations on it. For situations other then this you can use `ChunkedWriteHandler` which we will dive into now.

But what do you do if you don't want to send a file or some other big chunk of data?

Netty comes with a special handler, called `ChunkedWriteHandler` that allows you to write big chunks of data by processing `ChunkedInput` implementations.

ChunkedInput

As `ChunkedInput` is only an interface you are free to implement your own `ChunkedInput` and be able to write it to a Channel as soon as you have the `ChunkedWriteHandler` in your `ChannelPipeline`.

Table 8.7 shows the `ChunkedInput` implementations that are part of Netty.

Table 8.7 Provided ChunkedInput implementations

Name	Description
`ChunkedFile`	`ChunkedInput` implementation that allows you to write a file (use only if your platform doesn't support zero-memory-copy or you need to transform some modification on the bytes like compression or encryption).
`ChunkedNioFile`	`ChunkedInput` implementation that allows you to write a file (use only if your platform doesn't support zero-memory-copy or you need to transform some modification on the bytes like compression or encryption).
`ChunkedNioStream`	`ChunkedInput` implementation that allows you to transfer content from a `ReadableByteChannel`.
`ChunkedStream`	`ChunkedInput` implementation that allows you to transfer content from an `InputStream`.

The `ChunkedStream` implementation is the most used in practice, so listing 8.14 is covering it too.

Listing 8.14 Transfer file content with `FileRegion`

```
public class ChunkedWriteHandlerInitializer
        extends ChannelInitializer<Channel> {
    private final File file;

    public ChunkedWriteHandlerInitializer(File file) {
        this.file = file;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(new ChunkedWriteHandler());                    #1
        pipeline.addLast(new WriteStreamHandler());                     #2
    }

    public final class WriteStreamHandler
            extends ChannelInboundHandlerAdapter {

        @Override
        public void channelActive(ChannelHandlerContext ctx)
                throws Exception {
            super.channelActive(ctx);
            ctx.writeAndFlush(
                    new ChunkedStream(new FileInputStream(file)));    #3
        }}
}
```

#1 Add ChunkedWriteHandler to handle ChunkedInput implementations
#2 Add WriteStreamHandler to write a ChunkedInput
#3 Write the content of the file via a ChunkedStream once the connection is established (we use a FileInputStream only for demo purposes, any InputStream works)

In this section you learned how you can transfer files in an efficient way using Netty by make use of zero-memory-copy. Beside this you also learned how you can write other big-data by using `ChunkedWriteHandler` without risking an `OutOfMemoryError`.
In the next section we will look into the various way to serialize Pojos and how Netty allows you to do so by make use of different serialization libraries.

8.6 Serializing data

When you want to transfer your POJOs over the network and act on them on the remote peer, Java offers the `ObjectOutputStream` and `ObjectInputStream` as its serialization marker interface. But there are also other options for doing this which are more performant.

POJO – What's that ?
POJO stands for Plain-Old-Java-Object and is in fact just some simple Java Object.

This section shows what Netty offers out of the box for this purpose.

8.6.1 Serialization via plain JDK

If you need to talk with other peers that use `ObjectOutputStream` and `ObjectInputStream`, and you need to keep compatibility or you don't want to have an external dependency, JDK Serialization[24] is the choice.

Table 8.8 list the serialization implementations for this.

Table 8.8 Provided JDK Serialization codec

Name	Description
`CompatibleObjectDecoder`	Uses plain JDK Serialization for decoding and can be used with other peers that don't use Netty and just use the `ObjectInputStream` / `ObjectOutputStream` that is provided by the JDK.
`CompatibleObjectEncoder`	Uses plain JDK Serialization for encoding and can be used with other peers that don't use Netty and just use the `ObjectInputStream` / `ObjectOutputStream` that is provided by the JDK.
`CompactObjectDecoder`	Uses custom serialization for decoding on top of JDK Serialization. Only use this if you want to gain speed but not be able to have a dependency, in which case the other serialization implementations are preferable.
`CompactObjectEncoder`	Uses custom serialization for encoding on top of JDK Serialization. Only use this if you want to gain speed but not be able to have a dependency, in which case the other serialization implementations are preferable.

8.6.2 Serialization via JBoss Marshalling

If you can take an extra dependency, JBoss Marshalling[25] is the way to go. It's up to three times faster than JDK Serialization and more compact.

Netty comes with either a compatible implementation that can be used with other peers using JDK Serialization or one that can be used for maximum speed if the other peers also use JBoss Marshalling.

Table 8.9 explains the JBoss Marshalling decoders and encoders.

[24] http://docs.oracle.com/javase/7/docs/technotes/guides/serialization/
[25] https://www.jboss.org/jbossmarshalling

Table 8.9 JBoss Marshalling codec

Name	Description
`CompatibleMarshallingDecoder`	Uses JDK Serialization for decoding and so can be used with other peers that don't use Netty and just use the `ObjectInputStream` / `ObjectOutputStream` that is provided by the JDK.
`CompatibleMarshallingEncoder`	Uses JDK Serialization for encoding and so can be used with other peers that don't use Netty and just use the `ObjectInputStream` / `ObjectOutputStream` that is provided by the JDK.
`MarshallingDecoder`	Uses custom serialization for decoding.
`MarshallingEncoder`	Uses custom serialization for encoding.

Listing 8.15 shows how you can make use of the previous shown `MarshallingDecoder` and `MarshallingEncoder`. For this we again just setup the `ChannelPipeline` accordantly.

Listing 8.15 Using JBoss Marshalling

```
public class MarshallingInitializer extends ChannelInitializer<Channel> {

    private final MarshallerProvider marshallerProvider;
    private final UnmarshallerProvider unmarshallerProvider;

    public MarshallingInitializer(
            UnmarshallerProvider unmarshallerProvider,
            MarshallerProvider marshallerProvider) {
        this.marshallerProvider = marshallerProvider;
        this.unmarshallerProvider = unmarshallerProvider;
    }

    @Override
    protected void initChannel(Channel channel) throws Exception {
        ChannelPipeline pipeline = channel.pipeline();
        pipeline.addLast(new MarshallingDecoder(unmarshallerProvider));
        pipeline.addLast(new MarshallingEncoder(marshallerProvider));
        pipeline.addLast(new ObjectHandler());
    }

    public static final class ObjectHandler
            extends SimpleChannelInboundHandler<Serializable> {
        @Override
        public void channelRead0(ChannelHandlerContext channelHandlerContext,
      Serializable serializable) throws Exception {
            // Do something
        }
    }
}
```

#1 Add ChunkedWriteHandler to handle ChunkedInput implementations

#2 Add WriteStreamHandler to write a ChunkedInput
#3 Write the content of the file via a ChunkedStream once the connection is established (we use a FileInputStream only for demo purposes, any InputStream works)

8.6.3 Serialization via ProtoBuf

The last solution for serialization that ships with Netty is a codec that makes use of ProtoBuf[26].

ProtoBuf was open-sourced by Google in the past and is a way to encode and decode structured data in a compact and efficient way. It comes with different bindings for all sorts of programming languages, making it a good fit for cross-language projects.

Table 8.10 shows the `ChannelHandler` implementations for ProtoBuf support.

Table 8.10 Protobuf codec

Name	Description
`ProtobufDecoder`	Decode message via ProtoBufs.
`ProtobufEncoder`	Encode message via ProtoBufs.
`ProtobufVarint32FrameDecoder`	A decoder that splits the received `ByteBufs` dynamically by the value of the Google Protocol `BuffersBase` 128 Varints integer length field in the message.

Again using Protobuf is just a matter of adding the right `ChannelHandler` in the `ChannelPipeline` as shown in Listing 8.16.

Listing 8.16 Using Google Protobuf

```
public class ProtoBufInitializer extends ChannelInitializer<Channel> {

    private final MessageLite lite;

    public ProtoBufInitializer(MessageLite lite) {
        this.lite = lite;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(new ProtobufVarint32FrameDecoder());          #1
        pipeline.addLast(new ProtobufEncoder());                       #2
        pipeline.addLast(new ProtobufDecoder(lite));                   #3
        pipeline.addLast(new ObjectHandler());                         #4
    }

    public static final class ObjectHandler
            extends SimpleChannelInboundHandler<Object> {
        @Override
```

[26] https://code.google.com/p/protobuf/

```
        public void channelRead0(ChannelHandlerContext ctx, Object msg)
                throws Exception {
            // Do something with the object
        }
    }
}
```

#1 Add ProtobufVarint32FrameDecoder to break down frames
#2 Add ProtobufEncoder to handle encoding of messages
#3 Add ProtobufDecoder that decodes to messages
#4 Add ObjectHandler to handle the decoded messages

In this last section of this chapter you learned how you can make use of various different serialization frameworks by using some of the provided decoders/encoders of Netty. This includes the usage of "stock" JDK serialization but also showed how you can use others like JBoss Marshalling or Google Protobuf.

8.7 Summary

This chapter gave you a brief overview of the included codecs and handlers that you can reuse in your application. This should help you to find your way through the provided pieces and prevent you from reinventing the wheel.

You also saw how you're able to combine different codecs/handlers to build up the logic that you need, and you learned how you can extend provided implementations to adjust them for the needed logic.

Another advantage to reusing what Netty provides is that those parts are well tested by many users and should be robust.

This chapter only covered the most-used codecs and handlers that are bundled with Netty. There are more; but to cover all of them would take a much bigger chapter. Please refer to Netty's API docs to get an overview of the rest of them.

In the next chapter we'll look at how you bootstrap your server and combine handlers to get things running.

9

Bootstrapping Netty applications

This chapter covers

- Bootstrapping clients and servers
- Bootstrapping clients from within a channel
- Adding ChannelHandlers
- Using ChannelOptions and attributes

In previous chapters you learned how to write your own `ChannelHandler` and codecs and add them to the `ChannelPipeline` of the `Channel`. Now the question is: How do you assemble all of this?

You can use bootstrapping. Netty provides you with an easy and unified way to bootstrap your servers and clients. What is bootstrapping, and how does it fit into Netty? Bootstrapping is the process by which you configure your Netty server and client applications. Bootstraps allow these applications to be easily reusable.

Bootstraps are available for both client and server Netty Applications. The purpose of each is to simplify the process of combining all of the components we've discussed previously (channels, pipeline, handlers, and so on). Bootstraps also provide Netty with a mechanism that ties in these components and makes them all work in the background.

This chapter will look specifically at how the following pieces fit together in a Netty application:

- `EventLoopGroup` type
- `Channel` type
- `ChannelOptions`
- `ChannelHandler` that will be called once `Channel` is registered
- Special attributes that are added on the `Channel`
- Local and remote address
- Binding and connecting

Once you know how to use the various bootstraps, you can use them to configure the server and client. You'll also learn when it makes sense to share a bootstrap instance and why, giving you the last missing piece to be able to assemble all the parts we learned about in previous chapters and allowing you to use Netty in your next application.

9.1 Different types of bootstrapping

Netty includes two different types of bootstraps, named `Bootstrap` and `ServerBootstrap`. One is used for server-like `Channels` that accept connections and create "child" `Channels` for the accepted connections. The second is for "client-like Channels" that don't accept new connections and process everything in the "parent" `Channel`.

The two different bootstrap implementations extend from one super-class named `AbstractBootstrap`. Figure 9.1 shows the hierarchy.

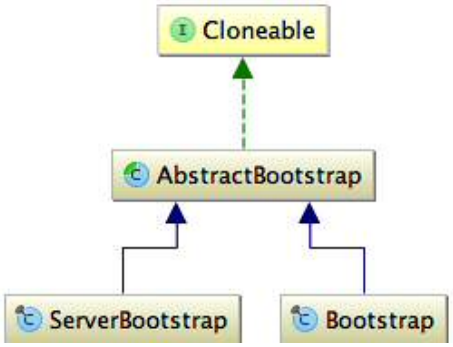


Figure 9.1 Bootstrap hierarchy

Many of the topics we've covered in previous chapters apply to both client and servers. In order to provide a common ground for the relationship between clients and servers, Netty uses the `AbstractBootstrap` class. By having a common ancestor, the client and server bootstraps discussed in this chapter are able to reuse common functionality without duplicating code or logic.

It's often the case that multiple channels are needed with the same or very similar settings. Instead of creating a new bootstrap for each of these channels, Netty has made the `AbstractBootstrap` cloneable. This means that a deep clone of an already configured bootstrap will return another bootstrap which is reusable without having to be reconfigured. Netty's clone operation only makes a shallow copy of the bootstrap's `EventLoopGroup`, which means the `EventLoopGroup` is shared between all of the cloned `Channels`. This is a good thing, as it's often the case that the cloned `Channels` are short-lived, for example, a `Channel` created to make an HTTP request.

The rest of the chapter will focus on `Bootstrap` and `ServerBootstrap`. Let's have a stab at `Bootstrap` first, as it's not as complex as `ServerBootstrap`.

9.2 Bootstrapping clients and connectionless protocols using Bootstrap

Whenever you need to bootstrap a client or some connectionless protocol you'll need to use the `Bootstrap` class. Working with it is easy, as we'll learn shortly. This section includes informations on the various methods for bootstrapping clients, explains how to bootstrap a client, and discusses how to choose a compatible channel implementation for the client.

9.2.1 Methods for bootstrapping clients

Before we go into much detail let's look at the various methods it provides. Table 9.1 gives an overview of them.

Table 9.1 Methods to bootstrap

Name	Description
`group(...)`	Set the `EventLoopGroup`, which should be used by the bootstrap. This `EventLoopGroup` is used to serve the I/O of the `Channel`.
`channel(...)` `channelFactory(...)`	The class of the `Channel` to instance. If the channel can't be created via a no-args constructor, you can pass in a `ChannelFactory` for this purpose.
`localAddress(...)`	The local address the `Channel` should be bound to. If not specified, a random one will be used by the operating system. Alternatively, you can specify the `localAddress` on `bind(...)` or `connect(...)`
`option(...)`	`ChannelOptions` to apply on the `Channel's` `ChannelConfig`. Those options will be set on the `Channel` on the `bind(…)` or `connect(…)` method depending on what is called first. Changing them after calling those methods has no effect. Which `ChannelOptions` are supported depends on the actual `Channel` you'll use. Please refer to the API docs of the `ChannelConfig` that's used by it.
`attr(...)`	Allow applying attributes on the `Channel`. Those options will be set on the channel on the `bind(…)` or `connect(…)` method, depending on what is called first. Changing them after calling those methods has no effect.
`handler(...)`	Set the `ChannelHandler` that's added to the `ChannelPipeline` of the channel and so receive notification for events..
`clone()`	Clone the `Bootstrap` to allow it to connect to a different remote peer with the same settings as on the original `Bootstrap`.
`remoteAddress(...)`	Set the remote address to connect to. Alternatively, you can also specify it when calling `connect(...)`.
`connect(...)`	Connect to the remote peer and return a `ChannelFuture`, which is notified once the connection operation is complete. This can either be because it was successful or because of an error. Be aware that this method will also bind the `Channel` before.
`bind(...)`	Bind the channel and return a `ChannelFuture`, which is notified once the bind operation is complete. This can either be because it was successful or because of an error. Be aware that you will need to call `Channel.connect(…)` to finally connect to the remote peer after the bind operation successed.

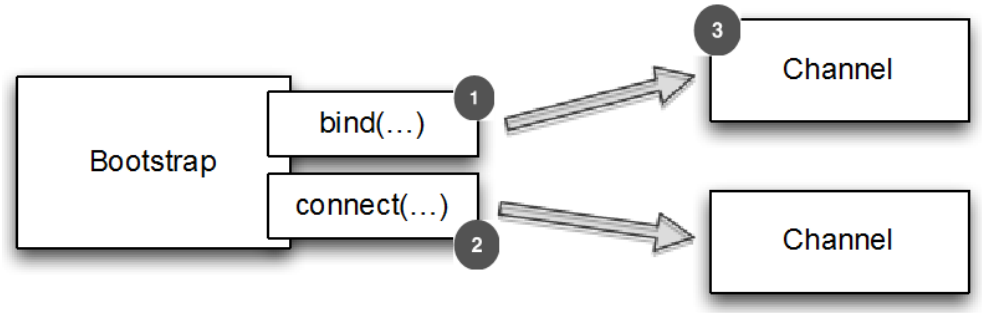
The next section explains how bootstrapping a client works and gives an example you can follow.

9.2.2 How to bootstrap a client

Now that you've been introduced to the different ways to bootstrap a client, I can show you how it works.

The `Bootstrap` is responsible for client and/or connectionless-based `Channels`, so it will create the channel after `bind(...)` or `connect(...)` is called.

Figure 9.2 shows how this works.



#1 Bootstrap will create a new channel when calling bind(..) you will then later call connect(..) on the channel itself to establish the connection
#2 Bootstrap will create a new channel when calling connect(...)
#3 The newly created channel

Figure 9.2 Bootstrap

Now that we know about all of the `Bootstrap` methods, we'll look at how you use one in action. Listing 9.1 shows how you bootstrap a client that's using the NIO TCP transport.

Transports

For other Transports please refer to Chapter 4.

Listing 9.1 Bootstrapping a client

```
EventLoopGroup group = new NioEventLoopGroup();
Bootstrap bootstrap = new Bootstrap();                                  #1
bootstrap.group(group)                                                  #2
        .channel(NioSocketChannel.class)                                #3
        .handler(new SimpleChannelInboundHandler<ByteBuf>() {           #4
                 @Override
             protected void channeRead0(
                     ChannelHandlerContext channelHandlerContext,
                     ByteBuf byteBuf) throws Exception {
                        System.out.println("Reveived data");
                        byteBuf.clear();
                    }
                });
ChannelFuture future = bootstrap.connect(
        new InetSocketAddress("www.manning.com", 80));                  #5
    future.addListener(new ChannelFutureListener() {
        @Override
```

```
        public void operationComplete(ChannelFuture channelFuture)
                throws Exception {
            if (channelFuture.isSuccess()) {
                System.out.println("Connection established");
            } else {
                System.err.println("Connection attempt failed");
                channelFuture.cause().printStackTrace();
            }
        }
    });
```

#1 Create a new bootstrap to create new client channels and connect them
#2 Specify the EventLoopGroup to get EventLoops from and register with the channels
#3 Specify the channel class that will be used to instance
#4 Set a handler which will handle I/O and data for the channel
#5 Connect to the remote host with the configured bootstrap

As you may have noticed, all of the methods that don't finalize the `Bootstrap` by either binding or connecting return a reference to the `Bootstrap` itself. This allows for method chaining and gives you a DSL-like way to operate on it.

9.2.3 Choosing compatible channel implementations

The `Channel` implementation and the `EventLoop` that are processed by the `EventLoopGroup` must be compatible. Which `Channel` is compatible to which `EventLoopGroup` can be found in the API docs. As a rule of thumb, you'll find the compatible pairs (`EventLoop` and `EventLoopGroup`) in the same package as the `Channel` implementation itself. For example, you'd use the `NioEventLoop`, `NioEventLoopGroup`, and `NioServerSocketChannel` together. Notice these are all prefixed with "Nio". You wouldn't, however, substitute any of these for another implementation with another prefix such as "Oio", that is, `OioEventLoopGroup` with `NioServerSocketChannel` would be incompatible, for example.

EventLoop and EventLoopGroup

Remember the `EventLoop` that is assigned to the `Channel` is responsible to handle all the operations for the `Channel`. Which means whenever you execute a method that returns a `ChannelFuture` it will be executed in the `EventLoop` that is assigned to the `Channel`.

The `EventLoopGroup` contains a number of `EventLoops` and is responsible to assign an `EventLoop` to the `Channel` during it's registration.

Chapter 15 covers this in detail.

If you try to use an incompatible `EventLoopGroup` it will fail, as shown in listing 9.2.

Listing 9.2 Bootstrap client with incompatible `EventLoopGroup`

```
EventLoopGroup group = new NioEventLoopGroup();
Bootstrap bootstrap = new Bootstrap();                                  #1
bootstrap.group(group)                                                  #2
        .channel(OioSocketChannel.class)                                #3
        .handler(new SimpleChannelInboundHandler<ByteBuf>() {           #4
                @Override
                protected void channelRead0(
                    ChannelHandlerContext channelHandlerContext,
                            ByteBuf byteBuf) throws Exception {
                        System.out.println("Reveived data");
                        byteBuf.clear();
                    }
                });
ChannelFuture future = bootstrap.connect(
        new InetSocketAddress("www.manning.com", 80));                  #5
future.syncUninterruptibly();
```

#1 Create new bootstrap to create new client channels and connect them
#2 Specify the EventLoopGroup that will be used to get EventLoops from and register with the channels
#3 Specify the channel class that will be used to instance. You may notice that we use the Nio version for the EventLoopGroup and Oio for the channel
#4 Set a handler which will handle I/O and data for the channel
#5 Try to connect to the remote peer. This will throw an IllegalStateException as NioEventLoopGroup isn't compatible with OioSocketChannel

Setting up an implementation with an incompatible `EventLoopGroup` will ultimately fail with an `IllegalStateException`, as shown in listing 9.3.

Listing 9.3 `IllegalStateException` thrown because of invalid configuration

```
Exception in thread "main" java.lang.IllegalStateException: incompatible event loop
      type: io.netty.channel.nio.NioEventLoop                           #1
      at
      io.netty.channel.AbstractChannel$AbstractUnsafe.register(AbstractChannel.java:5
      71)
      at
      io.netty.channel.SingleThreadEventLoop.register(SingleThreadEventLoop.java:57)
      at
      io.netty.channel.MultithreadEventLoopGroup.register(MultithreadEventLoopGroup.j
      ava:48)
      at
      io.netty.bootstrap.AbstractBootstrap.initAndRegister(AbstractBootstrap.java:298
      )
      at io.netty.bootstrap.Bootstrap.doConnect(Bootstrap.java:133)
      at io.netty.bootstrap.Bootstrap.connect(Bootstrap.java:115)
      at
      com.manning.nettyinaction.chapter9.InvalidBootstrapClient.bootstrap(InvalidBoot
      strapClient.java:30)
      at
      com.manning.nettyinaction.chapter9.InvalidBootstrapClient.main(InvalidBootstrap
      Client.java:36)
      at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
      at
      sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
      at
      sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.ja
```

```
va:43)
at java.lang.reflect.Method.invoke(Method.java:601)
at com.intellij.rt.execution.application.AppMain.main(AppMain.java:120)
```

#1 IllegalStateException tells you the configuration is not compatible

Use caution when constructing new bootstrap instances. Beside this there are other situations where an `IllegalStateException` will be thrown. This is when you haven't specified all the needed parameters before calling `bind(…)` or `connect(…)`.

The required parameters are:

- `group(...)`
- `channel(...)` or `channnelFactory(...)`
- `handler(...)`

Once these parameters are provided, your application can make full use of the Netty architecture as discussed so far. Pay special attention to the `handler(...)` method, as the channel pipeline needs to be configured appropriately as discussed in previous chapters.

9.3 Bootstrapping Netty servers with ServerBootstrap

After seeing how you can bootstrap clients and connectionless-based `Channels`, it's time to see how to bootstrap a server. You'll see it's quite similar to and also shares some logic with bootstrapping clients. This section includes information on the various methods for bootstrapping servers and explains how to bootstrap a server.

9.3.1 Methods for bootstrapping servers

Again, let's first look over the provided methods of `ServerBootstrap`.

Table 9.2 Methods of ServerBootstrap

Name	Description
`group(...)`	Set the `EventLoopGroup`, which should be used by the `ServerBootstrap`. This `EventLoopGroup` is used to serve the I/O of the `ServerChannel` and accepted `Channels`.
`channel(...)` `channelFactory(...)`	The class of the `ServerChannel` to instance. If the channel can't be created via a no-args constructor, you can pass in a `ChannelFactory` for this purpose.
`localAddress(...)`	The local address the `ServerChannel` should be bound to. If not specified, a random one will be used by the operating system. Alternatively, you can specify the `localAddress` on `bind(...)` or `connect(...)`
`option(...)`	`ChannelOptions` to apply on the `ServerChannel` `ChannelConfig`. Those options will be set on the channel on the bind or connect method depending on what's called first. Changing them after calling those methods has no effects. Which `ChannelOptions` are supported depends on the

	actual channel you use. Please refer to the API docs of the `ChannelConfig` that is used.
`childOption(...)`	`ChannelOptions` to apply on the accepted `Channels` `ChannelConfig`. Those options will be set on the channels once accepted. Which `ChannelOptions` are supported depends on the actual channel you use. Please refer to the API docs of the `ChannelConfig` that is used.
`attr(...)`	Allow applying attributes on the `ServerChannel`. Those attributes will be set on the channel on the `bind(…)`. Changing them after calling `bind(…)` has no effects
`childAttr(...)`	Allow applying attributes on the accepted `Channels`. Those attributes will be set on the `Channel` once accepted. Changing them after calling those methods has no effects.
`handler(...)`	Set the `ChannelHandler` that is added to the `ChannelPipeline` of the `ServerChannel` and receive notification for events. You will not often specify one here. More important is the `childHandler(...)` when working with `ServerBootstrap`.
`childHandler(...)`	Set the `ChannelHandler` that's added to the `ChannelPipeline` of the accepted `Channels` and receive notification for events. The difference is between `handler(…)` and `childHandler(…)` is that `handler(…)` allows to add a handler which is processed by the "accepting" `ServerChannel`, while `childHandler(…)` allows to add a handler which processed by the "accepted" `Channel`. The accepted `Channel` represents here a bound Socket to a remote peer.
`clone()`	Clone the `ServerBootstrap` and allow using it to connect to a different remote peer with the same settings as on the original `ServerBoostrap`.
`bind(...)`	Bind the `ServerChannel` return to a `ChannelFuture`, which is notified once the connection operation is complete. This can either be because it was successful or because of an error.

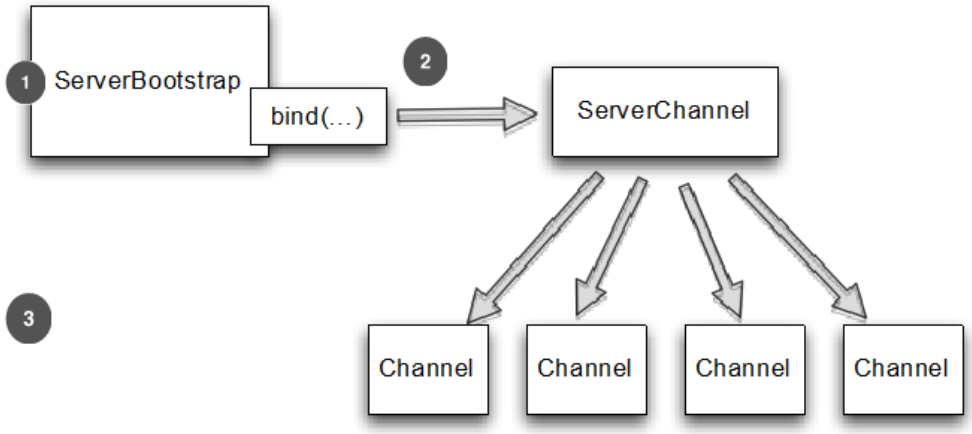
The next section explains how bootstrapping a server works and gives an example you can follow.

9.3.2 How to bootstrap a server

As you may have noticed, the methods in the previous section are similar to what you saw in the bootstrap class.

There is only one difference, which makes a lot of sense once you think about it. While `ServerBootstrap` has `handler(...)`, `attr(...)`, and `option(...)` methods, it also offers those with the child prefix. This is done as the `ServerBootstrap` bootstraps `ServerChannel` implementations, which are responsible for creating child `Channels`. Those `Channels` represent the accepted connections. `ServerBootstrap` offer the child* methods in order to make applying settings on accepted `Channels` as easy as possible.

Figure 9.3 shows more detail about how the `ServerBootstrap` creates the `ServerChannel` on `bind(..)` and this `ServerChannel` manages the child `Channels`.



#1 Bootstrap will create a new channel when calling bind(..) . This channel will then accept child channels once the bind is successful
#2 Accept new connections and create child channels that will serve an accepted connection
#3 Channel for an accepted connection

Figure 9.3 Logic of ServerBootstrap

Remember the child* methods will operate on the child `Channels`, which are managed by the `ServerChannel`.

To make its use clearer, let's look at listing 9.4, which shows how to use `ServerBootstrap`, which will create a `NioServerSocketChannel` instance once `bind(...)` is called. This `NioServerChannel` is responsible for accepting new connections and creating `NioSocketChannel` instances for them.

Listing 9.4 Bootstrapping a server

```
NioEventLoopGroup group = new NioEventLoopGroup();
ServerBootstrap bootstrap = new ServerBootstrap();                    #1
bootstrap.group(group)                                                #2
        .channel(NioServerSocketChannel.class)                        #3
        .childHandler(new SimpleChannelInboundHandler<ByteBuf>() {    #4

           @Override
           protected void channelRead0(ChannelHandlerContext ctx,
                ByteBuf byteBuf) throws Exception {
                System.out.println("Reveived data");
                byteBuf.clear();
           }
    });
ChannelFuture future = bootstrap.bind(new InetSocketAddress(8080));   #5
future.addListener(new ChannelFutureListener() {
```

```
@Override
public void operationComplete(ChannelFuture channelFuture)
        throws Exception {
    if (channelFuture.isSuccess()) {
        System.out.println("Server bound");
    } else {
        System.err.println("Bound attempt failed");
        channelFuture.cause().printStackTrace();
    }
}
});
```

#1 Create a new ServerBootstrap to create new SocketChannel channels and bind them
#2 Specify the EventLoopGroups that will be used to get EventLoops from and register with the ServerChannel and the accepted channels
#3 Specify the channel class that will be used to instance
#4 Set a child handler which will handle I/O and data for the accepted channels
#5 Bind the channel as it's connectionless with the configured bootstrap

In this section you learned how you use `Bootstrap` for bootstrapping client channels. In the next section we will look into how you can not only use the learned technics but even optimize them if you need to bootstrap from within a `ChannelHandler`.

9.4 Bootstrapping clients from within a channel while share the EventLoop

Sometimes situations exist when you need to bootstrap a client `Channel` from within another `ChannelHandler`, for example, if you're writing a proxy or need to retrieve data from other systems. The case where you may need to fetch data from other systems is common since many Netty applications must integrate with an organization's existing systems. The integration could simply be to provide a point where the Netty application authenticates with an internal system and then queries a database.

Sure, you could create a new `Bootstrap` and use it as described in section 9.2.1. In fact, there's nothing wrong with that, it's just not as efficient as it could be, because you'll use another `EventLoop` for the newly created client `Channel`, and if you need to exchange data between your accepted `Channel` and the client `Channel` you'll have to do context-switching between `Threads`.

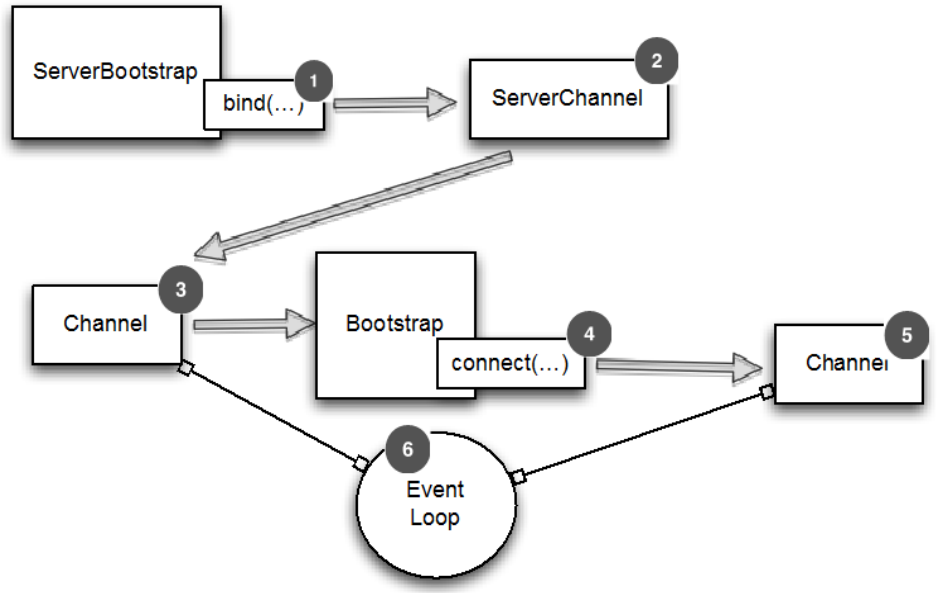
Fortunately Netty helps you optimize this by allowing you to pass in the `EventLoop` of the accepted channel to the `group(...)` method of the bootstrap, thereby allowing the client channel to operate on the same `EventLoop`. This eliminates all extra context switching and works because `EventLoop` extends `EventLoopGroup`. We will reer to this as „sharing" throughout" the rest of the section.

Besides eliminating context-switching, you are able to use the `Bootstrap` without needing to create more threads under the hood.

Why share the EventLoop?

When you share the `EventLoop` you can be sure all Channels that are assigned to the EventLoop are using the same Thread. Remember an `EventLoop` is assigned to a `Thread` that executes the operations of it.

Because of using the same `Thread` there is no context-switching evolved and thus less overhead.



#1 Bootstrap will create a new channel when calling bind(..) . This channel will then accept child channels once the bind is successful
#2 Accept new connections and create child channels that will serve a accepted connection
#3 Channel for an accepted connection
#4 Bootstrap which is created by the channel itself and create a new channel once the connect operation is called
#5 The newly created channel which is connected to the remote peer
#6 EventLoop which is shared between the channel which was created after accept and the new one that was created by connect

Figure 9.4 Share EventLoop between Channels with ServerBootstrap and Bootstrap

Sharing the `EventLoop`, as shown in figure 9.4, only needs some special attention while setting the EventLoop on the bootstrap via the `Boostrap.eventLoop(...)` method.

Listing 9.5 shows exactly what you need to do to handle this specific case.

Listing 9.5 Bootstrapping a server

```
ServerBootstrap bootstrap = new ServerBootstrap();                          #1
bootstrap.group(new NioEventLoopGroup(), new NioEventLoopGroup())           #2
        .channel(NioServerSocketChannel.class)                              #3
    .childHandler(new SimpleChannelInboundHandler<ByteBuf>() {              #4
        ChannelFuture connectFuture;

        @Override
        public void channelActive(ChannelHandlerContext ctx)
                throws Exception {
            Bootstrap bootstrap = new Bootstrap();                          #5
            bootstrap.channel(NioSocketChannel.class)                       #6
                     .handler(
                        new SimpleChannelInboundHandler<ByteBuf>() {        #7
                            @Override
                            protected void channelRead0(
                                    ChannelHandlerContext ctx,
                                    ByteBuf in) throws Exception {
                                System.out.println("Reveived data");
                                 in.clear();
                            }
                         });

            bootstrap.group(ctx.channel().eventLoop());                     #8
            connectFuture = bootstrap.connect(
                    new InetSocketAddress("www.manning.com", 80));          #9
            }

            @Override
            protected void channelRead0(ChannelHandlerContext channelHandlerContext,
      ByteBuf byteBuf) throws Exception {
                if (connectFuture.isDone()) {
                    // do something with the data                          #10
                }
            }
        });

ChannelFuture future = bootstrap.bind(new InetSocketAddress(8080));         #11
future.addListener(new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture channelFuture)
            throws Exception {
        if (channelFuture.isSuccess()) {
            System.out.println("Server bound");
        } else {
            System.err.println("Bound attempt failed");
            channelFuture.cause().printStackTrace();
        }
    }
});
```

#1 Create a new ServerBootstrap to create new SocketChannel channels and bind them
#2 Specify the EventLoopGroups to get EventLoops from and register with the ServerChannel and the accepted channels
#3 Specify the channel class which will be used to instance
#4 Set a handler which will handle I/O and data for the accepted channels
#5 Create a new bootstrap to connect to remote host
#6 Set the channel class

#7 Set a handler to handle I/O
#8 Use the same EventLoop as the one that's assigned to the accepted channel to minimize context-switching and so on
#9 Connect to remote peer
#10 Do something with the data if the connect is complete, for example, proxy it
#11 Bind the channel as it's connectionless with the configured bootstrap

This section showed how to share the `EventLoop`. Whenever possible, this resource should be reused in Netty applications. If `EventLoops` are not reused, take care to ensure that not too many instances are created, which could result in exhausting system resources.

9.5 Adding multiple ChannelHandlers during a bootstrap

In all of the code examples shown we only added one `ChannelHandler` during the bootstrap process by setting the instance via `handler(...)` or `childHandler(...)`. This may be good enough for simple applications but not for more complex ones. For example, in an application that must support multiple protocols, such as HTTP or WebSockets, a WebSocket fallback such as SockJS or Flash sockets would require a `ChannelHandler` for each. Attempting to handle all these protocols in one channel handler would result in a large and complicated handler. Netty simplifies this by allowing you to provide as many `ChannelHandlers` as are required.

One of Netty's strengths is its ability to "stack" many `ChannelHandlers` in the `ChannelPipeline` and write reusable code. But how do you do this if you can only set one `ChannelHandler` during the bootstrap process?

The answer is simple. Use only one, but use a special one . . .

For exactly this use case Netty provides a special abstract base class called `ChannelInitializer`, which you can extend to initialize a `Channel`. This `ChannelHandler` will be called once the channel is registered on its `EventLoop` and allows you to add `ChannelHandlers` to the `ChannelPipeline` of the channel. This special initializer `ChannelHandler` will remove itself from the `ChannelPipeline` once it's done with initializing the `Channel`.

Sounds complex? Not really. Once you've seen it in action in listing 9.6 it should seem as simple as it really is.

Listing 9.6 Bootstrap and using `ChannelInitializer`

```
ServerBootstrap bootstrap = new ServerBootstrap();                      #1
bootstrap.group(new NioEventLoopGroup(), new NioEventLoopGroup())       #2
    .channel(NioServerSocketChannel.class)                              #3
    .childHandler(new ChannelInitializerImpl());                        #4

ChannelFuture future = bootstrap.bind(new InetSocketAddress(8080));     #5
future.sync();


final class ChannelInitializerImpl extends ChannelInitializer<Channel> {#6
    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();                       #7
        pipeline.addLast(new HttpClientCodec());
```

```
            pipeline.addLast(new HttpObjectAggregator(Integer.MAX_VALUE));

        }
}
```

#1 Create a new ServerBootstrap to create new Socketchannel channels and bind them
#2 Specify the EventLoopGroups which will be used to get EventLoops from and register with the ServerChannel and the accepted channels
#3 Specify the channel class which will be used to instance
#4 Set a handler for I/O and data for the accepted channels
#5 Bind the channel as it's connectionless with the configured bootstrap
#6 ChannelInitializer which is responsible to set up the ChannelPipeline
#7 Add needed handlers to ChannelPipeline. Once the initChannel(…) method completes the ChannelInitializer removes itself from the ChannelPipeline.

As mentioned before, more complex Applications tend to require multiple `ChannelHandlers`. By providing this special `ChannelInitializer`, Netty allows you to insert as many `ChannelHandler` into the `ChannelPipeline` as your Application requires.

9.6 Using Netty ChannelOptions and attributes

It would be annoying if you had to manually configure every channel when it's created. To avoid this, Netty allows you to apply what are called `ChannelOptions` to a bootstrap. These options are automatically applied to all `Channels` created in the bootstrap. The various options available allow you to configure low-level details about connections such as the channel "keep-alive" or "timeout" properties.

Netty Applications are often integrated with an organization's proprietary software. In some cases, Netty components, such as the `Channel`, are passed around and used outside the normal Netty lifecycle. In cases like these, not all the usual properties and data are available. This is just one example, but for cases like these, Netty offers `Channel` attributes.

Attributes allow you to associate data with `Channels` in a safe way, and these attributes work just as well for both client and server `Channels`.

For example, imagine a web server Application where the client has made a request. In order to track which user a `Channel` belongs to, the Application can store the user's ID as an attribute of that `Channel`. Similarly, any object or piece of data can be associated with a `Channel` by using attributes.

Making use of the `ChannelOptions` and attributes is also simple and can make things a lot easier. For example, imagine a case where a Netty WebSocket server was automatically routing messages depending on the user. By using attributes, the application can store the user's ID with the `Channel` to determine where a message should be routed. The Application could be further automated by using a channel option that automatically terminates the connection if no messages have been received for routing within a given time.

Listing 9.7 shows what happens when an invalid configuration is run.

Listing 9.7 `IllegalStateException` thrown due to invalid configuration

```
final AttributeKey<Integer> id = new AttributeKey<Integer>("ID");       #1

Bootstrap bootstrap = new Bootstrap();                                  #2
bootstrap.group(new NioEventLoopGroup())                                #3
        .channel(NioSocketChannel.class)                                #4
        .handler(new SimpleChannelInboundHandler<ByteBuf>()             #5
            @Override
            public void channelRegistered(ChannelHandlerContext ctx)
                    throws Exception {
                Integer idValue = ctx.channel().attr(id).get();         #6
                // do something  with the idValue
            }

            @Override
            protected void channelRead0(
                    ChannelHandlerContext channelHandlerContext,
                    ByteBuf byteBuf) throws Exception {
                System.out.println("Reveived data");
                byteBuf.clear();
            }
        });
bootstrap.option(ChannelOption.SO_KEEPALIVE,true)
        .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 5000);            #7
bootstrap.attr(id, 123456);                                             #8
ChannelFuture future = bootstrap.connect(
        new InetSocketAddress("www.manning.com", 80));                  #9
future.syncUninterruptibly();
```

#1 Create a new AttributeKey under which we'll store the attribute value
#2 Create a new bootstrap to create new client channels and connect them
#3 Specify the EventLoopGroup to get EventLoops from and register with the channels
#4 Specify the channel class that will be used to instance
#5 Set a handler which will handle I/O and data for the channel
#6 Retrieve the attribute with the AttributeKey and its value
#7 Set the ChannelOptions that will be set on the created channels on connect or bind
#8 Assign the attribute
#9 Connect to the remote host with the configured bootstrap

In the previous listings we used a `SocketChannel` in the `Bootstrap`, which is TCP-based. As stated before, a `Bootstrap` is also used for connectionless protocols such as UDP. For this Netty provides various `DatagramChannel` implementations. The only difference here is that you won't call `connect(...)` but only `bind(...)`, as shown in listing 9.8.

Listing 9.8 Using Bootstrap with `DatagramChannel`

```
Bootstrap bootstrap = new Bootstrap();
bootstrap.group(new OioEventLoopGroup()).channel(OioDatagramChannel.class)
       .handler(new SimpleChannelInboundHandler<DatagramPacket>(){

               @Override
           public void channelRead0(ChannelHandlerContext ctx,
                   DatagramPacket msg) throws Exception {
               // Do something with the packet
           }
       });
```

```
ChannelFuture future = bootstrap.bind(new InetSocketAddress(0));
future.addListener(new ChannelFutureListener() {
        @Override
        public void operationComplete(ChannelFuture channelFuture)
                throws Exception {
            if (channelFuture.isSuccess()) {
                System.out.println("Channel bound");
            } else {
                System.err.println("Bound attempt failed");
                channelFuture.cause().printStackTrace();
            }
        }
   });
```

#1 Create a new bootstrap to create new datagram channels and bind them
#2 Specify the EventLoopGroup to get EventLoops from and register with the channels
#3 Specify the channel class that will be used to instance
#4 Set a handler that will handle I/O and data for the channel
#5 Bind the channel as it's connectionless with the configured bootstrap

Netty comes with sane default configuration settings. In many cases you won't need to change these, but there are situations that demand absolute fine-grained control over how your application works and handles data. In these cases, Netty gives you the ability to provide these detailed configurations without too much effort.

9.7 Shutdown a previous bootstrapped Client/Server

Once you have bootstrap your Client or Server and everything is running you will also need to gracefully shutdown those once i.e. the life-time of your application ends. You could just not care about shutdown and let everything be handled by the JVM on exit but this would not be a graceful shutdown, which takes care of release resources in a clean fashion. There is not much magic needed for your netty based application to shutdown, but still there are some things you should be aware of.

Basically the only thing you really need to worry about when shutting down is to remember to shutdown the used `EventLoopGroup`. This will take care of handle pending events / tasks and release all used threads once the handling is complete. This way you will have any threads leaked.

Doing so is as easy as calling `EventLoopGroup.shutdownGracefully()`.This call will return a `Future` which is notified once the shutdown completed. What is important here is that `shutdownGracefully()` is again an asynchronous operation and so you will need to either block until it is complete or have a Listener registered on the returned `Future` which will be notified once the shutdown was completed.

Listing 9.9 Shutdown and release resources

```
EventLoopGroup group = new NioEventLoopGroup()                          #1

Bootstrap bootstrap = new Bootstrap();                                  #2
bootstrap.group(group)
         .channel(NioSocketChannel.class);
```

```
...
...
Future<?> future = group.shutdownGracefully();                    #5
// block until the group was shutdown
future.sync();                                                    #6
```

#1 Create the EventLoopGroup that is used to handle the IO
#2 Create a new Bootstrap and configure it
#3 Finally shutdown the EventLoopGroup gracefully and so release the resources. This will also close all the Channels which are currently in use.

Alternative you can also call `Channel.close()` on all active channels by yourself before call `EventLoopGroup.shutdownGracefully()`.

9.8 Summary

In this chapter you learned how to bootstrap your Netty-based server and client implementation. You learned how you can specify configuration options that affect the and how you can use attributes to attach information to a `Channel` and use it later.

You also learned how to bootstrap connectionless protocol-based applications and how they are different from connection-based ones.

In the next chapter we will look into how you can test your `ChannelHandler` implementations and so provide a way to ensure correctness of them.

10

Unit-test your code

In this Chapter

- `Unit testing`
- `EmbeddedChannel`

As you learned in the previous chapters most of the time you implement one or more `ChannelHandler` for fulfill the various steps needed to actual handle received / send messages. But how to test that it works as expected and not break when refactor the code?

Fortunately testing your `ChannelHandler` implementations is quite easy as Netty offers you two extra classes, which can be used to do this.

After reading this Chapter you will be able to master the last missing piece of make your Netty application robust. Which is test your code…

As the shown tests use JUnit 4, a basic understanding of its usage is required. If you don't have it yet, take your time to read-up on using it. It's really simple but very powerful. You can find all needed information on the JUnit website[27]. Beside this you also should have read the `ChannelHandler` and Codec chapters, as this Chapter will focus on testing your own implementation of a `ChannelHandler` or Codec.

10.1 General

As you learned before Netty provides an easy way to "stack" different `ChannelHandler` implementations together and so build up its `ChannelPipeline`. All of the `ChannelHandler` will then be evaluated while processing events. This design decision allows you to separate the logic into small reusable implementations, which only handle one task each. This not only makes the code cleaner but also allow to easier testing, which will be shown in this Chapter.

Testing of your `ChannelHandlers` is possible by using the "embedded" transport, which allows to easy pass events trough the pipeline and so test your implementations. For this the "embedded" transport offers a special `Channel` implementation, called `EmbeddedChannel`.

But how does it work? It's quite simple, the `EmbeddedChannel` allows you to write either inbound or outbound data into it and then check if something hit the end of the `ChannelPipeline`. This allows you to check if messages where encoded/decoded or triggered any action in your `ChannelHandler`.

[27] http://junit.org/

Inbound vs. Outbound

What is the difference between write inbound vs. outbound? Inbound data is processed by `ChannelInboundHandler` implementations and represent data that is read from the remote peer. Outbound data is processed by `ChannelOutboundHandler` implementations and represent data that will be written to the remote peer.

Thus depending on the `ChannelHandler` you want to test you would choose `writeInbound(...)` or `writeOutbound(…)` (or even both).

That said let me note that `ChannelInboundHandler` and `ChannelOutboundHandler` interfaces were merged into `ChannelHandler` in Netty 5. So while the semantics are the same the interfaces / abstract classes changed.

Those operations are provided by special methods and shown in Table 10.1.

Table 10.1 Special EmbeddedChannel methods

Name	Responsibility
`writeInbound(…)`	Write a message to the inbound of the `Channel`. This means it will get passed through the `ChannelPipeline` in the inbound direction. Returns `true` if something can now be read from the `EmbeddedChannel` via the `readInbound()` method.
`readInbound(…)`	Read a message from the `EmbeddedChannel`. Everything that will be returned here processed the whole `ChannelPipeline`. This method returns `null` if nothing is ready to read.
`writeOutbound(…)`	Write a message to the outbound of the `Channel`. This means it will get passed through the `ChannelPipeline` in the outbound direction. Returns `true` if something can now be read from the `EmbeddedChannel` via the `readOutbound()` method.
`readOutbound(…)`	Read a message from the `EmbeddedChannel`. Everything that will be returned here processed the whole `ChannelPipeline`. This method returns `null` if nothing is ready to read.
`finish()`	Mark the `EmbeddedChannel` as complete and return `true` if something can be read from either the inbound or outbound. This will also call close on the `Channel`.

To make it even more clear how the flow is let us have a look at Figure 10.1, which shows how messages / bytes will get passed through the `ChannelPipeline` by those explained methods.

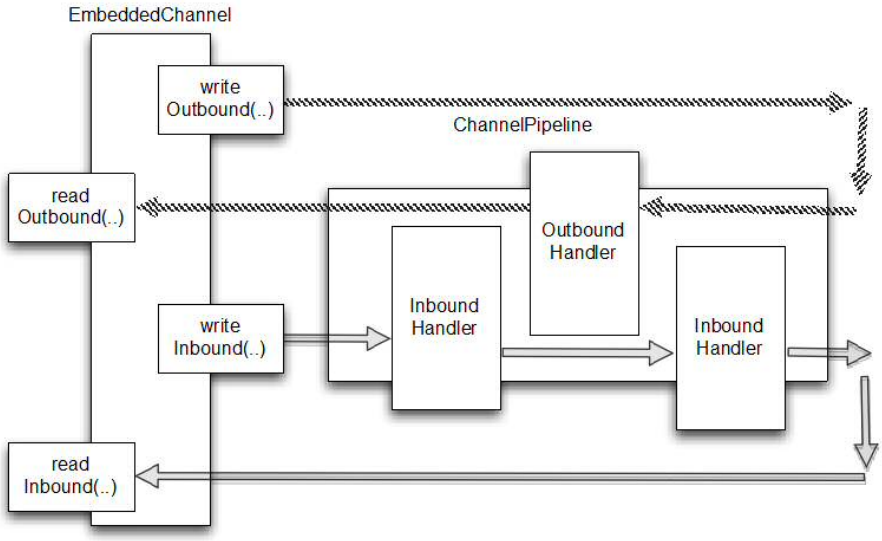


Figure 10.1 Message / byte flow

As shown in Figure 10.1 you can use the `writeOutbound(…)` method to write a message to the `Channel` and so let it pass trough the `ChannelPipeline` in the outbound direction. Later then you can read the processed message with `readOutbound(…)` and so see if the result is what you expect. The same is true for inbound, for which you can use `writeInbound(…)` and `readInbound(…)`. So the semantic is the same between handling outbound and inbound processing, it always traverses the whole `ChannelPipeline` and then is stored in the `EmbeddedChannel` if it hits the end of the `ChannelPipeline`.

With this general information it's time to have a detailed look at both of them and see how you can use each of them to test your logic.

10.2 Testing ChannelHandler

For testing a `ChannelHandler` your best bet is to use `EmbeddedChannel`.

To illustrate how you can use `EmbeddedChannel` for testing a `ChannelHandler` let us make use of it in this chapter and explain it with an example.

10.2.1 Testing inbound handling of messages

For this let us write a simple `ByteToMessageDecoder` implementation, which will produce frames of a fixed size once enough data can be read. If not enough data is ready to read it will wait for the next chunk of data and check again if a frame can be produced. And so on….

Figure 10.2 shows how received bytes will be re-assembled if it operates on frames with a fixed size of 3.

Figure 10.2 Decoding via FixedLengthFrameDecoder

As you can see in Figure 10.3 it may more then once "event" to get enough bytes to produce a frame and pass it to the next `ChannelHandler` in the `ChannelPipeline`.

After the logic should be clear let us have a look at the implementation. It's shown in in Listing 10.1.

Listing 10.1 FixedLengthFrameDecoder implementation

```
public class FixedLengthFrameDecoder extends ByteToMessageDecoder {       #1

    private final int frameLength;

    public FixedLengthFrameDecoder(int frameLength) {                     #2
        if (frameLength <= 0) {
            throw new IllegalArgumentException(
                    "frameLength must be a positive integer: " + frameLength);
        }
        this.frameLength = frameLength;
    }

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in,
            List<Object> out) throws Exception {
        while (in.readableBytes() >= frameLength) {                       #3
            ByteBuf buf = in.readBytes(frameLength);                      #4
            out.add(buf);                                                 #5
        }
    }
}
```

#1 Extend ByteToMessageDecoder and so handle inbound bytes and decode them to messages
#2 Specify the length of frames that should be produced
#3 Check if enough bytes are ready to read for process the next frame
#4 Read a new frame out of the ByteBuf
#5 Add the frame to the List of decoded messages.

Once the implementation is done it's always a good idea to write a unit test to make sure it works like expected. Even if you are sure you should write a test just to guard yourself from problems later as you may refactor your code at some point. This way you will be able to spot problems before it's deployed in production.

Now let us have a look how this is accomplished by using the `EmbeddedChannel`.

Listing 10.2 shows the test case using `EmbeddedChannel`.

Listing 10.2 Test the FixedLengthFrameDecoder

```
public class FixedLengthFrameDecoderTest {

    @Test                                                                #1
    public void testFramesDecoded() {
        ByteBuf buf = Unpooled.buffer();                                 #2
        for (int i = 0; i < 9; i++) {
            buf.writeByte(i);
        }
        ByteBuf input = buf.duplicate();

        EmbeddedChannel channel = new EmbeddedChannel(
                new FixedLengthFrameDecoder(3));                         #3
        // write bytes                                                   #4
        Assert.assertTrue(channel.writeInbound(input));

        Assert.assertTrue(channel.finish());                             #5

        // read messages                                                 #6
        Assert.assertEquals(buf.readBytes(3), channel.readInbound());
        Assert.assertEquals(buf.readBytes(3), channel.readInbound());
        Assert.assertEquals(buf.readBytes(3), channel.readInbound());
        Assert.assertNull(channel.readInbound());
    }


    @Test
    public void testFramesDecoded2() {
        ByteBuf buf = Unpooled.buffer();
        for (int i = 0; i < 9; i++) {
            buf.writeByte(i);
        }
        ByteBuf input = buf.duplicate();

        EmbeddedChannel channel = new EmbeddedChannel(new
      FixedLengthFrameDecoder(3));
        Assert.assertFalse(channel.writeInbound(input.readBytes(2)));
        Assert.assertTrue(channel.writeInbound(input.readBytes(7)));

        Assert.assertTrue(channel.finish());
        Assert.assertEquals(buf.readBytes(3), channel.readInbound());
        Assert.assertEquals(buf.readBytes(3), channel.readInbound());
        Assert.assertEquals(buf.readBytes(3), channel.readInbound());
        Assert.assertNull(channel.readInbound());
    }
}
```

#1 Annotate with @Test to mark it as a test method
#2 Create a new ByteBuf and fill it with bytes
#3 Create a new EmbeddedByteChannel and feed in the FixedLengthFrameDecoder to test it
#4 Write bytes to it and check if they produced a new frame (message)
#5 Mark the channel finished
#6 Read the produced messages and test if its what was expected

To better understand what is done here let us have a deep look in the concept. The `testFramesDecoded()` method wants to test that a `ByteBuf`, which contains 9 readable bytes is decoded in 3 `ByteBuf`, which contains 3 readable bytes each. As you may notice it write the

`ByteBuf` with 9 readable bytes in one call of `writeInbound(…)` here. After this the `finish()` method is executed to mark the `EmbeddedByteChannel` as complete. Finally it calls `readInbound()` to read the produced frames out of the `EmbeddedByteChannel` until nothing can be read anymore.

The `testFramesDecoded2()` act the same way but with one difference. Here the inbound `ByteBufs` are written in two steps. When call `writeInbound(input.readBytes(2))` false is returned as the `FixedLengthFrameDecoder` will only produce output when at least 3 bytes are readable (as you may remember). The rest of the test works quite the same as `testFramesDecoded()`.

10.2.2 Testing outbound handling of messages

After you know all you need to test inbound handling of messages it's time to review how you would handle messages for outbound processing and test them the right way. The concept is quite similar as when handling inbound data, but let us review this while working through an example again.

For show how to test outbound byte processing let us implement the `AbsIntegerEncoder`. What it does is the following:

- Once a `flush(…)` is received it will read all ints from the `ByteBuf` and call `Math.abs(..)` on them
- Once done it write the bytes to the next `ByteBuf` in the `ChannelHandlerPipeline`.

Figure 10.3 shows the logic.

Figure 10.3 Encoding via AbsIntegerEncoder

The logic in Figure 10.4 also is the building ground for the actual implementation, which we will use to write our tests against.

Listing 10.3 shows it in detail.

Listing 10.3 AbsIntegerEncoder

```
public class AbsIntegerEncoder extends MessageToMessageEncode<ByteBuf> { #1
    @Override
    protected void encode(ChannelHandlerContext channelHandlerContext, ByteBuf in,
      List<Object> out) throws Exception {
        while (in.readableBytes() >= 4) {                                  #2
            int value = Math.abs(in.readInt());                            #3
            out.add(value);                                                #4
        }
    }
```

```
}
```

#1 Extend MessageToMessageEncode to encode a message to another message
#2 Check if there is enough bytes to encode
#3 Read the next int out of the input ByteBuf and calculate the absolute int out of it
#4 Write the int to the List of encoded messages

The implementation shown in Listing 10.3 is an exact implementation of the descripted behavior. So it works the same way as shown in Figure 10.4.

As before we make use of `EmbeddedChannel` again to test the `AbsIntegerEncoder`.

Listing 10.4 Test the AbsIntegerEncoder

```
public class AbsIntegerEncoderTest {

    @Test                                                          #1
    public void testEncoded() {
        ByteBuf buf = Unpooled.buffer();                           #2
        for (int i = 1; i < 10; i++) {
            buf.writeInt(i * -1);
        }

        EmbeddedChannel channel = new EmbeddedChannel(
                new AbsIntegerEncoder());                          #3
        Assert.assertTrue(channel.writeOutbound(buf));             #4

        Assert.assertTrue(channel.finish());                       #5

        // read bytes                                              #6
        ByteBuf output = (ByteBuf) channel.readOutbound();
        for (int i = 1; i < 10; i++) {
            Assert.assertEquals(i, output.readInt());
        }
        Assert.assertFalse(output.isReadable());
        Assert.assertNull(channel.readOutbound());
    }
}
```

#1 Annotate with @Test to mark it as a test method
#2 Create a new ByteBuf and fill it with bytes
#3 Create a new EmbeddedChannel and feed in the AbsIntegerEncoder to test it
#4 Write bytes to it and check if they produced bytes
#5 Mark the channel finished
#6 Read the produced messages and test if its what was expected

What the test code does is the following:

- Create a new `ByteBuf` which holds 10 ints
- Create a new `EmbeddedChannel`
- Write the `ByteBuf` to the outbound of the `EmbeddedChannel`. Remember `MessageToMessageEncoder` is a `ChannelOutboundHandler` which will manipulate data that should be written to the remote per, thus we need to use `writeOutbound(…)`.
- Mark the channel finish
- Read all ints out of the outbound output of the `EmbeddedChannel` and check if it only

contain absolute ints

10.3 Testing exception handling

Sometimes transform inbound or outbound data is not enough; often you may need to also throw for example an Exception in some situations. This may be because you guard from malformed input or from handling to big resources or some other cause.

Again let us write an implementation, which throws a `TooLongFrameException` if the input bytes are more then a given limit. Such a feature is often used to guard against resource exhaustion.

Figure 10.4 shows how it will work if the frame size is limited to 3 bytes max.

Figure 10.4 Decoding via FrameChunkDecoder

Figure 10.4 shows the logic. As soon as the input bytes exceed a limit the bytes are discarded and a `TooLongFrameException` is thrown. The other `ChannelHandler` implementations in the `ChannelPipeline` can then handle the `TooLongFrameException` or just ignore it. The handling of the exception would be done in the `ChannelHandler.exceptionCaught(…)` method, for which the `ChannelHandler` implementation would provide some specific implementation.

The implementation is shown in Figure 10.5.

Listing 10.5 FrameChunkDecoder

```
public class FrameChunkDecoder extends ByteToMessageDecoder {            #1

    private final int maxFrameSize;

    public FrameChunkDecoder(int maxFrameSize) {
        this.maxFrameSize = maxFrameSize;
    }

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in,
            List<Object> out) throws Exception {
        int readableBytes = in.readableBytes();                           #2
        if (readableBytes > maxFrameSize)  {
            // discard the bytes                                          #3
            in.clear();
            throw new TooLongFrameException();
        }
        ByteBuf buf = in.readBytes(readableBytes);                        #4
        out.add(buf);                                                     #5
    }
}
```

#1 Extend ByteToMessageDecoder and so handle inbound bytes and decode them to messages
#2 Specify the max size of the frames that should be produced
#3 Discard the frame if its to big and throw a new TooLongFrameException which is an Exception provided by Netty and often used for this purpose.
#4 Read a new frame out of the ByteBuf
#5 Add the frame the List of decoded messages.

Again the best bet to test the code is to use `EmbeddedChannel`. The testing code is shown in Listing 10.6.

Listing 10.6 Testing FixedLengthFrameDecoder

```
public class FrameChunkDecoderTest {

    @Test                                                              #1
    public void testFramesDecoded() {
        ByteBuf buf = Unpooled.buffer();                               #2
        for (int i = 0; i < 9; i++) {
            buf.writeByte(i);
        }
        ByteBuf input = buf.duplicate();

        EmbeddedChannel channel = new EmbeddedChannel(
                new FrameChunkDecoder(3));                             #3

        Assert.assertTrue(channel.writeInbound(input.readBytes(2)));   #4
        try {
            channel.writeInbound(input.readBytes(4));                  #5
            Assert.fail();
        } catch (TooLongFrameException e) {
            // expected
        }
        Assert.assertTrue(channel.writeInbound(input.readBytes(3)));


        Assert.assertTrue(channel.finish());                           #6

        // Read frames                                                 #7
        Assert.assertEquals(buf.readBytes(2), channel.readInbound());
        Assert.assertEquals(buf.skipBytes(4).readBytes(3), channel.readInbound());
    }
}
```

#1 Annotate with @Test to mark it as a test method
#2 Create a new ByteBuf and fill it with bytes
#3 Create a new EmbeddedByteChannel and feed in the FixedLengthFrameDecoder to test it
#4 Write bytes to it and check if they produced a new frame (message)
#5 Write a frame which is bigger then the max frame size and check if this cause an TooLongFrameException
#6 Mark the channel finished
#7 Read the produced messages and test if its what was expected

At first glance this looks quite similar to the test-code we had written in Listing 10.2. But one thing is "special" about it; the handling of the `TooLongFrameException`. After looking at Listing 10.6 you should notice the try / catch block which is used here. This is one of the "special" things of using the `EmbeddedChannel`. If one of the "`write*`" methods produce an

Exception it will be thrown directly wrapped in a `RuntimeException`. This way it's easy to test if an `Exception` was thrown directly.

Even if we used the `EmbeddedChannel` with a `ByteToMessageDecoder`

It should be noted that the same could be done with every `ChannelHandler` implementation that throws an `Exception`.

10.4 Summary

In this chapter you learned how you are be able to test your custom `ChannelHandler` and so make sure it works like you expected. Using the shown techniques you are now be able to make use of JUnit and so ultimately test your code as your are used to.

Using the technics shown in the chapter you will be able to guarantee a high quality of your code and also guard it from misbehavior.

In the next chapters we will focus on writing "real" applications on top of Netty and so show you how you can make real use of it. Even if the applications don't contain any test-code remember it is quite important to do so if you will write your next-gen application.

11

WebSockets

This chapter covers

- WebSockets
- ChannelHandler, Decoder and Encoder
- Bootstrapping your Application

The term "real-time-web" is everywhere these days. Most users expect to get information in real-time while visiting their website. But providing information in real-time is not strength of HTTP itself. This is mainly true because HTTP follows a strict request-response pattern which means the Server can only send something back to the Client (in many cases the Browser) if the Client itself first request it. This is a pretty bad fit if you think about real time updates. Over the years a lot of "workarounds" were invented to allow real time updates while still using HTTP, but none of them was optimal.

To address this problem (and others) the WebSockets Protocol was invented, as a possibility to provide a way to send data from the Client to the Server and from the Server to the Client without the need of the request-response pattern. Today almost all Browser support WebSockets, as its client side API is part of HTML 5 itself. This empowers you with a lot of possibilities how you can make use of WebSockets and so provide real time updates / data to the connected Clients.

Netty comes with support for WebSockets[28], which includes support for all of the well-used versions out there. This makes it a no-brainer to use it in your next application. Because of this you will not need to worry about the protocol internals at all. Just use it in an easy way.

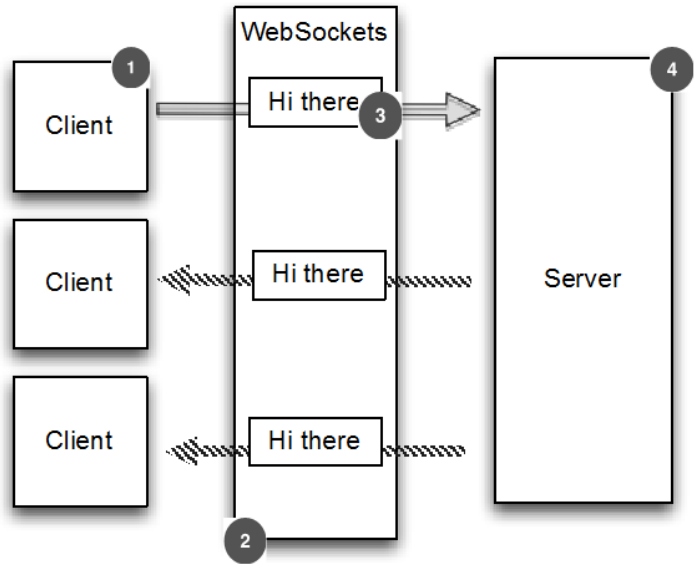
In this chapter you will develop an example Application, which makes use of WebSockets and so help you to understand how Netty supports it. You will learn how you can integrate the shown parts in your Application and so built your next Application with WebSocket support provided.

11.1 The Challenge

To show the "real-time" support of WebSockets, the application will use WebSockets to implement an IRC like chat Application that can be used from within the browser. Kind of what you may know from Facebook where you can send text-messages to another person. But here we will go even further. In this application different users will be able to talk to each other at the same time. So really like IRC.

Figure 11.1 shows the Application logic.

[28] http://tools.ietf.org/html/rfc6455



#1 Client / User which connects to the Server and so is part of the chat
#2 Chat messages are exchanged via WebSockets
#3 Messages are sent form the Clients to the Server and from the Server to the Clients
#4 The Server which handles all the Clients / User

Figure 11.1 Application logic

As shown in Figure 11.1 the logic is quite easy:

- One client sends a message
- The message is broadcasted to all other connected clients

It works just like how you would expect a chat-room to work; everyone can talk to each other. In this example we will only write the server part as the client will be a browser that accesses a web-page on which the chat-room will be displayed.

With this general idea in mind it's time to implement it now, which is quite straight forward, as you will see over the next pages.

11.2 Implementation

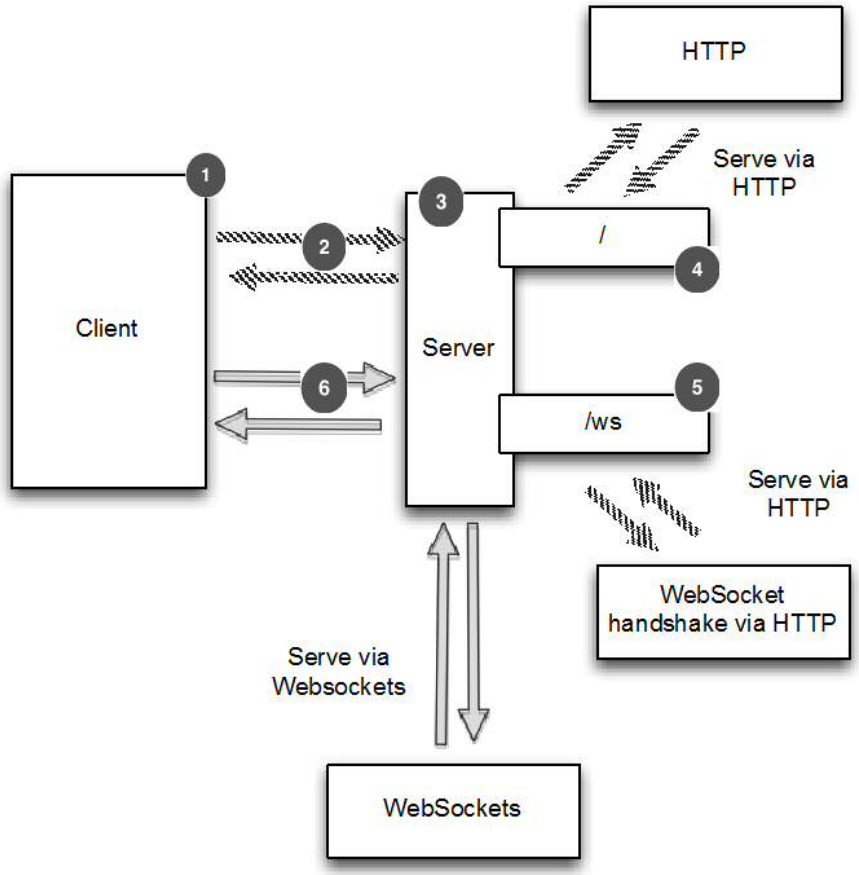
WebSockets uses the HTTP upgrade[29] mechanism to switch from a "normal" HTTP(s) connection to the WebSockets protocol. Because of this an application, which uses WebSockets always starts with HTTP(s) and then perform the upgrade. When exactly the

[29] https://developer.mozilla.org/en-US/docs/HTTP/Protocol_upgrade_mechanism

upgrade happens is specific to the application itself. Some directly perform the upgrade as first action some does it only after a specific URL was requested.

In the case of our Application we will only upgrade to WebSockets if the URL ends with /ws . Otherwise the server will transmit a website to the client via normal HTTP(s). Once upgraded the connection will transmit all data using the WebSockets protocol.

Figure 11.2 shows the Server logic.



#1 Client / User which connects to the Server and so is part of the chat
#2 Request Website or WebSocket handshake via HTTP
#3 The Server which handles all the Clients / Users
#4 Handles responses to the uri "/", which in this case is by send page the index.html page
#5 Handles the WebSockets upgrade if the uri "/ws" is accessed
#6 Send chat messages via WebSockets after the upgrade was complete

Figure 11.2 Server logic

As always the logic will be implemented by different `ChannelHandlers` as this allows for easier reuse and testing of the implementations.

Now with the logic in mind, the next section will give you more details about the `ChannelHandlers` that are needed and the various techniques they use for this purpose.

11.2.1 Handle HTTP requests

As mentioned before the Server will act as a kind of hybrid to allow handling HTTP(s) and WebSocket at the same time. This is needed as the application also needs to serve the html page, which is used to interact with the Server and display messages that are sent via the chatroom itself.

Because of this we need write a `ChannelInboundHandler` that will handle `FullHttpRequests` that are not sent to the /ws URI.

First let us have a look at Listing 11.1, which shows the implementation and then go in the details to explain all the used "features/techniques".

Listing 11.1 Handles HTTP Requests

```
public class HttpRequestHandler
        extends SimpleChannelInboundHandler<FullHttpRequest> {          #1
    private final String wsUri;
    private static final File INDEX;

    static {
        URL location = HttpRequestHandler.class.getProtectionDomain()
                .getCodeSource().getLocation();
        try {
            String path = location.toURI() + "index.html";
            path = !path.contains("file:") ? path : path.substring(5);
            INDEX = new File(path);
        } catch (URISyntaxException e) {
            throw new IllegalStateException("Unable to locate index.html", e);
        }
    }

    public HttpRequestHandler(String wsUri) {
        this.wsUri = wsUri;
    }

    @Override
    public void channelRead0(ChannelHandlerContext ctx, FullHttpRequest request)
      throws Exception {
        if (wsUri.equalsIgnoreCase(request.getUri())) {
            ctx.fireChannelRead(request.retain());                      #2
        } else {
            if (HttpHeaders.is100ContinueExpected(request)) {
                send100Continue(ctx);                                   #3
            }

            RandomAccessFile file =
                    new RandomAccessFile(INDEX, "r");                   #4

            HttpResponse response = new DefaultHttpResponse(
                    request.getProtocolVersion(), HttpResponseStatus.OK);
```

```
            response.headers().set(
                    HttpHeaders.Names.CONTENT_TYPE,
                    "text/plain; charset=UTF-8");

            boolean keepAlive = HttpHeaders.isKeepAlive(request);

            if (keepAlive) {                                                #5
                response.headers().set(
                        HttpHeaders.Names.CONTENT_LENGTH, file.length());
                response.headers().set(
                        HttpHeaders.Names.CONNECTION,
                        HttpHeaders.Values.KEEP_ALIVE);
            }
            ctx.write(response);                                            #6

            if (ctx.pipeline().get(SslHandler.class) == null) {             #7
                ctx.write(new DefaultFileRegion(
                        file.getChannel(), 0, file.length()));
            } else {
                ctx.write(new ChunkedNioFile(file.getChannel());
            }
            ChannelFuture future = ctx.writeAndFlush(
                    LastHttpContent.EMPTY_LAST_CONTENT);                    #8
            if (!keepAlive) {
                future.addListener(ChannelFutureListener.CLOSE);            #9
            }
        }
    }

    private static void send100Continue(ChannelHandlerContext ctx) {
        FullHttpResponse response = new DefaultFullHttpResponse(
                HttpVersion.HTTP_1_1, HttpResponseStatus.CONTINUE);
        ctx.writeAndFlush(response);
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
            throws Exception {
        cause.printStackTrace();
        ctx.close();
    }
}
```

#1 Extend SimpleChannelInboundHandler and handle FullHttpRequest messages
#2 Check if the request is an WebSocket Upgrade request and if so retain it and pass it to the next ChannelInboundHandler in the ChannelPipeline.
#3 Handle 100 Continue requests to conform HTTP 1.1
#4 Open the index.html file which should be written back to the client
#5 Add needed headers depending of if keepalive is used or not
#6 Write the HttpRequest to the client. Be aware that we use a HttpRequest and not a FullHttpRequest as it is only the first part of the request. Also we not use writeAndFlush(..) as this should be done later.
#7 Write the index.html to the client. Depending on if SslHandler is in the ChannelPipeline use DefaultFileRegion or ChunkedNioFile
#8 Write and flush the LastHttpContent to the client which marks the requests as complete.
#9 Depending on if keepalive is used close the Channel after the write completes.

The `HttpRequestHandler` shown in Figure 11.1 does the following:

- Check if the HTTP request was sent to the URI /ws and if so call `retain()` on the

`FullHttpRequest`. This `FullHttpRequest` is then forwarded to the next `ChannelInboundHandler` in the `ChannelPipeline` by calling `ChannelHandlerContext.fireChannelRead(msg)`. The call of `retain()` is needed as after `channelRead0(…)` completes it will call `release()` on the `FullHttpRequest` and so release the resources of it. Remember this is how `SimpleChannelInboundHandler` works.

- Send a 100 Continue response if requested
- Write an `HttpResponse` back to the client after the headers are set.
- Create a new `DefaultFileRegion`, which holds the content of the index.html page and write it back to the client. By doing so, zero-copy is archived which gives the best performance when transfer files over the network. Because this is only possible when no encryption / compression etc. is needed it checks if the `SslHandler` is in the `ChannelPipeline` and if so fallback to use `ChunkedNioFile`.
- Write a `LastHttpContent` to mark the end of the response and so terminate it
- Add a `ChannelFutureListener` to the `ChannelFuture` of the last write if no keep-alive is used and close the connection. The important thing to note here is that it called `writeAndFlush(…)` which means all previous written messages will also be flushed out to the remote peer.

But this is only one part of the application; we still need to handle the WebSocket frames, which will be used to transmit the chat messages. This will be covered in the next section.

11.2.2 Handle WebSocket frames

So we are already able to handle pure HTTP requests. Now it's time for some WebSockets action.

WebSockets supports six different frames in the official RFC. Netty provides a POJO implementation for all of them.

Table 11.1 lists all of the frame types and gives you an overview what these are used for.

Table 11.1 WebSocketFrame types

Method name	Description
`BinaryWebSocketFrame`	`WebSocketFrame` that contains binary data
`TextWebSocketFrame`	`WebSocketFrame` that contains text data
`ContinuationWebSocketFrame`	`WebSocketFrame` that contains text or binary data that belongs to a previous `BinaryWebSocketFrame` or `TextWebSocketFrame`
`CloseWebSocketFrame`	`WebSocketFrame` that represent a CLOSE request and contains close status code and a phrase
`PingWebSocketFrame`	`WebSocketFrame` which request the send of a `PongWebSocketFrame`

`PongWebSocketFrame`	`WebSocketFrame` which is send as response of a `PingWebSocketFrame`

For our application we are only interested in handling a few of them.

Those are:

- `CloseWebSocketFrame`
- `PingWebSocketFrame`
- `PongWebSocketFrame`
- `TextWebSocketFrame`

Fortunately we only need to explicitly handle the `TextWebSocketFrame` . The `WebSocketServerProtocolHandler` will automatically handle all the other `WebSocketFrame` implementations. This makes things a lot easier for our implementation and us.

Listing 11.2 shows the `ChannelInboundHandler` responsible for handling the `TextWebSocketFrames`. Beside this it also tracks all the active WebSockets connections in its `ChannelGroup`.

Listing 11.2 Handles Text frames

```
public class TextWebSocketFrameHandler extends
        SimpleChannelInboundHandler<TextWebSocketFrame> {              #1
    private final ChannelGroup group;

    public TextWebSocketFrameHandler(ChannelGroup group) {
        this.group = group;
    }

    @Override
    public void userEventTriggered(ChannelHandlerContext ctx,
            Object evt) throws Exception {                             #2
        if (evt == WebSocketServerProtocolHandler
                .ServerHandshakeStateEvent.HANDSHAKE_COMPLETE) {
                ctx.pipeline().remove(HttpRequestHandler.class);       #3

            group.writeAndFlush(new TextWebSocketFrame("Client " +
                    ctx.channel() + " joined"));                       #4
             group.add(ctx.channel());                                 #5

        } else {
            super.userEventTriggered(ctx, evt);
        }
    }

    @Override
    public void channelRead0(ChannelHandlerContext ctx,
            TextWebSocketFrame msg) throws Exception {
        group.writeAndFlush(msg.retain());                             #6
    }
}
```

#1 Extend SimpleChannelInboundHandler and handle TextWebSocketFrame messages
#2 Override the userEventTriggered(…) method to handle custom events

#3 If the event is received that indicate that the handshake was successful remove the HttpRequestHandler from the ChannelPipeline as no further HTTP messages will be send.
#4 Write a message to all connected WebSocket clients about a new Channel that is now also connected
#5 Add the now connected WebSocket Channel to the ChannelGroup so it also receive all messages
#6 Retain the received message and write and flush it to all connected WebSocket clients.

The `TextWebSocketFrameHandler` shown in Figure 11.2 has again a very limited set of responsibilities, which are:

- Once the WebSocket handshake complete successfully write a message to the `ChannelGroup`, which holds all active WebSockets `Channels`. This means every `Channel` in the `ChannelGroup` will received it. Also it's adding the Channel to the `ChannelGroup` as it is active now too.
- If a `TextWebSocketFrame` is received, call `retain()` on it and write and flush it to the `ChannelGroup` so every already connected WebSockets `Channel` receives it. Calling `retain()` is needed because otherwise the `TextWebSocketFrame` would be released once the `channelRead0(…)` method returned. This is a problem as `writeAndFlush(…)` may complete later, remember everything is done asynchronously.

That's almost all you need as the rest is provided by Netty itself as stated before. So the only thing left is to provide a `ChannelInitializer` implementation, which will be used to initialize the `ChannelPipeline` for a new `Channel`.

11.2.3 Initialize the ChannelPipeline

With the `ChannelHandler` implemented we need to make sure it is also used. As part of this we initialize the `ChannelPipeline` with all the `ChannelHandlers` that are needed to support WebSockets. As in other chapters this is done by extending the `ChannelInitializer` class and implementing the `initChannel(…)` method.

Listing 11.3 shows our `ChatServerInitializer` that is responsible for this.

Listing 11.3 Init the ChannelPipeline

```
public class ChatServerInitializer extends ChannelInitializer<Channel> { #1
    private final ChannelGroup group;

    public ChatServerInitializer(ChannelGroup group) {
        this.group = group;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {            #2
            ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(new HttpServerCodec());
        pipeline.addLast(new ChunkedWriteHandler());
        pipeline.addLast(new HttpObjectAggregator(64 * 1024));
        pipeline.addLast(new HttpRequestHandler("/ws"));
        pipeline.addLast(new WebSocketServerProtocolHandler("/ws"));
        pipeline.addLast(new TextWebSocketFrameHandler(group));
    }
}
```

#1 Extend ChannelInitializer as this should init the ChannelPipeline
#2 Add all needed ChannelHandler to the ChannelPipeline

The `initChannel(…)` method setup the `ChannelPipeline` of the newly registered `Channel`, this is done by adding the different `ChannelHandler` to the `ChannelPipeline` that are needed to provide the logic needed.

Let us recap what `ChannelHandlers` are added to the `ChannelPipeline` and what are their responsibilities in this case.

Table 11.2 Responsibilities of the ChannelHandlers

ChannelHandler	Responsibility
`HttpServerCodec`	Decode bytes to `HttpRequest`, `HttpContent`, `LastHttpContent` . Encode `HttpRequest`, `HttpContent`, `LastHttpContent` to bytes.
`ChunkedWriteHandler`	Allows writing a file.
`HttpObjectAggregator`	Aggregate decoded `HttpRequest` , `HttpContent` , `LastHttpContent` to `FullHttpRequest`. This way you will always receive only “full” Http requests in the `ChannelHandler` which is placed after it.
`HttpRequestHandler`	Handle `FullHttpRequest` which are not send to /ws URI.
`WebSocketServerProtocolHandler`	Handle the WebSocket upgrade and `PingWebSocketFrames`, `PongWebSocketFrames` and `CloseWebSocketFrames` to be RFC compliant
`TextWebSocketFrameHandler`	Handles `TextWebSocketFrames` and handshake completion events

Something is special about the `WebSocketServerProtcolHandler` so it deserves a bit more of explanation. The `WebSocketServerProtocol` not only handles `PingWebSocketFrames`, `PongWebSocketFrames` and `CloseWebSocketFrames` but also the handshake itself.

This is done by execute the handshake and once successful modify the `ChannelPipeline` to add all the needed `ChannelHandler` and remove these that are not needed anymore.

To make it more clear have a look at Figure 11.3.

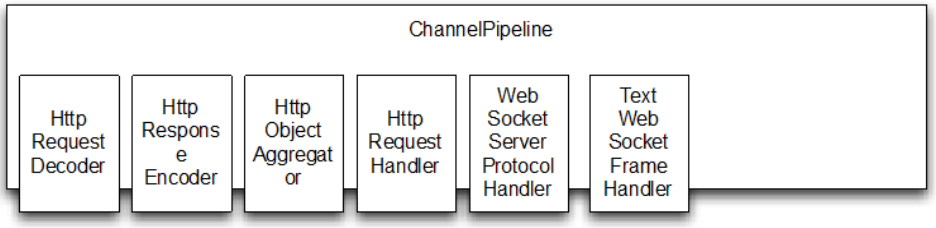


Figure 11.3 ChannelPipeline before WebSocket upgrade took place

Figure 11.3 shows the `ChannelPipeline` like it was initialized by the `initChannel(…)` method of `ChatServerInitializer`. But things change as soon as the handshake was completed. Once this is done the `WebSocketServerProtocolHandler` will take care of replace the `HttpRequestDecoder` with a `WebSocketFrameDecoder` and the `HttpResponseEncoder` with the `WebSocketFrameEncoder`. Beside this it also removes all `ChannelHandlers` that are not needed anymore as part of a WebSocket connection, to gain optimal performance. Those are the `HttpObjectAggregator` and the `HttpRequestHandler`.

Figure 11.4 shows how the `ChannelPipeline` looks like once the handshake completes. In this example we assume Version 13 of the WebSocket protocol is used, and so the `WebSocketFrameDecoder13` and `WebSocketFrameEncoder13` are used. Choosing the right `WebSocketFrameDecoder` and `WebSocketFrameEncoder` is done for you depending on what the Client (in our case the Browser) supports.

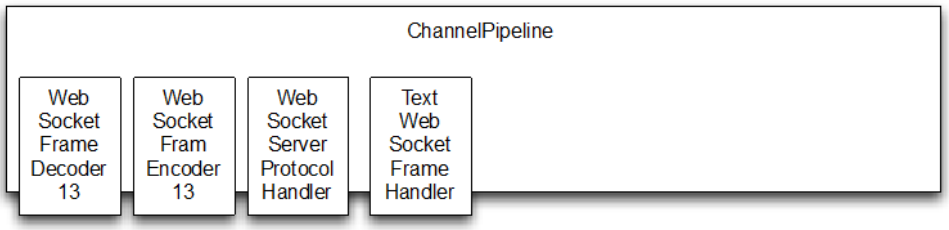


Figure 11.4 ChannelPipeline after WebSocket upgrade completes

The modification of the `ChannelPipeline` is done without you even noticing it as it is executed "under the hood". This is once again only possible because of the very flexible way of updating the `ChannelPipeline` on the fly and separate tasks in different `ChannelHandler` implementations.

11.2.4 Wire things together – a.k.a Boostrapping

As always you need to wire things together to make use of them. You learned before this is done via bootstrapping the server and setting the correct `ChannelInitializer`.

Listing 11.4 shows the `ChatServer` class, which does all of this.

Listing 11.4 Bootstrap the Server

```
public class ChatServer {

    private final ChannelGroup channelGroup =
        new DefaultChannelGroup(ImmediateEventExecutor.INSTANCE);       #1
    private final EventLoopGroup group = new NioEventLoopGroup();
    private Channel channel;

    public ChannelFuture start(InetSocketAddress address) {
        ServerBootstrap bootstrap  = new ServerBootstrap();             #2
        bootstrap.group(group)
                .channel(NioServerSocketChannel.class)
                .childHandler(createInitializer(channelGroup));
        ChannelFuture future = bootstrap.bind(address);
        future.syncUninterruptibly();
        channel = future.channel();
        return future;
    }

    protected ChannelInitializer<Channel> createInitializer(
           ChannelGroup group) {                                        #3
       return new ChatServerInitializer(group);
    }

    public void destroy() {                                             #4
        if (channel != null) {
            channel.close();
        }
        channelGroup.close();
        group.shutdownGracefully();
    }

    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Please give port as argument");
            System.exit(1);
        }
        int port = Integer.parseInt(args[0]);

        final ChatServer endpoint = new ChatServer();
        ChannelFuture future = endpoint.start(
                new InetSocketAddress(port));

        Runtime.getRuntime().addShutdownHook(new Thread() {
            @Override
            public void run() {
                endpoint.destroy();
            }
        });
        future.channel().closeFuture().syncUninterruptibly();
    }
}
```

#1 Create DefaultChannelGroup that will hold all connected WebSocket Channels
#2 Bootstrap the server
#3 Create the to be used ChannelInitializer
#4 Handle destroying of the Server, this includes release of all resources

That's all needed for the Application itself; it's ready now for a test-run.

The easiest way to start up the Application is to use the source-code provided for this book[30]. It has everything ready to use and will use Apache Maven[31] to automatically setup everything and download dependencies.

Starting it up via Apache Maven is as easy as shown in Listing 11.5

Listing 11.5 Compile and start the ChatServer

```
Normans-MacBook-Pro:netty-in-action$ mvn clean package exec:exec -Pchapter11-
      ChatServer
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building netty-in-action 0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
...
...
...
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action ---
[INFO] Building jar: /Users/norman/Documents/workspace-intellij/netty-in-
      action/target/netty-in-action-0.1-SNAPSHOT.jar
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ netty-in-action ---
```

Now you can point your web-browser to **http://localhost:9999** and access the application. Figure 11.5 show the interface included in the code for this chapter.

[30] https://github.com/normanmaurer/netty-in-action
[31] http://maven.apache.org

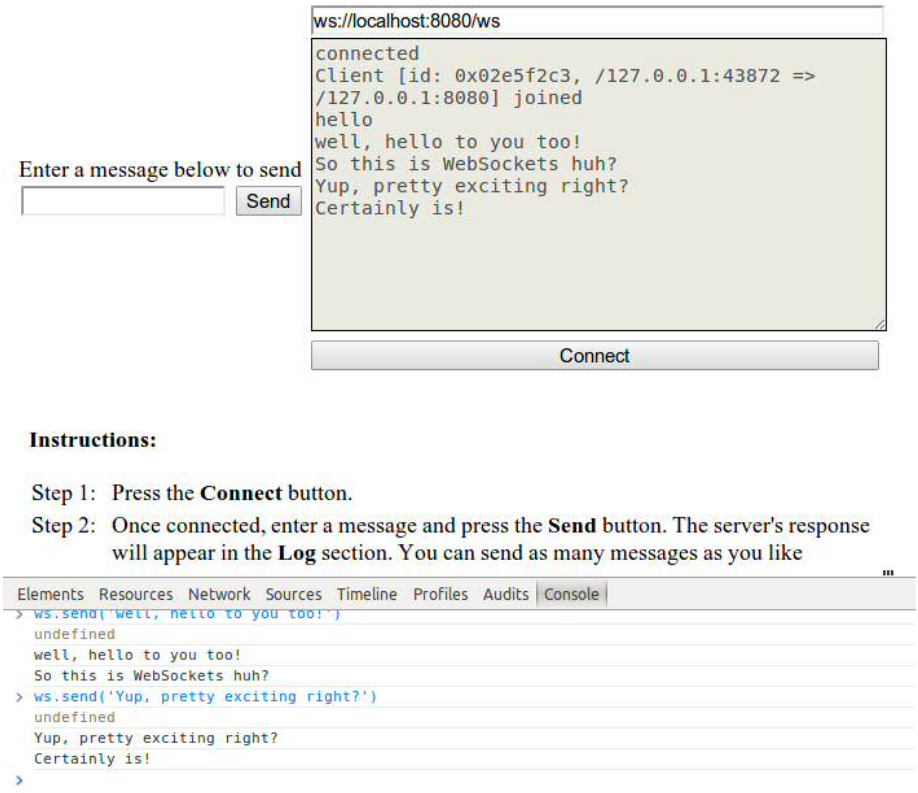


Figure 11.5 WebSocket chat demonstration

In figure 11.5 above, we have two clients connected. The first client is connected using the interface at the top. The second client is connected via the browser's command line at the bottom. You'll notice that there are messages sent from both clients and each message is displayed to both.

This is a very simple example of real-time communication that WebSocket enables in the Browser.

11.2.5 What about Encryption?

Even as the application works quite well you may notice at some point that requirements change. A typical change is the requirement of encryption, as you start to get concerned about transmitted data in plaintext.

Often adding this kind of new feature is not an easy task, and need big changes to the project. But not when using Netty! It's as easy as adding the `SslHandler` to the

`ChannelPipeline`. So more or less a "one-line" change if you ignore the fact that you also need to configure the `SslContext` somehow.

There is not much you need to adjust to make it use of encryption. Thanks again to the design of Netty's `ChannelPipeline`.

Let us have a look what changes are needed.

First of it's needed to add a `SslHandler` to the `ChannelPipeline`. Listing 11.6 extends the previous created `ChatServerInitializer` to make this happen.

Listing 11.6 Add encryption to the ChannelPipeline

```
public class SecureChatServerIntializer extends ChatServerInitializer { #1
    private final SSLContext context;

    public SecureChatServerIntializer(ChannelGroup group, SSLContext context) {
        super(group);
        this.context = context;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        super.initChannel(ch);
        SSLEngine engine = context.createSSLEngine();
        engine.setUseClientMode(false);
        ch.pipeline().addFirst(new SslHandler(engine));                #2
    }
}
```

#1 Extend the ChatServerInitializer as we want to enhance it's functionality
#2 Add the SslHandler to the ChannelPipeline to enable encryption

That's basically all what is needed. Now the only thing left is to adjust the `ChatServer` class to use the `SecureChatServerInitializer` and pass in the `SSLContext` to use.

Listing 11.7 shows the `SecureChatServer`.

Listing 11.7 Add encryption to the ChatServer

```
public class SecureChatServer extends ChatServer {                    #1

    private final SSLContext context;

    public SecureChatServer(SSLContext context) {
        this.context = context;
    }

    @Override
    protected ChannelInitializer<Channel> createInitializer(
            ChannelGroup group) {
        return new SecureChatServerInitializer(group, context);        #2
    }

    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Please give port as argument");
            System.exit(1);
```

```
        }
        int port = Integer.parseInt(args[0]);

        SSLContext context = BogusSslContextFactory.getServerContext();
        final SecureChatServer endpoint = new SecureChatServer(context);
        ChannelFuture future = endpoint.start(new InetSocketAddress(port));

        Runtime.getRuntime().addShutdownHook(new Thread() {
            @Override
            public void run() {
                endpoint.destroy();
            }
        });
        future.channel().closeFuture().syncUninterruptibly();
    }
}
```

#1 Extend ChatServer to enhance the functionality
#2 Return the previous created SecureChatServerInitializer to enable encryption

That's all the magic needed. The Application is now using SSL/TLS[32] to encrypt the whole network communication

As before you can use Apache Maven to startup the application and have it pull in all needed dependencies.

Listing 11.8 shows how to start it.

Listing 11.7 Add encryption to the ChatServer

```
Normans-MacBook-Pro:netty-in-action$ mvn clean package exec:exec -Pchapter11-
      SecureChatServer
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building netty-in-action 0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
...
...
...
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action ---
[INFO] Building jar: /Users/norman/Documents/workspace-intellij/netty-in-
      action/target/netty-in-action-0.1-SNAPSHOT.jar
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ netty-in-action ---
```

After it starts up you can access it via https://localhost:9999. Note that we use https now which means the network communication is encrypted and so safe to use.

[32] http://tools.ietf.org/html/rfc5246

11.3 Summary

In this chapter you learned how you could make use of WebSockets in a Netty based Application to use it for real-time data in the web.

You learned what types of data you can transmit and how you can make use of them. Beside this you should now have an idea about why WebSockets is one of the "new big things" out there.

Also you got a short overview about limitations which you may see while use WebSockets and why it may not be possible to use it in all the cases.

In the next chapter you will learn about another Web 2.0 thing, which may help you to make your service more appealing. The chapter will show you what all the hype of SPDY[33] is about and why you may want to support it in your next Application.

[33] http://www.chromium.org/spdy/spdy-whitepaper

12

SPDY

In this Chapter we're going to look at

- An overview of SPDY in general
- ChannelHandler, Decoder and Encoder
- Bootstrap a Netty based Application
- Test SPDY / HTTPS

Netty comes bundled with support for SPDY and so makes it a piece of cake to have your application make use of it without worrying about all the protocol internals at all.

After this chapter you will know what you need to do to make your application "SPDY-ready" and also know how to support SPDY and HTTP at the same time.

This Chapter contains everything you need to know to get the example application working with SPDY and HTTP at the same time. While it can't harm to have read at least the `ChannelHandler` and Codec Chapter it is not needed as everything is provided here. However, this chapter will not go into all the details that are covered in the previous chapters.

12.1 SPDY some background

SPDY was developed by Google to solve their scaling issues. One of its main tasks is to make the "loading" of content as fast as possible.

For this SPDY does a few things:

- Every header that is transferred via SDPY gets compressed. Compression the body is optional as it can be problematic for Proxy servers.
- Everything is encrypted via TLS
- Multiple transfer per connection possible
- Allows you to set priority for different data enabling critical files to be transferred first

Table 12.1 shows how this compares to HTTP.

Table 12.1 Compare HTTP and SPDY

Browser	HTTP 1.1	SPDY
`Encrypted`	Not by default	Yes
`Header Compression`	No	Yes
`Full-Duplexing`	No	Yes
`Server-Push`	No	Yes
`Priorities`	No	Yes

Some benchmarks have shown that a speed up of 50% percent is possible while using SPDY over HTTP. But the exact numbers of course depends on the use-case.

While SPDY was only supported by Google Chrome in the early years it is now supported in most of the Web-browsers at the time of writing.

SPDY is currently available in 3 different versions based on Drafts:

- Draft 1
- Draft 2
- Draft 3

Netty supports Draft 2 and 3 at the moment. Those are the drafts that are supported by the well-known Browsers.

Table 12.1 shows the browsers that support SPDY on the time of writing.

Table 12.1 Browsers

Browser	Version
`Chrome`	19+
`Chromium`	19+
`Mozilla Firefox`	11+ (enabled by default since 13)
`Opera`	12.10+

12.2 The Challenge

To keep it simple we will write a very simple application, which will show you, how you can integrate SPDY in your next application.

The application does only do one thing. It serves some static content. Based on whether its standard HTTPS or SPDY it will write different content to the client back. The switch to SPDY will get done automatically based on if the client (browser) supports the SPDY version, which is provided by our server.

Figure 12.1 shows the flow.

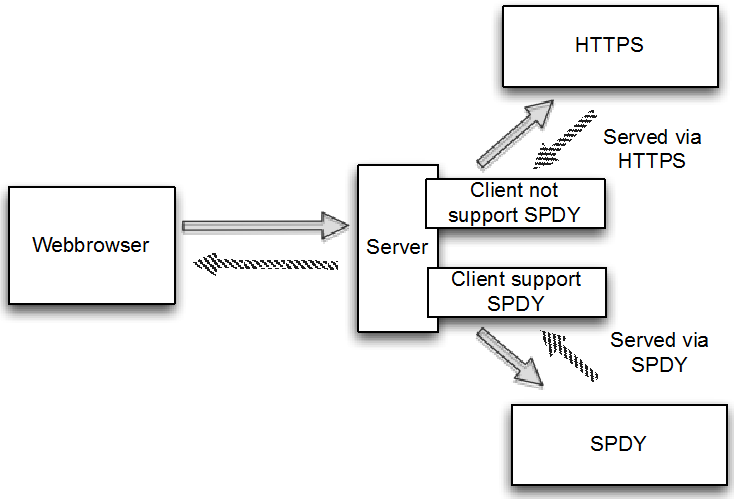


Figure 12.1 Application logic

For this application we will only write the server component, which handles HTTPS and SPDY. For the client side we will use a Web-browser, one that supports SPDY and one that does not, in order to show the difference.

12.3 Implementation

SPDY uses a TLS-Extension, which is called Next Protocol Negotiation (NPN) to choose the protocol. Using Java there are currently two different ways that allows hooking in here.

1. Use ssl_npn
2. Use NPN via the Jetty extension library

In this example we will use the jetty library to make use of SPDY. If you want to use ssl_npn please refer to the documentation of the project.

Jetty NPN Library

The Jetty NPN Library is an external library and so not part of Netty itself. It is responsible for the Next Protocol Negotiation, which is used to detect if the client supports SPDY or not.

12.3.1 Integration with Next Protocol Negotiation

The jetty library provides an interface called `ServerProvider`, which is used to determine the protocol to use. The `ServerProvider` implementation will then allow hooking in into the

protocol selection. Depending on which HTTP versions and which SPDY versions should be supported it may differ in terms of implementation.

Listing 12.1 shows the implementation used by this example.

Listing 12.1 Implementation of ServerProvider

```
public class DefaultServerProvider implements NextProtoNego.ServerProvider {
    private static final List<String> PROTOCOLS =
            Collections.unmodifiableList(
                    Arrays.asList("spdy/2", "spdy/3", "http/1.1"));     #1

    private String protocol;

    @Override
    public void unsupported() {
        protocol = "http/1.1";                                          #2
    }

    @Override
    public List<String> protocols() {
        return PROTOCOLS;                                               #3
    }

    @Override
    public void protocolSelected(String protocol) {
        this.protocol = protocol;                                       #4
    }

    public String getSelectedProtocol() {
        return protocol;                                                #5
    }
}
```

#1 Define all supported protocols by this ServerProvider implementation
#2 Set the protocol to http/1.1 if SPDY detection failed
#3 Return all the supported protocols by this ServerProvider
#4 Set the selected protocol
#5Return the previous selected protocol

For this application we want to support 3 different protocols, and so have make them selectable in the implementation of our `ServerProvider`.

Those are:

- SPDY draft 2
- SPDY draft 3
- HTTP 1.1

If it fails to detect the protocol it will use HTTP 1.1 as default and so allow serving all clients even if SPDY isn't supported.

This is the only integration code that needs to get written for make use of NPN. Next on the list is to write the actual code to make use of it.

12.3.2 Implementation of the various ChannelHandlers

Now it's time to write the Simple`ChannelInboundHandler`, which will handle the HTTP requests for the clients that do not support SPDY.

As mentioned before SPDY is still considered very new and so not all clients support it yet. For exactly this reason you should always provide the ability to fallback to HTTP as otherwise you will give many users a bad experience while using your service.

Listing 12.2 shows the handler, which handles HTTP traffic

Listing 12.2Implementation which handles HTTP

```
@ChannelHandler.Sharable
public class HttpRequestHandler
        extends SimpleChannelInboundHandler<FullHttpRequest> {
    @Override
    public void channelRead0(ChannelHandlerContext ctx,
FullHttpRequest request) throws Exception {                              #1
        if (HttpHeaders.is100ContinueExpected(request)) {
            send100Continue(ctx);                                        #2
        }

        FullHttpResponse response = new DefaultFullHttpResponse(
                request.getProtocolVersion(), HttpResponseStatus.OK);    #3

response.content().writeBytes(getContent()
.getBytes(CharsetUtil.UTF_8));                                           #4
        response.headers().set(HttpHeaders.Names.CONTENT_TYPE,
 "text/plain; charset=UTF-8");                                           #5

        boolean keepAlive = HttpHeaders.isKeepAlive(request);

        if (keepAlive) {                                                 #6
            response.headers().set(HttpHeaders.Names.CONTENT_LENGTH,
 response.content().readableBytes());
            response.headers().set(HttpHeaders.Names.CONNECTION,
 HttpHeaders.Values.KEEP_ALIVE);
        }
        ChannelFuture future = ctx.writeAndFlush(response);              #7

        if (!keepAlive) {
            future.addListener(ChannelFutureListener.CLOSE);             #8
        }
    }

    protected String getContent() {                                      #9
        return "This content is transmitted via HTTP\r\n";
    }

    private static void send100Continue(ChannelHandlerContext ctx) {    #10
        FullHttpResponse response = new DefaultFullHttpResponse(
HttpVersion.HTTP_1_1, HttpResponseStatus.CONTINUE);
        ctx.writeAndFlush(response);
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx,
Throwable cause)throws Exception {                                      #11
```

```
        cause.printStackTrace();
        ctx.close();
    }
}
```

#1 Override the messageReceived(…) method which will be called for each FullHttpRequest that is received
#2 Check if a Continue response is expected and if so write one
#3 Create a new FullHttpResponse which ill be used as "answer" to the request itself
#4 Generate the content of the response and write it to the paylod of it
#5 Set the header so the client knows how to interpret the payload of the response
#6 Check if the request was set with the keepalive enabled, if so set the needed headers to fulfill the http rfc
#7 Write the response back to the client and get a reference to the future which will get notified once the write completes
#8 If the response is not using keepalive take care of close the connection after the write completes
#9 Return the content which will be used as payload of the response
#10 Helper method to generate the 100 continue response and write it back to the client
#11 Close the channel if an exception was thrown during processing

Let us recap what the implementation in Listing 12.2 actual does:

1. The **`channelRead0(….)`** method is called once a **`FullHttpRequest`** was received.
2. A **`FullHttpResponse`** with status code 100 (continue) is written back to the client if expected
3. A new **`FullHttpResponse`** is created with status code 200 (ok) and its payload is filled with some content
4. Depending on if keep-alive is set in the **`FullHttpRequest`**, headers are set to the **`FullHttpResponse`** to be RFC compliant.
5. The **`FullHttpResponse`** is written back to the client and the channel (connection) is closed if keep-alive is not set

This is how you typically would handle HTTP with Netty. You may decide to handle URI's differently and may also respond with different status codes, depending on if a resource is present or not but the concept is the same.

But wait this chapter is about SPDY, isn't it? So while it's nice to be able to handle HTTP as fallback it's still not what we want as the preferred protocol. Thanks to the provided SPDY handlers by Netty this is a very easy.

Netty will just allow you to re-use the `FullHttpRequest` / `FullHttpResponse` messages and transparently receive/send them via SPDY. Thus we basically can reuse our previous written handler.

One thing to change is the content to write back to the client. This is mainly done to show the difference; in real world you may just write back the same content.

Listing 12.3 shows the implementation, which just extends the previous written `HttipRequestHandler`.

Listing 12.3 Implementation which handles SPDY

```
public class SpdyRequestHandler extends HttpRequestHandler {            #1
    @Override
    protected String getContent() {
        return "This content is transmitted via SPDY\r\n";               #2
    }
}
```

#1 Extends HttpRequestHandler and so share the same logic
#2 Generate the content which is written to the payload. This overrides the implementation of this method in HttpRequestHandler.

The inner working of the `SpdyRequestHandler` is the same as `HttpReuestHandler`, as it extends it and so share the same logic. Only with one difference; The Content, which is written to the payload, contains the details that the response was written over SPDY.

Now with the two handlers in place it's time to implement the logic, which will choose the right one depending on what protocol is detected. But adding the previous written handler to the `ChannelPipeline` is not enough, as the correct codecs needs to also be added. Its responsibility is to detect the transmitted bytes and finally allow working with the `FullHttpResponse` and `FullHttpRequest` abstraction.

Fortunately this easier than it sounds, as Netty ships with a base class which does exactly this. All you need to do is to implement the logic to select the Protocol and which handler to use for HTTP and which for SPDY.

Listing 12.4 shows the implementation, which makes use of the abstract base class provided by Netty.

Listing 12.4 Implementation which handles SPDY

```
public class DefaultSpdyOrHttpChooser extends SpdyOrHttpChooser {

    public DefaultSpdyOrHttpChooser(int maxSpdyContentLength,
int maxHttpContentLength) {
        super(maxSpdyContentLength, maxHttpContentLength);
    }

    @Override
    protected SelectedProtocol getProtocol(SSLEngine engine) {
        DefaultServerProvider provider =
(DefaultServerProvider) NextProtoNego.get(engine);                       #1
        String protocol = provider.getSelectedProtocol();
        if (protocol == null) {
            return SelectedProtocol.UNKNOWN;                             #2
        }

        switch (protocol) {
            case "spdy/2":
                return SelectedProtocol.SPDY_2;                          #3
            case "spdy/3":
                return SelectedProtocol.SPDY_3;                          #4
            case "http/1.1":
                return SelectedProtocol.HTTP_1_1;                        #5
            default:
```

```
                return SelectedProtocol.UNKNOWN;                          #6
            }
        }

        @Override
        protected ChannelInboundHandler createHttpRequestHandlerForHttp() {
            return new HttpRequestHandler();                              #7
        }

        @Override
        protected ChannelInboundHandler createHttpRequestHandlerForSpdy() {
            return new SpdyRequestHandler();                              #8
        }
}
```

#1 Use he NextProtoNego to obtain a reference to the DefaultServerProvider which is in use for the SslEngine
#2 The protocol could not be detected. Once more bytes are ready to read the detect process will start again
#3 SPDY Draft 2 was detected
#4 SPDY Draft 3 was detected
#5 HTTP 1.1 was detected
#6 No known protocol detected
#7 Will be called to add the handler which will handle FullHttpRequest messages. This method is only called if no SPDY is supported by the client and so HTTPS should be used
#8 Will be called to add the handler which will handle FullHttpRequest messages. This method is called if SPDY is supported.

The implementation shown in 12.4 will take care of detecting the correct protocol and setting up the `ChannelPipeline` for you. It only handles SPDY 2 / 3 and HTTP 1.1 but could be adjusted to only support a specific version of SPDY etc.

12.3.3 Setuping the ChannelPipeline

With this we have everything in place and just need to wire the handlers together. This is done by an implementation of `ChannelInitializer` as usual. As we learned before the responsible for implementations of this is to setup the `ChannelPipeline` and so add all needed `ChannelHandlers` into it.

Because of the nature of SPDY those are actual two:

1. The `SslHandler`, as if you remember the detection of SPDY is done via a TLS Extension.
2. Our `DefaultSpdyOrHttpChooser`, which will add the correct `ChannelHandlers` into the `ChannelPipeline` once the protocol, was detected.

Besides adding the `ChannelHandlers` to the `ChannelPipeline`, it has also another responsibility. It assigns the previous created `DefaultServerProvider` to the `SslEngine` which is used by the `SslHandler` in the `ChannelPipeline`. This is done by making use of the `NextProtoNego` helper class that is provided by the jetty NPN library.

Listing 12.5 shows the implementation in detail.

Listing 12.5 Implementation which handles SPDY

```
public class SpdyChannelInitializer extends
ChannelInitializer<SocketChannel> {                                    #1

    private final SSLContext context;

    public SpdyChannelInitializer(SSLContext context) {                #2
        this.context = context;
    }

    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        SSLEngine engine = context.createSSLEngine();                  #3
        engine.setUseClientMode(false);                                #4

        NextProtoNego.put(engine, new DefaultServerProvider());        #5
        NextProtoNego.debug = true;

        pipeline.addLast("sslHandler", new SslHandler(engine));        #6
        pipeline.addLast("chooser",
new DefaultSpdyOrHttpChooser(1024 * 1024, 1024 * 1024));#7
    }
}}
```

#1 Extend ChannelIntializer for an easy starting point
#2 Pass the SSLContext in which will be used to create the SSLEngines from
#3 Create a new SSLEngine which will be used for the new Channel / Connection
#4 Configure the SSLEngine for non-client usage
#5 Bind the DefaultServerProvider to the SSLEngine via the NextProtoNego helper class
#6 Add the SslHandler into the ChannelPipeline, this will stay in the ChannelPipeline even after the protocol was detected.
#7 Add the DefaultSpyOrHttpChooser into the ChannelPipeline. This implementation will detect the protocol, add the correct ChannelHandlers in the ChannelPipeline and remove itself.

The "real" `ChannelPipeline` setup is done later by the `DefaultSpdyOrHttpChooser` implementation, as it is only possible to know at this point if the client supports SPDY or not.

To illustrate this let us recap and have a look at the different `ChannelPipeline` "states" during the "life-time" of the connection with the client.

Figure 12.2 shows the `ChannelPipeline`, which will be used after the `Channel` is initialized.

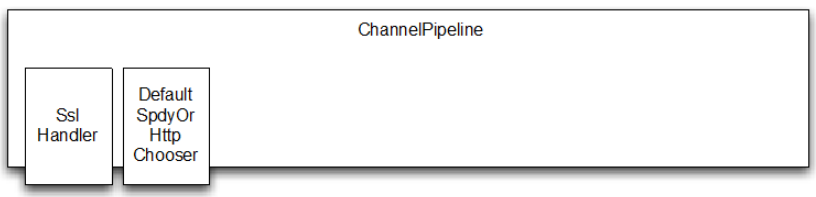


Figure 12.2 ChannelPipeline after connect

Now depending on if the client supports SPDY or not it will get modified by the `DefaultSpdyOrHttpChooser` to handle the protocol. The `DefaultSpdyOrHttpChooser` does this by first adding the required `ChannelHandlers` to the `ChannelPipeline` and then removing itself from the `ChannelPipeline` (as it is no longer needed). All this logic is encapsulated and hidden by the abstract `SpdyOrHttpChooser` class, from which `DefaultSpdyOrHttpChooser` extend.

Figure 12.3 shows the `ChannelPipeline`, which will be used if SPDY is usable for the connected client.

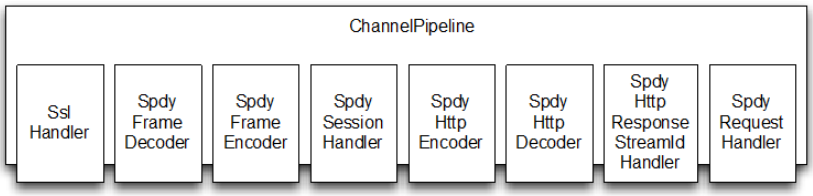


Figure 12.3 ChannelPipeline if SPDY is supported

All of the added `ChannelHandlers` have a responsibility and fulfill a small part of the work. This match perfectly the way you normally write Netty based Applications. Every "piece" does a small amount of work making it re-usable and more generic.

The responsibilities of the `Channelhandlers` are shown in Table 12.1.

Table 12.1 Responsibilities of the ChannelHandlers

Name	Responsibility
`SslHandler`	Encrypt / Decrypt data which is exchanged between the two peers
`SpdyFrameDecoder`	Decode the received bytes to SPDY frames
`SpdyFrameEncoder`	Encoder SPDY frames back to bytes
`SpdySessionHandler`	SPDY session handling
`SpdyHttpEncoder`	Encoder HTTP messages to SPDY frames
`SpdyHttpDecoder`	Decoder SDPY frames into Http messages
`SpdyHttpResponseStreamIdHandler`	Handle the mapping between requests and responses based on the SPDY id
`SpdyRequestHandler`	Handle the FullHttpRequests, which were decoded from the SPDY frame and so allow transparent usage of SPDY.

Like said before the `ChannelPipeline` will look different for when HTTP(s) is in use as the protocol, which is exchanged differs. Figure 13.4 shows the `ChannelPipeline` in this case.

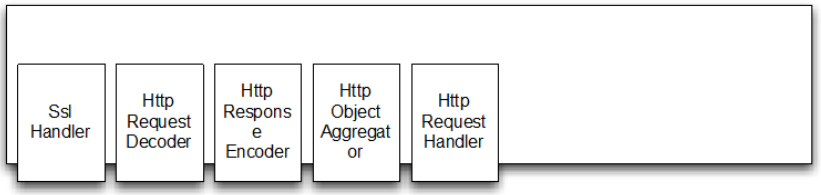


Figure 12.3 ChannelPipeline if SPDY is not supported

As before each of the `ChannelHandlers` has a responsibility, which is highlighted in Table 12.2.

Table 12.2 Responsibilities of the ChannelHandlers

Name	Responsibility
`SslHandler`	Encrypt / Decrypt data which is exchanged between the two peers
`HttpRequestDecoder`	Decode the received bytes to Http requests
`HttpResponseEncoder`	Encode Http responses to bytes
`HttpObjectAggregator`	SPDY session handling
`HttpRequestHandler`	Handle the FullHttpRequests, which were decoded

12.3.4 Wiring things together

After all the **`ChannelHandler`** implementations are prepared the only part, which is missing is the implementation that wires all the previous written implementation together and bootstrap the Server.

For this purpose the **`SpdyServer`** implementation is provided as shown in Listing 12.6.

Listing 12.6 SpdyServer implementation

```
public class SpdyServer {

    private final NioEventLoopGroup group = new NioEventLoopGroup();     #1
    private final SSLContext context;
    private Channel channel;

    public SpdyServer(SSLContext context) {                              #2
        this.context = context;
    }

    public ChannelFuture start(InetSocketAddress address) {
        ServerBootstrap bootstrap  = new ServerBootstrap();              #3
        bootstrap.group(group)
                .channel(NioServerSocketChannel.class)
```

```
                .childHandler(new SpdyChannelInitializer(context));     #4
        ChannelFuture future = bootstrap.bind(address);                 #5
        future.syncUninterruptibly();
        channel = future.channel();
        return future;
    }

    public void destroy() {                                             #6
        if (channel != null) {
            channel.close();
        }
        group.shutdownGracefully();
    }

    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Please give port as argument");
            System.exit(1);
        }
        int port = Integer.parseInt(args[0]);

        SSLContext context = BogusSslContextFactory.getServerContext(); #7

        final SpdyServer endpoint = new SpdyServer(context);
        ChannelFuture future = endpoint.start(new InetSocketAddress(port));

        Runtime.getRuntime().addShutdownHook(new Thread() {
            @Override
            public void run() {
                endpoint.destroy();
            }
        });
        future.channel().closeFuture().syncUninterruptibly();
    }
}
```

#1 Construct a new NioEventLoopGroup which handles the IO
#2 Pass in the SSLContext to use for encryption
#3 Create a new ServerBootstrap to configure the server
#4 Configure the ServerBootstrap
#5 Bind the server so it accept connections
#6 Destroy the server which means it close the channel and shutdown the NioEventLoopGroup
#7 Obtain a SSLContext from BogusSslContextFactory. This is a dummy implementation which can be used for testing purposes. For a real implementation you would configure the SslContext to use via a proper KeyStore. Refer to the java API documentation for details.

So to be understand what it does let us have a look about the "flow":

1. The `ServerBootstrap` is created and configured with the previous created `SpdyChannelInitializer`. The `SpdyChannelInitializer` sets up the `ChannelPipeline` once the Channel was accepted
2. Bind to the given `InetSocketAddress` and so accept connections on it.

Beside it also register a shutdown hook to release all resource once the JVM exists.

Now it's time to actual test the `SpdyServer` and see if it works like expected.

12.4 Start the SpdyServer and test it

One thing that is important when you work with the jetty NPN library is that you need to pass a special argument to the JVM when starting it. Otherwise it will not work, this is needed because of how it hooks into the `SslEngine`. So while we won't go into details how you compile the code here be aware that once it is compiled and you start it via the java command you will need to take special care here ensuring you set the `"-Xbootclasspath"` JVM argument to the path of the npn boot JAR file.

The "-Xbootclasspath" option allows to override standard implementations of classes that are shipped with the JDK itself.

Listing 12.7 shows the special argument (-Xbootclasspath) to use.

Listing 12.7 SpdyServer implementation

```
java -Xbootclasspath/p:<path_to_npn_boot_jar> ….
```

The easiest way to start up the application is to use the source-code provided for this book. It has everything ready to use and will use maven automatically to setup everything and download dependencies. Starting it up via mvn is as easy as shown in Listing 12.8

Listing 12.8 Compile and start SpdyServer with maven

```
Normans-MacBook-Pro:netty-in-action-privatenorman$ mvn clean package exec:exec -
      Pchapter12-SpdyServer
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building netty-in-action 0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
...
...
...
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action ---
[INFO] Building jar: /Users/norman/Documents/workspace-intellij/netty-in-action-
      private/target/netty-in-action-0.1-SNAPSHOT.jar
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ netty-in-action ---
```

Once the Server is successfully started it's time to actual test it. For this purpose we will use two different Browsers. A Browser that supports SPDY and another Browser that does not. At the time of writing this is Google Chrome (supports SPDY) and Safari (not supports SPDY). But it would also be possible to use Google Chrome and for example a unix tool like **`wget`** to test for fallback to HTTPS. You may get a "warning" if you use a self-signed certificate, but that's ok. For a production system you probability want to buy an official certificate.

Open up Google Chrome and navigate to the url "https://127.0.0.1:9999" or the matching ipaddress / port that you used.

Figure 12.4 shows it in action.

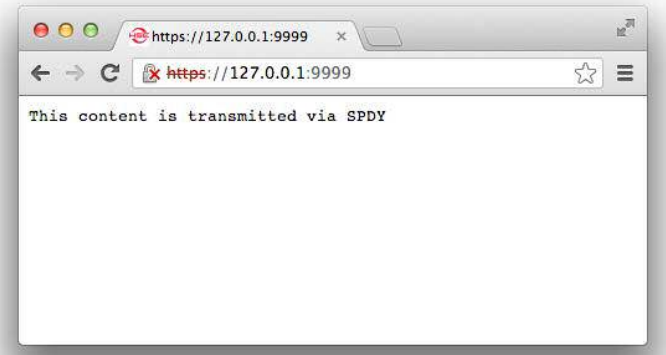


Figure 12.4 SPDY supported by Google Chrome

If you look at Figure 12.4 you should see that it's served via SPDY, as it use the previous created `SpdyRequestDecoder` and so writes "This content is transmitted via SPDY" back to the client.

A nice feature in Google Chrome is that you can check some statistics about SPDY with it and so get a deeper understanding what is transmitted and how many connections are currently handled by SPDY. For this purpose open the url "chrome://net-internals/#spdy".

Figure 12.5 shows the statistics, which are exposed here.

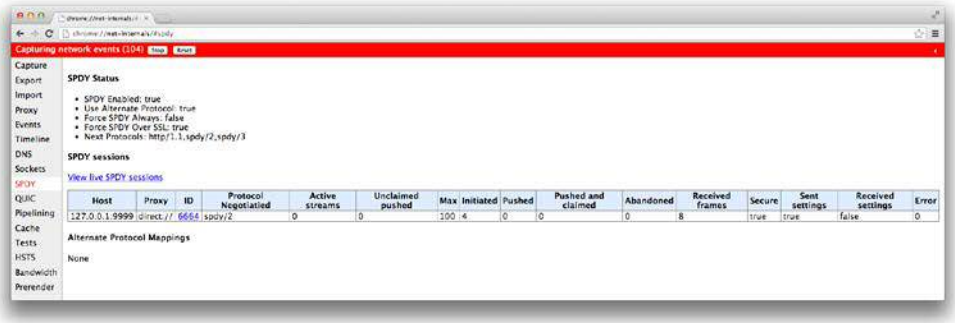


Figure 12.5 SPDY statistics

This gives you an overview over all the SPDY connections, which are established. And some more details about how many bytes are transmitted and many more. To get an even more detailed overview about the transmitted data click on the "ID" link and refresh the

window in which you haven opened “https://127.0.0.1:9999”. This will give you details about the different SPDY frames, which are transmitted to fulfill the request.

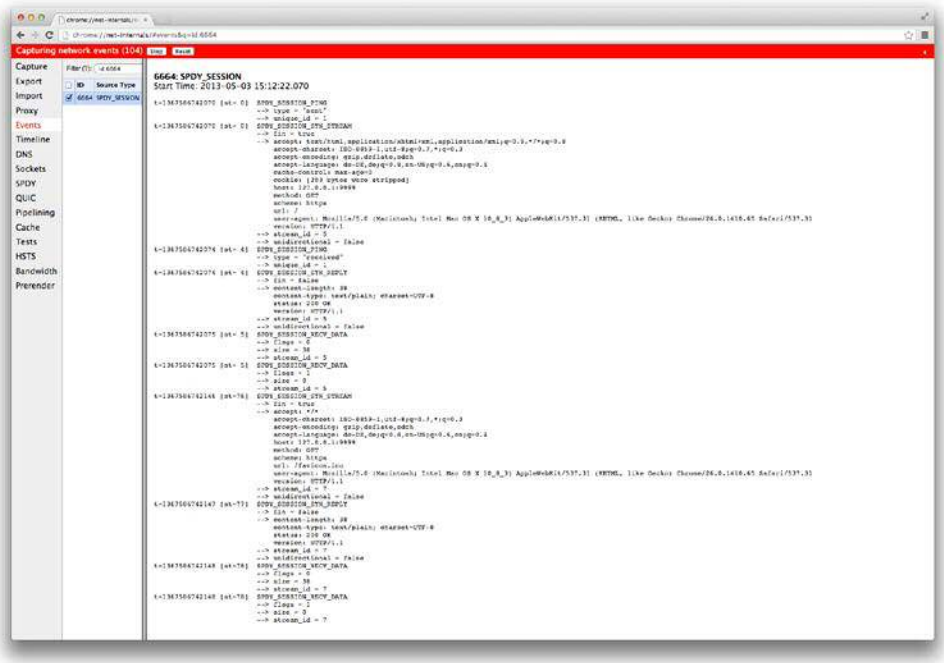


Figure 12.6 SPDY statistics

Going through all the details displayed in Figure 12.6 is out of scope of this book, but you should be able to spot easily some details about which HTTP headers are transmitted and which url was accessed. These kinds of statistics are quite useful if you debug SPDY in general.

What about the clients that do not support SPDY? Those will just be served via HTTPS and so should not have a problem to access the resources provided by the `SpdyServer`.

To test this open up a Browser, that does not support SPDY, which is Safari in our case. Like before access the url “https://127.0.0.1” and see what response is written back the client (browser).

Figure 12.7 shows it.

Figure 12.7 SPDY not supported by Safari

Figure 12.7 makes it clear that it's served via HTTP(s) as it writes back "This content is transmitted via HTTP" to the client. This is part of the `HttpRequestHandler` and so should be quite familiar to you.

12.5 Summary

In this chapter you learned how easy it is to use SPDY and HTTP(s) at the same time in a Netty based application. The shown application should give you a good starting point to support new clients (with support of SPDY) and the best performance while also allowing "old" clients to access the application.

You learned how you make use of the provided helper classes for SPDY in Netty and so re-use what is provided out of the box. Also you saw how you can use Google Chrome to get more information about SPDY connections and so understand what happens under the hood.

The shown techniques should help you and adapt them in your next Netty based application. Also you saw again how modification of the `ChannelPipeline` could help you to build powerful "multiplexers" which allow you to switch between different protocols during one connection.

In the next chapter you will learn how you can use UDP to write apps and make use of its high-performance connection-less nature.

13

Broadcasting events via UDP

In this Chapter

- UDP in general
- ChannelHandler, Decoder and Encoder
- Bootstrap an Netty based Application

The previous chapters have focused on examples, which use connection based protocols such as TCP. In this chapter we will focus on using UDP. UDP itself is a connection-less protocol, which is normally used if performance is most critical and loosing packets is not an issue. One of the most well known UDP based protocols is DNS, which is used for domain name resolution. The domain name resolution allows you to be able to use domain names (i.e. netty.io) and not need to type in the ipaddress directly.

Thanks to its unified Transport API, Netty makes writing UDP based applications easy as taking a breath. This allows you to reuse existing `ChannelHandlers` and other utilities that you have written for other Netty based Applications.

After this chapter you will know what is special about Connection-less Protocols, in this case UDP. And why UDP may be a good fit for your Application.

This is a self-contained chapter. You will be able to go through the chapter even without reading other chapters of the book first. It should contain everything you need to get the example application running an gain an understanding of UDP in Netty. While the chapter is self-contained, we won't re-iterate previous topics in great detail. Knowing the basics of the Netty API (as covered in earlier chapters) is more than sufficient for you to get through.

13.1 Handle UDP

Before we dive into the application the rest of this chapter focuses on, we're take a moment to look at UDP, what it is and the various limitations (or at least issues) that you should be aware of before using it.

As stated before UDP is connection-less. Which means that there is no "connection" for the lifetime of the communication between clients and servers. So it is not so easy to say all those messages belong to "one connection".

Another "limitation" is that UDP has no "error correction" like TCP. In TCP if packets are lost during transmission the peer that sends it will know because the receiver sends an acknowledgement of packets received. Failure to receive an acknowledgement causes the peer to re-sent the same packets again. This is not true for UDP. UDP is more like "fire and forget", which means it has no idea if the data sent was received at all.

Those "limitations" are one of the reasons why UDP is so fast compared to TCP. As it eliminates all the overhead of handshaking and other overhead that comes with TCP (like acknowledge of received data).

Knowing this, should hopefully point out to you that UDP is only a good fit for applications that can "handle" the case of lost messages. Which means it is not a very good fit for applications that can't lose any message, like an application that handles money transactions.

In the next section we're going to design the UDP application we use for the rest of this chapter.

13.2 UDP Application structure & Design

The rest of this chapter will write an application, which will broadcast messages via UDP to another part, which will receive them. In this case the application opens up a file and broadcasts every new line via UDP.

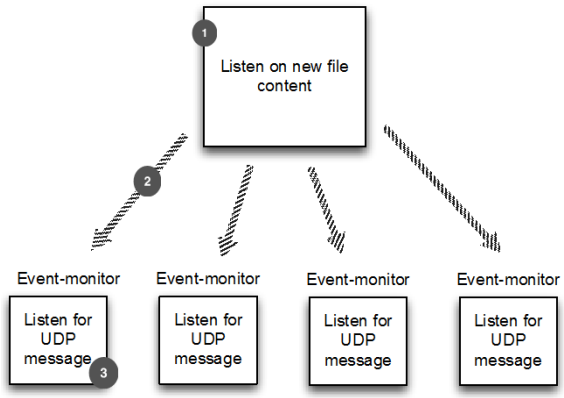
If you are familiar with UNIX-like operation systems this may sound very familiar to you because this is what Syslog does. In fact this is a simplified version of it.

UDP is a perfect fit for such an application as it may not be to dramatic if you loose one log line as it is stored on the base-system anyway. More importantly for the system may be the ability to send a lot of "events" over the network.

Also one additional advantage is that by using UDP broadcast there is no extra actions needed for adding new "monitors" that receive the log messages via UDP. All that is needed is to fire up the instance on the specific port and you will see the log messages flow in. But this is not only a "good thing", as everyone will be able to get the log messages once he is able to start an instance. This is one of the reasons why UDP broadcast should only be used in secure environments / networks where such things are not an issue. Also be aware that broadcast messages usually only work in the same network as routers will usually block them. These types of applications are typically classified as "pub-sub", where one side of the application or service, "publishes" events and one or more others are able to "subscribe" to these events.

Before we go any further, let's take a high-level look at the application we're going to build.

Figure 13.1 shows an overview over the application layout



#1 Application listen for new file content
#2 Broadcast file content via UDP
#3 Print out new log content

Figure 13.1 Application overview

So the Application will consist of two parts. The log file broadcaster and the "monitor", which will receive the broadcast. Where there could be more then one receiver.

To keep things simple we will not do any kind of authentication, verification or encryption.

If you feel the application may be a good fit for one of your needs you can adjust it for your needs later by replacing parts of it. This should be quite simple as the logic is broken down into easy to follow sections.

The next section will give you some insight about what you need to know about UDP and what is the difference compared to writing TCP – protocol –based applications.

13.3 EventLog POJOs

As in other applications that are not based on Netty, you usually have some kind of "message POJO" that is used to hold application-based information to pass around. Think of it as an "Event message".

For our Application we will do this too and share it between both parts of the Application. We call it `LogEvent`, as it stores the event data, which was generated out of the Log file.

Listing 13.1 shows the simple POJO.

Listing 13.1 LogEvent message

```
public final class LogEvent {
        public static final byte SEPARATOR = (byte) ':';               #1

    private final InetSocketAddress source;
    private final String logfile;
    private final String msg;
    private final long received;

    public LogEvent(String logfile, String msg) {                       #1
        this(null, -1, logfile, msg);
    }

    public LogEvent(InetSocketAddress source, long received,
String logfile, String msg) {                                           #2
        this.source = source;
        this.logfile = logfile;
        this.msg = msg;
        this.received = received;
    }

    public InetSocketAddress getSource() {                              #3
        return source;
    }

    public String getLogfile() {                                        #4
        return logfile;
    }

    public String getMsg() {                                            #5
        return msg;
    }

    public long getReceivedTimestamp() {                                #6
```

```
            return received;
        }
    }
```

#1 Instance new LogEvent for sending purposes
#2 Instance new LogEvent for receiving pruposes
#3 Get the InetSocketAddress of the source which send the LogEvent
#4 Get the name of the log file for which the LogEvent was send
#5 Get the actual log message of the LogEvent
#6 Get the timestamp at which the LogEvent was received

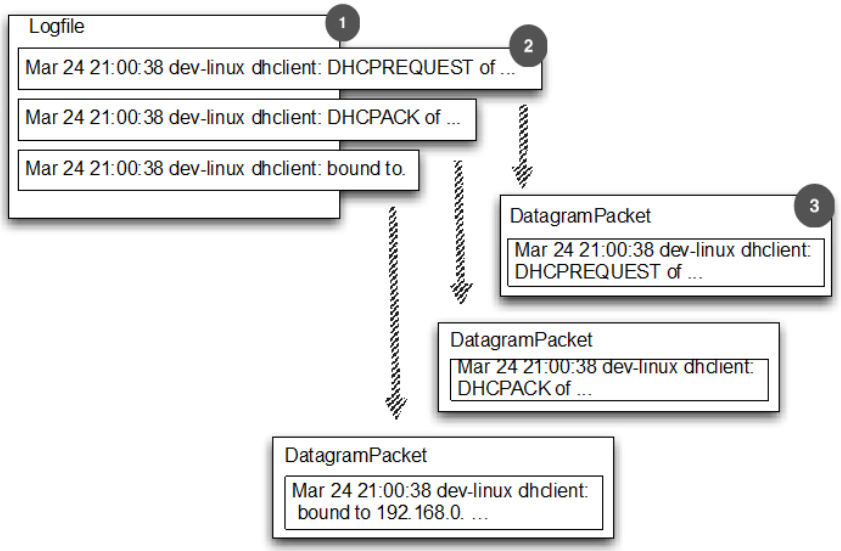
With the common / shared part ready it's time to implement the actual logic. In the next sections we will focus on doing this.

We will start with writing the "Broadcaster" in the next section and explain how it works exactly.

13.4 Writing the "Broadcaster"

Like illustrated before one "part" of our Application is the Log broadcaster. In this section we will go through all the steps that are needed to write it.

What we want to do is to broadcast one `DatagramPacket` per Log entry as shown in Figure 13.2.



#1 The Log file
#2 A log entry in the Log file
#3 The DatagramPacket which holds the log entry

Figure 13.2 Log entry to DatagramPacket

As shown in figure 13.2 this means we have a 1-to-1 relation from Log entries to `DatagramPackets`.

First off let us have a look at the `ChannelPipeline` of the `LogEventBroadcaster` and how the `LogEvent` (which was created before) will flow through it. Figure 13.3 gives some high-level overview.

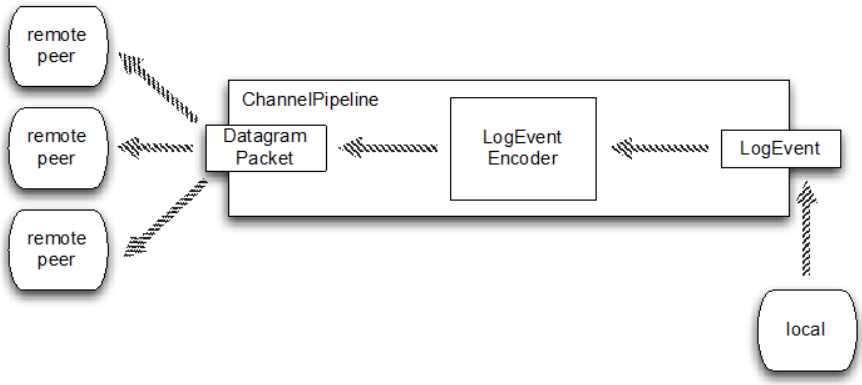


Figure 13.3 LogEventBroadcasterChannelPipeline and LogEvent flow

So let's recap on Figure 13.3. The `LogEventBroadcaster` use `LogEvent` messages and write them via the Channel on the local side. This means all the information is encapsulated in the `LogEvent` messages. These messages are then sent trough the `ChannelPipeline`.

While flowing through the `ChannelPipeline` the `LogEvent` messages are encoded to `DatagramPacket` messages, which finally are broadcasted via UDP to the remote peers.

This boils down to the need of having one custom `ChannelHandler`, which encodes from `LogEvent` messages to `DatagramPacket` messages. Remembering what we learned in Chapter 7 (Codec API) it should be quite clear what kind of abstract base class for the `LogEventEncoder` should be used to keep the amount of code to be written to the minimum and so make it easier to maintain. If you've not read Chapter 7 yet, don't worry. You will be able to go through this chapter without it.

Listing 13.2 shows how the encoder was implemented.

Listing 13.2 LogEventEncoder

```
public class LogEventEncoder extends MessageToMessageEncoder<LogEvent> {
    private final InetSocketAddress remoteAddress;

    public LogEventEncoder(InetSocketAddress remoteAddress) {           #1
        this.remoteAddress = remoteAddress;
    }

    @Override
    protected void encode(ChannelHandlerContext channelHandlerContext,
            LogEvent logEvent, List<Object> out) throws Exception {
```

```
        ByteBuf buf = channelHandlerContext.alloc().buffer();
        buf.writeBytes(logEvent.getLogfile()
                .getBytes(CharsetUtil.UTF_8));#2
        buf.writeByte(LogEvent.SEPARATOR);                              #3
        buf.writeBytes(logEvent.getMsg().getBytes(CharsetUtil.UTF_8));  #4

        out.add(new DatagramPacket(buf, remoteAddress));                #5
    }
}
```

#1 Create a new LogEventEncoder which will create DatagramPacket messages that will send to the given InetSocketAddress
#2 Write the filename in the ByteBuf
#3 Separate it via the SEPARATOR
#4 Write the actual log message to the ByteBuf
#5 Add the DatagramPacket to the List of encoded messages.

Why MessageToMessageEncoder?

`MessageToMessageEncoder` was used in Listing 13.2, as the implementation needs to encode one Message type to another one. `MessageToMessageEncoder` allows this in an easy way and so offer an easy-to-use abstract. But keep in mint `MessageToMessageEncoder` is just a `ChannelOutboundHandler` implementation, so you could also just have implement the `ChannelOutboundHandler` interface directly. `MessageToMessageEncoder` just makes it a lot easier.

After we were able to build the `LogEventEncoder` we need to bootstrap the Application and use it. Bootstrapping is the part of the Application, which is responsible to configure the actual `Bootstrap`. This includes set various `ChannelOptions` and some `ChannelHandlers` that are used in the `ChannelPipeline` of the `Channel`.

This is done in Listing 13.3 by our `LogEventBroadcaster` class.

Listing 13.3LogEventBroadcaster

```
public class LogEventBroadcaster {
    private final EventLoopGroup group;
    private final Bootstrap bootstrap;
    private final File file;

    public LogEventBroadcaster(InetSocketAddress address,
File file) {
        group = new NioEventLoopGroup();
        bootstrap = new Bootstrap();
            bootstrap.group(group)
                    .channel(NioDatagramChannel.class)
                    .option(ChannelOption.SO_BROADCAST, true)
                    .handler(new LogEventEncoder(address));             #1

        this.file = file;
    }

    public void run() throws IOException {
```

```
        Channel ch = bootstrap.bind(0).syncUninterruptibly().channel(); #2
        long pointer = 0;
        for (;;) {
            long len = file.length();
            if (len < pointer) {
                // file was reset
                pointer = len;                                          #3
            } else if (len > pointer) {
                // Content was added
                RandomAccessFile raf = new RandomAccessFile(file, "r");
                raf.seek(pointer);                                      #4
                String line;
                while ((line = raf.readLine()) != null) {
                    ch.writeAndFlush(new LogEvent(null, -1,
                            file.getAbsolutePath(), line));             #5
                }
                pointer = raf.getFilePointer();                         #6
                raf.close();
            }
            try {
                Thread.sleep(1000);                                     #7
            } catch (InterruptedException e) {
                Thread.interrupted();
                break;
            }
        }
    }

    public void stop() {
group.shutdown();
    }

    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            throw new IllegalArgumentException();
        }

        LogEventBroadcaster broadcaster = new LogEventBroadcaster(
new InetSocketAddress("255.255.255.255",
                Integer.parseInt(args[0])), new File(args[1]));         #8
        try {
            broadcaster.run();
        } finally {
            broadcaster.stop();
        }
    }
}
```

#1 Bootstrap the NioDatagramChannel. It is important to set the SO_BROADCAST option as we want to work with broadcasts
#2 Bind the Channel so we can use it. Note that there is no connect when use DatagramChannel as those are connectionless.
#3 Set the current filepointer to the last byte of the file as we only want to broadcast new log entries and not what was written before the application was started
#4 Set the current filepointer so nothing old is send
#5 Write a LogEvent to the Channel which holds the filename and the log entry (we exepect every logentry is only one line long)
#6 Store the current position within the file so we can later continue here
#7 Sleep for 1 second and then check if there is something new in the logfile which we can send

#8 Construct a new instance of LogEventBroadcaster and start it.

This is the point where we have the first part of our Application done. We can see the first results at this point, event without the second part. For this let us fire up the Netcat program, which should be installed if you use a UNIX-like OS or can be installed on windows via the available installer[34]

Netcat allows you to listen on a specific port and just print all data received to STDOUT. This is perfect to test-drive our implementation.

Listing 13.4 shows how you can start Netcat to listen for UDP data on port 9090.

Listing 13.4Netcat in action

```
normans-macbook-pro: norman$ nc -l -u 9090
```

After this is done it's time to startup `LogEventBroadcaster` pointing it to a log file that changes in short intervals and using port 9090.

Once the log file gets a new entry written it will broadcast the entry via UDP, and so it will be written out in the console in which you have started the `nc` command before.

Using Netcat for testing purposes is an easy thing, but not a production System. This brings us the second part of the Application, which we will implement in the next section of this Chapter.

13.5 Writing the "monitor"

Having the first part of the application in place it's time to replace our Netcat usage with some real code. For this we will develop the `EventLogMonitor` program in this section.

This program will do the following:

- Receive UDP `DatagramPackets` that was broadcasted via the `LogEventBroadcaster`
- Decode them to `LogEvent` messages
- Handle the `LogEvent` messages by write them to `System.out`

To handle the logic we will write our own `ChannelHandler` implementations again. Let us have a look on the `ChannelPipeline` of the `LogEventMonitor` and how the `LogEvent` (which was created before) will flow through it.

Figure 13.4 shows the details.

[34] http://nmap.org/ncat/

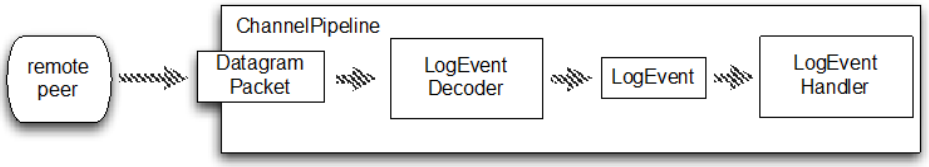


Figure 13.4LogEventMontorChannelPipeline and LogEvent flow

In Figure 13.4 you see there are two custom `ChannelHandler` implementations present. Let us give a spin on implement both of them.

The first one to implement is the `LogEventDecoder`. The `LogEventDecoder` is responsible to decode `DatagramPackets` that are received over the network into `LogEvent` messages. Again this is how most of your Netty Applications will start if you receive data on which you operate.

Listing 13.4 shows how the encoder implementation looks like.

Listing 13.4 LogEventDecoder

```
public class LogEventDecoder extends MessageToMessageDecoder<DatagramPacket> {


        @Override
        protected void decode(ChannelHandlerContext ctx,
                    DatagramPacket datagramPacket, List<Object> out)
                throws Exception {
            ByteBuf data = datagramPacket.data();                          #1
            int i = data.indexOf(0, data.readableBytes(),
                     LogEvent.SEPARATOR);                                  #2
            String filename = data.slice(0, i)
                     .toString(CharsetUtil.UTF_8);                         #3
            String logMsg =  data.slice(i + 1,
                     data.readableBytes()).toString(CharsetUtil.UTF_8);    #4

            LogEvent event = new LogEvent(datagramPacket.remoteAddress(),
                     System.currentTimeMillis(),filename,logMsg);          #5
            out.add(event);
        }
    }
```

#1 Get a reference to the data in the DatagramPacket
#2 Get the index of the SEPARATOR
#3 Read the filename out of the data
#4 Read the log message out of the data
#5 Construct a new LogEvent object and finally return it

Having the decoder in place it is time to implement the actual `ChannelHandler` that will do something with the `LogEvent` messages. In our case we just want them to get written to `System.out`. In a real-world application you may want to write them in some aggregated log file, database or do something useful with it. This is just to show you the moving parts.

Listing 13.5shows the `LogEventHandler` that writes the content of the `LogEvent` to `System.out`.

Listing 13.4 LogEventHandler

```
public class LogEventHandler
            extends SimpleChannelInboundHandler<LogEvent> {             #1

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx,
            Throwable cause) throws Exception {
        cause.printStackTrace();                                        #2
        ctx.close();
    }

    @Override
    public void channelRead0(ChannelHandlerContext ctx,
LogEvent event) throws Exception {
        StringBuilder builder = new StringBuilder();                    #3
        builder.append(event.getReceivedTimestamp());
        builder.append(" [");
        builder.append(event.getSource().toString());
        builder.append("] [");
        builder.append(event.getLogfile());
        builder.append("] : ");
        builder.append(event.getMsg());

        System.out.println(builder.toString());                         #4
    }
}
```

#1 Extend SimpleChannelInboundHandler to handle LogEvent messages
#2 Print the Stacktrace on error and close the channel
#3 Create a new StringBuilder and append the infos of the LogEvent to it
#4 Print out the LogEvent data

The `LogEventHandler` prints the `LogEvents` out in an easy to read format which consist out of those:

- Received timestamp in milliseconds
- `InetSocketAddress` of the sender which consist of ipaddress and port
- The absolute name of the file which the `LogEvent` was generated from
- The actual log message, which represent on line in the log file.

Having now written the decoder and the handler, it's time to assemble the parts to the previously outlined `ChannelPipeline`.

Listing 13.5 shows how this is done as part of the `LogEventMonitor` class.

Listing 13.5 LogEventMonitor

```
public class LogEventMonitor {
    private final EventLoopGroup group;
    private final Bootstrap bootstrap;
```

```
    public LogEventMonitor(InetSocketAddress address) {
        group = new NioEventLoopGroup();
        bootstrap = new Bootstrap();
        bootstrap.group(group)
                .channel(NioDatagramChannel.class)
                .option(ChannelOption.SO_BROADCAST, true)
                .handler(new ChannelInitializer<Channel>() {
                    @Override
                    protected void initChannel(Channel channel)
throws Exception {
                        ChannelPipeline pipeline = channel.pipeline();
                        pipeline.addLast(new LogEventDecoder());
                        pipeline.addLast(new LogEventHandler());
                    }
                }).localAddress(address);                                #1

    }

    public Channel bind() {
        return bootstrap.bind().syncUninterruptibly().channel();         #2
    }

    public void stop() {
        group.shutdownGracefully();
    }

    public static void main(String[] main) throws Exception {
        if (args.length != 1) {
            throw new IllegalArgumentException(
                    "Usage: LogEventMonitor <port>");
        }
        LogEventMonitor monitor = new LogEventMonitor(
                new InetSocketAddress(args[0]));                         #3
        try {
            Channel channel = monitor.bind();
            System.out.println("LogEventMonitor running");

            channel.closeFuture().syncUninterruptibly();
        } finally {
            monitor.stop();
        }
}
}
```

#1 Bootstrap the NioDatagramChannel. It is important to set the SO_BROADCAST option as we want to work with broadcasts
#2 Bind the Channel so we can use it. Note that there is no connect when use DatagramChannel as those are connectionless.
#3 Construct a new LogEventMonitor and use it.

13.6 Using the LogEventBroadcaster and LogEventMonitor

Having written our Application its time to see it in action. The easiest way is again to use maven. Please see Chapter 2 if you need some guidance how to setup maven and the other parts to run the code as this is out of scope of this chapter.

You will need to open up two different console windows as both of the Applications will keep running until you stop them via "CRTL+C".

First off let us start `LogEventBroadcaster`, which will broadcast the actual log messages via UDP.

Listing 13.5 shows how this can be done using the `mvn` command.

Listing 13.5 Compile and start the LogEventBroadcaster

```
Normans-MacBook-Pro:netty-in-action-private norman$ mvn clean package exec:exec -
      Pchapter13-LogEventBroadcaster
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building netty-in-action 0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
...
...
...
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action ---
[INFO] Building jar: /Users/norman/Documents/workspace-intellij/netty-in-action-
      private/target/netty-in-action-0.1-SNAPSHOT.jar
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ netty-in-action -
-

 LogEventBroadcaster running
```

By default it will use "`/var/log/messages`" as log file and port 9999 to send the log messages to. If you want to change this you can do it with the log file and port System properties when starting the `LogEventBroadcaster`.

Listing 13.6 shows how you can use port 8888 and the log file "`/var/log/mail.log`".

Listing 13.6 Compile and start the LogEventBroadcaster

```
Normans-MacBook-Pro:netty-in-action-private norman$ mvn clean package exec:exec -
      Pchapter13-LogEventBroadcaster -Dport=8888 -Dlogfile=/var/log/mail.log
....
....
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ netty-in-action -
-

 LogEventBroadcaster running
```

When you see "`LogEventBroadcaster` running" you know it started up successfully. If any error was thrown it will print out an Exception.

Having the `LogEventBroadcaster` running it will start to write out log messages once those are written to the log file.

Get the Log messages sent via UDP is nice but without doing something with them its kind of useless. So time for us to start up the `LogEventMonitor` to receive and handle them. Again we will use maven for doing so as everything is prepared for it.

Listing 13.7 Compile and start the LogEventBroadcaster

```
Normans-MacBook-Pro:netty-in-action-private norman$ mvn clean package exec:exec -
      Pchapter13-LogEventMonitor
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building netty-in-action 0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
...
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action ---
[INFO] Building jar: /Users/norman/Documents/workspace-intellij/netty-in-action-
      private/target/netty-in-action-0.1-SNAPSHOT.jar
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ netty-in-action ---
LogEventMonitor running
```

When you see "`LogEventMonitor` running" you know it started up successfully. If any error was thrown it will print out an Exception. Now watch the console and see how new log messages will start to get printed in it once they are written to the log file.

Listing 13.8 shows how such an output will look like.

Listing 13.8 LogEventMonitor output

```
1364217299382 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 13:55:08 dev-linux
      dhclient: DHCPREQUEST of 192.168.0.50 on eth2 to 192.168.0.254 port 67
1364217299382 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 13:55:08 dev-linux
      dhclient: DHCPACK of 192.168.0.50 from 192.168.0.254
1364217299382 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 13:55:08 dev-linux
      dhclient: bound to 192.168.0.50 -- renewal in 270 seconds.
1364217299382 [/192.168.0.38:63182] [[/var/log/messages] : Mar 25 13:59:38 dev-linux
      dhclient: DHCPREQUEST of 192.168.0.50 on eth2 to 192.168.0.254 port 67
1364217299382 [/192.168.0.38:63182] [/[/var/log/messages] : Mar 25 13:59:38 dev-linux
      dhclient: DHCPACK of 192.168.0.50 from 192.168.0.254
1364217299382 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 13:59:38 dev-linux
      dhclient: bound to 192.168.0.50 -- renewal in 259 seconds.
1364217299383 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 14:03:57 dev-linux
      dhclient: DHCPREQUEST of 192.168.0.50 on eth2 to 192.168.0.254 port 67
1364217299383 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 14:03:57 dev-linux
      dhclient: DHCPACK of 192.168.0.50 from 192.168.0.254
1364217299383 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 14:03:57 dev-linux
      dhclient: bound to 192.168.0.50 -- renewal in 285 seconds.
```

If you look at the output in the console you will notice it contains all the information of the `LogEvent` in the format like specified in the `LogEventHandler`.

If you have no "real" log file to use you can also create a custom file, which will fill content in by hand to see our Application in action.

For doing so we use UNIX commands. If you are using Windows you can just create an empty file with notepad editor. In Listing 13.10 we first use touch to create an empty log file, which we want to broadcast the `LogEvents` for.

Listing 13.10 Create logfile for testing

```
Normans-MacBook-Pro:netty-in-action-private norman$ touch ~/mylog.log
```

Now having the log file in place startup the `LogEventBroadcaster` again and point it to it. As you may remember you can set it via the log file System property.

Listing 13.11 shows it.

Listing 13.11 Use created logfile

```
Normans-MacBook-Pro:netty-in-action-private norman$ mvn clean package exec:exec -
      Pchapter13-LogEventBroadcaster -Dport=8888 -Dlogfile=~/mylog.log
....
```

Once the `LogEventBroadcaster` is running again you can start to fill the log file with messages and so see how they are broadcasted. Doing this is as easy as just using `echo` and directing the output to the file.

Listing 13.12 shows how this is done.

Listing 13.12 Fill logfile with data

```
Normans-MacBook-Pro:netty-in-action-private norman$ echo 'Test log entry' >>
      ~/mylog.log
```

You can repeat the `echo` as often as you like and see its input written in the console in which you run the `LogEventMonitor`.

One thing to keep in mind again is that you can start as many instances of the monitor as you like and watch all of them receive the same messages without the broadcaster Application changing at all.

13.7 Summary

In this chapter you learned how you will be able to use Netty to write UDP based Applications while still using the same API as when you use TCP as underlying Transport protocol. You also saw how you usually divide logic into different `ChannelHandler` and add them to the `ChannelPipeline`. We did this by separating the decoder logic from the actual logic that handles the message object.

You got an introduction what is special about connection-less protocols like UDP and what is needed to know to work with them and use them in your next application.

The next chapter will give you some deeper insight how you would implement your own codec with Netty and so make it possible to re-use it.

14

Implement a custom codec

In this Chapter

- `Decoder`
- `Encoder`
- `Unit testing`

This chapter will show you how you can easily implement custom codecs with Netty for your favorite protocol. Those codecs are easy to reuse and test, thanks to the flexible architecture of Netty.

To make it easier to gasp, Memcached is used as example for a protocol, which should be supported via the codec.

Memcached is "free & open source, high-performance, distributed memory object caching system, generic in nature, but intended for use in speeding up dynamic web applications by alleviating database load." Memcached is in effect an in-memory key-value store for small chunks of arbitrary data.

You may ask yourself "Why Memcached"? In short, simplicity and coverage. The Memcached protocol is very straightforward; this makes it ideal for inclusion in a single chapter, without let it become to big.

14.1 Scope of the codec

We will only implement a subset of the Memcached protocol, just enough for us to add, retrieve and remove objects. This is supported by the `SET`, `GET` and `DELETE` commands, which are part of the Memcached protocol.

Memcached supports many other commands, but focus on just three of them allows us to better explain what is needed while keep things straight. Everything learned here could be used to latter implement the missing commands.

Memcached has a binary and plain text protocol. Both of them can be used to communicate with a Memcached server, depending on if the server supports either one of them or all. This chapter focuses on implement the binary protocol as often users are faced with implement protocols that are binary.

14.2 Implementing the Memcached codec

Whenever you are about to implement a codec for a given protocol, you should spend some time to understand its workings. Often the protocol itself is documented in some great detail. How much detail you will find here depends. Fortunately the binary protocol of Memcached is written down in great extend.

The specification for this is written up in an RFC style document and available at the project homepage[35].

As mention before we won't implement the entire protocol in this chapter instead we're going to only implement three operations, namely `SET`, `GET` and `DELETE`. This is done to keep things simple and not extend the scope of the chapter. You can easily adapt the classes and so add support for other operations by yourself. The shown code serves as examples and should not be taken as a full implementation of the protocol itself.

14.3 Getting to Know the Memcached Binary Protocol

So as we said, we're going to implement Memcached's `GET`, `SET` and `DELETE` operations (also referred to interchangeably as opcodes). We'll focus on these but Memcached's protocol has a generic structure where only a few parameters change in order to change the meaning of a request or response. This means that you can easily extend the implementation to add other commands. In general the protocol has a 24 bytes header for requests and responses. This header can be broken down in exactly the same order as denoted by the "Byte offset" column in table 14.1.

Table 14.1 Sample Memcached header byte structure

Field	Byte offset	Value
Magic	0	0x80 for requests 0x81 for responses
OpCode	1	0x01...0x1A
Key length	2 and 3	1... 32,767
Extra length	4	0x00, x04 or 0x08
Data type	5	0x00
Reserved	6 and 7	0x00
Total body length	8 - 11	Total size of body inclusive extras
Opaque	12 - 15	Any signed 32 bit integer, this one will be also included in the response and so make it easier to map requests to responses.
CAS	16 - 23	Data version check

35 https://code.google.com/p/Memcached/wiki/MemcacheBinaryProtocol

Notice how many bytes are used in each section. This tells us what data type we should use later. i.e. if a byte offset uses only byte 0 then we use a Java byte to represent it, if it uses 6 and 7 (2 bytes) we use a Java short, if it uses 12-15 (4 bytes) we use a Java int and so on.

```
<30 new binary client connection.
30: going from conn_new_cmd to conn_waiting
30: going from conn_waiting to conn_read
30: going from conn_read to conn_parse_cmd
<30 Read binary protocol data:
<30    0x80 0x01 0x00 0x01
<30    0x08 0x00 0x00 0x00          1
<30    0x00 0x00 0x00 0x0c
<30    0x87 0x90 0xa7 0xd9
<30    0x00 0x00 0x00 0x00
<30    0x00 0x00 0x00 0x00
30: going from conn_parse_cmd to conn_nread
<30 SET a Value len is 3
> NOT FOUND a
>30 Writing bin response:
>30    0x81 0x01 0x00 0x00
>30    0x00 0x00 0x00 0x00
>30    0x00 0x00 0x00 0x00          2
>30    0x87 0x90 0xa7 0xd9
>30    0x00 0x00 0x00 0x00
>30    0x00 0x00 0x00 0x01
30: going from conn_nread to conn_mwrite
30: going from conn_mwrite to conn_new_cmd
30: going from conn_new_cmd to conn_waiting
30: going from conn_waiting to conn_read
```

#1 Request (only headers are shown)
#2 Response

Figure 14.2 – Real world Memcached request and response headers

In figure 14.2, the highlighted section 2 is the response and section 1 represents a request to Memcached (only the request header is shown), which in this case is telling Memcached to SET the key "a" to the value "abc".

Each row in the highlighted sections represents 4 bytes; since there are 6 rows this means the request header is made up of 24 bytes as we said before. Looking back at table 14.1, you can begin to see how the header information from that table is represented in a real request. For now, this is all we need to know about the Memcached binary protocol. In the next section we need to take a look at just how we can start making these requests with Netty.

14.4 Netty Encoders and DEcoders

Netty is a complex and advanced framework but it's not psychic. When we make a request to set some key to a given value, we now know that an instance of the Request class is created to represent this request. This is great and works for us but Netty has no idea how this Request object translates to what Memcached expects. And the only thing Memcached expects

is a series of bytes, regardless of the protocol you use, the data sent over the network is always a series of bytes.

To convert from a Request object to the series of bytes Memcached needs, Netty has what are known as encoders, so called because they encode things such as this `MemcachedRequest` class to another format. Notice I said another format, this is because encoders don't just encode from object to bytes, they can encode from one object to another type of object or from an object to a string and so on. Encoders are covered in greater detail later in chapter 7, including the various types Netty provides.

For now we're going to focus on the type that converts from an object to a series of bytes. To do this Netty offers an abstract class called `MessageToByteEncoder`. It provides an abstract method, which should convert a message (in this case our `MemcachedRequest` object) to bytes. You indicate what message your implementation can handle through use of Java's generics i.e. `MessageToByteEncoder<MemcachedRequest>` says this encoder encodes objects of the type `MemcachedRequest`.

MessageToByteEncoder and Generics

As said it's possible to use Generics to tell the `MessageToByteEncoder` to handle specific message type. If you want to handle many different message types with the same encoder you can also use `MessageToByteEncoder<Object>`. Just be sure to do proper instance checking of the message in this case.

All this is also true for decoders, except decoders convert a series of bytes back to an object .For this Netty provides the `ByteToMessageDecoder` class which instead of encode, provides a decode method. In the next two sections we'll see how we can implement a Memcached decoder and encoder. Before we do however, it's important to realize that you don't always need to provide your own encoders and decoders when using Netty. We're only doing it now because Netty does not have built in support for Memcached. If the protocol was HTTP or another one of the many standard protocols Netty support then there would already been an encoder and decoder provided by Netty.

Encoder vs. Decoder

Remember encoder handles outbound and decoder inbound. Which basically means the encoder will encode data which is about to get written to the remote peer . The Decoder will handle data which was read from the remote peer.

It's important to remember there are two different directions related to outbound and inbound.

Be aware that our encoder and decoder do not verify any values for max size to keep the implementation simple. In a real world implementation you would most likely want to place in some verification checks and raise an `EncoderException` or `DecoderException` (or a sub-class of them) if there is protocol violation was detected.

14.4.1 Implementing the Memcached Encoder

This section puts our brief introduction to encoders into action. As said before the encoder is responsible to encode a message into a series of bytes. Those bytes can then be sent over the wire to the remote peer. To represent a request we ill first create the `MemcachedRequest` class, which we later then encode into a series of bytes with the actual encoder implementation.

Listing 14.1 shows our `MemcachedRequest` class, which represent the request.

Listing 14.1 – Implementation of a Memcached request

```
public class MemcachedRequest {                                          #1
    private static final Random rand = new Random();
    private int magic = 0x80;//fixed so hard coded
    private byte opCode; //the operation e.g. set or get
    private String key; //the key to delete, get or set
    private int flags = 0xdeadbeef; //random
    private int expires; //0 = item never expires
    private String body; //if opCode is set, the value
    private int id = rand.nextInt(); //Opaque
    private long cas; //data version check...not used
    private boolean hasExtras; //not all ops have extras

    public MemcachedRequest(byte opcode, String key, String value) {
        this.opCode = opcode;
        this.key = key;
        this.body = value == null ? "" : value;
        //only set command has extras in our example
        hasExtras = opcode == Opcode.SET;
    }

    public MemcachedRequest(byte opCode, String key) {
       this(opCode, key, null);
   }

   public int magic() {                                                  #2
       return magic;
   }

   public int opCode() {                                                 #3
       return opCode;
   }

   public String key() {                                                 #4
       return key;
   }

   public int flags() {                                                  #5
       return flags;
   }
```

```
    public int expires() {                                          #6
        return expires;
    }

    public String body() {                                          #7
        return body;
    }

    public int id() {                                               #8
        return id;
    }

    public long cas() {                                             #9
        return cas;
    }

    public boolean hasExtras() {                                   #10
        return hasExtras;
    }
    }
```

#1 The class that represent a request which will be send to the Memcached server
#2 The magic number
ä3 The opCode which reflects for which operation the response was created
#4 The key for which the operation should be executed
#5 The extra flags used
#6 Indicate if the an expire should be used
#7 The body of any
#8 The id of the request. This id will be echoed back in the response.
#9 The compare-and-check value
#10 Returns true if any extras are used

The most important things about the `MemcachedRequest` class occur in the constructor, everything else is just there for support use later. The initialization that occurs here is reminiscent of what was shown in Table 14.1 earlier. In fact it is a direct implementation of the fields in table 14.1. These attributes are taken directly from the Memcached protocol specification.

For every request to set, get or delete, an instance of this `MemcachedRequest` class will be created. This means that should you want to implement the rest of the Memcached protocol you would only need to translate a "client.op*" (op* is any new operation you add) method to one of these requests. Before we move on to the Netty specific code, we need two more support classes.

Listing 14.2 – Possible Memcached operation codes and response statuses

```
public class Status {
    public static final short NO_ERROR = 0x0000;
    public static final short KEY_NOT_FOUND = 0x0001;
    public static final short KEY_EXISTS = 0x0002;
    public static final short VALUE_TOO_LARGE = 0x0003;
    public static final short INVALID_ARGUMENTS = 0x0004;
    public static final short ITEM_NOT_STORED = 0x0005;
    public static final short INC_DEC_NON_NUM_VAL = 0x0006;
```

```
}
public class Opcode {
    public static final byte GET = 0x00;
    public static final byte SET = 0x01;
    public static final byte DELETE = 0x04;
}
```

An `Opcode` tells Memcached which operations you wish to perform. Each operation is represented by a single byte. Similarly, when Memcached responds to a request, the response header contains two bytes, which represents the response status. The status and `Opcode` classes represent these Memcached constructs. Those `Opcodes` can be used when construct a new `MemcachedRequest` to specify which "action" should be triggered by it.

But let us concentrate on the encoder for now, which will encode the previous created `MemcachedRequest` class in a series of bytes. For this we extend the `MessageToByteEncoder`, which is a perfect fit for this use-case.

Listing 14.3 shows the implementation in detail.

Listing 14.3 – MemcachedRequestEncoder implementation

```
public class MemcachedRequestEncoder extends
        MessageToByteEncoder<MemcachedRequest> {                          #1
    @Override
    protected void encode(ChannelHandlerContext ctx, MemcachedRequest msg,
                          ByteBuf out) throws Exception {
        // convert key and body to bytes array                            #2
        byte[] key = msg.key().getBytes(CharsetUtil.UTF_8);
        byte[] body = msg.body().getBytes(CharsetUtil.UTF_8);
        // total size of body = key size + content size + extras size     #3
        int bodySize =
                key.length + body.length + (msg.hasExtras() ? 8 : 0);

        // write magic byte                                               #4
        out.writeByte(msg.magic());
        // write opcode byte                                              #5
        out.writeByte(msg.opCode());
        // write key length (2 byte) i.e a Java short                     #6
        out.writeShort(key.length);
        // write extras length (1 byte)                                   #7
           int extraSize =  msg.hasExtras() ? 0x08 : 0x0;
        out.writeByte(extraSize);
        // byte is the data type, not currently implemented in
        // Memcached but required                                         #8
        out.writeByte(0);
        // next two bytes are reserved, not currently implemented
        // but are required                                               #9
        out.writeShort(0);

        // write total body length ( 4 bytes - 32 bit int)               #10
        out.writeInt(bodySize);
        // write opaque ( 4 bytes)  -  a 32 bit int that is returned
        // in the response                                               #11
        out.writeInt(msg.id());

        // write CAS ( 8 bytes)
        // 24 byte header finishes with the CAS                          #12
```

```
        out.writeLong(msg.cas());

        if (msg.hasExtras()) {
            // write extras
            // (flags and expiry, 4 bytes each) - 8 bytes total         #13
            out.writeInt(msg.flags());
            out.writeInt(msg.expires());
        }
        //write key                                                     #14
        out.writeBytes(key);
        //write value                                                   #15
        out.writeBytes(body);
    }
}
```

#1 The class that is responsible to encode the MemachedRequest into a series of bytes
#2 Convert the key and the actual body of the request into byte arrays.
#3 Calculate body size
#4 Write the magic as byte to the ByteBuf
#5 Write the opCode as byte
#6 Write the key length as short
#7 Write the extra length as byte
#8 Write the data type, which is always 0 as it is currently not used in Memcached but may be used in later versions
#9 Write in a short for reserved bytes which may be used in later versions of Memcached
#10 Write the body size as a long
#11 Write the opaque as int
#12 Write the cas as long. This is the last field of the header, after this the body starts
#13 Write the extra flags and expire as int if there are some present
#14 Write the key
#15 Write the body. After this the request is complete.

In summary, our encoder takes a request and using the `ByteBuf` Netty supplies, converts the `MemcachedRequest` into a correctly sequenced set of bytes.

In detail that is:

- Write magic byte
- Write opcode byte
- Write key length (2 byte)
- Write extras length (1 byte)
- Write data type (1 byte)
- Write null bytes for reserved bytes (2 bytes)
- Write total body length (4 bytes - 32 bit int)
- Write opaque (4 bytes) - a 32 bit int that is returned in the response
- Write CAS (8 bytes)
- Write extras (flags and expiry, 4 bytes each) - 8 bytes total
- Write key
- Write value

Whatever you put into the output buffer (the `ByteBuf` called out) Netty will send to the server the request is being written to. The next section will show how this works in reverse via decoders.

14.4.2 Implementing the Memcached Decoder

The same way we need to convert a `MemcachedRequest` object to a series of bytes, Memcached will only return bytes so we need to convert those bytes back into a response object that we can use in our application. This is where decoders come in.

So again we first create a POJO, which is used to represent the response in an easy to use manner.

Listing 14.7 - Implementation of a MemcachedResponse

```
public class MemcachedResponse {                                        #1

    private byte magic;
    private byte opCode;
    private byte dataType;
    private short status;
    private int id;
    private long cas;
    private int flags;
       private int expires;
    private String key;
    private String data;

    public MemcachedResponse(byte magic, byte opCode,
            byte dataType, short status, int id, long cas,
            int flags, int expires, String key, String data) {
        this.magic = magic;
        this.opCode = opCode;
        this.dataType = dataType;
        this.status = status;
        this.id = id;
        this.cas = cas;
        this.flags = flags;
        this.expires = expires;
        this.key = key;
        this.data = data;
    }

    public byte magic() {                                               #2
        return magic;
    }

    public byte opCode() {                                               #3
        return opCode;
    }

    public byte dataType() {                                             #4
        return dataType;
    }

    public short status() {                                               #5
        return status;
```

```
    }

    public int id() {                                                    #6
        return id;
    }

    public long cas() {                                                  #7
        return cas;
    }

    public int flags() {                                                 #8
        return flags;
    }

    public int expires() {                                               #9
       return expires;
    }

    public String key() {                                               #10
       return key;
    }

    public String data() {                                              #11
       return data;
    }
}
```

#1 The class that represent a response which was send back from the Memcached server
#2 The magic number
ä3 The opCode which reflects for which operation the response was created
#4 The data type which indicate if its binary or text based
#5 The status of the response which indicate of the request was successful etc
#6 The unique id
#7 The compare –and-set value
#8 The extra flags used
#9 Indicate if the value stored for this response will be expire at some point
#10 The key for which the response was created
#11 The actual data

The previous created `MemcachedResponse` class will now be used in our decoder to represent the response send back from the Memcached server. As we want to decode a series of bytes into a `MemcachedResponse` we will make use of the `ByteToMessageDecoder` base class.

Listing 14.4 shows the `MemcachedResponseDecoder` in detail.

Listing 14.4 – MemcachedResponseDecoder class

```
public class MemcachedResponseDecoder extends ByteToMessageDecoder {      #1

    private enum State {                                                   #2
        Header,
        Body
    }

    private State state = State.Header;
    private int totalBodySize;
    private byte magic;
    private byte opCode;
```

```
private short keyLength;
private byte extraLength;
private byte dataType;
private short status;
private int id;
private long cas;

@Override
protected void decode(ChannelHandlerContext ctx, ByteBuf in,
                      List<Object> out) {
    switch (state) {                                                    #3
        case Header:
            if (in.readableBytes() < 24) {
                return;//response header is 24 bytes                    #4
            }
            // read header                                              #5
            magic = in.readByte();
            opCode = in.readByte();
            keyLength = in.readShort();
            extraLength = in.readByte();
            dataType = in.readByte();
            status = in.readShort();
            totalBodySize = in.readInt();
            //referred to in the protocol spec as opaque
            id = in.readInt();
            cas = in.readLong();

            state = State.Body;
            // fallthrough and start to read the body
        case Body:
            if (in.readableBytes() < totalBodySize) {
                return; //until we have the entire payload return    #6
            }
            int flags = 0, expires = 0;
            int actualBodySize = totalBodySize;
            if (extraLength > 0) {                                      #7
                flags = in.readInt();
                actualBodySize -= 4;
            }
            if (extraLength > 4) {                                      #8
                expires = in.readInt();
                actualBodySize -= 4;
            }
            String key = "";
            if (keyLength > 0) {                                        #9
                ByteBuf keyBytes = in.readBytes(keyLength);
                key = keyBytes.toString(CharsetUtil.UTF_8);
                actualBodySize -= keyLength;
            }
            ByteBuf body = in.readBytes(actualBodySize);               #10
            String data = body.toString(CharsetUtil.UTF_8);
            out.add(new MemcachedResponse(                             #11
                    magic,
                    opCode,
                    dataType,
                    status,
                    id,
                    cas,
                    flags,
                    expires,
```

```
                        key,
                        data
                ));

                state = State.Header;
            }

        }
    }
```

#1 The class that is responsible to create the MemcachedResponse out of the read bytes
#2 Represent current parsing state which means we either need to parse the header or body next
#3 Switch based on the parsing state
#4 If not at least 24 bytes are readable it's impossible to read the whole header, so return here and wait to get notified again once more data is ready to be read
#5 Read all fields out of the header
#6 Check if enough data is readable to read the complete response body. The length was read out of the header before
#7 Check if there are any extra flags to read and if so do it
#8 Check if the response contains an expire field and if so read it
#9 check if the response contains a key and if so read it
#10 Read the actual body payload
#11 Construct a new MemachedResponse out of the previous read fields and data

So what happened in the implementation?

We know that a Memcached response has a 24 byte header, what we don't know is whether all the data that makes up a response will be included in the input `ByteBuf` when the decode method is called. This is because the underlying network stack may break the data into chunks. So to ensure we only decode when we have enough data, the above code checks if the amount of readable bytes available is at least 24 bytes. Once we have the first 24 bytes we can determine how big the entire message is because this information is contained within the 24 bytes header.

When we've decoded an entire message, we create a `MemcachedResponse` and add it to the output list. Any object added to this list will be forwarded to the next `ChannelInboundHandler` in the `ChannelPipeline` and so allows processing it.

14.5 Testing the codec

With the previous created encoder and decoder we have our codec in place. But there is still something missing; Tests…

Without tests you will only see if it works when running it against some real server, which is not what you should depend on. As seen in chapter 10 writing tests for custom `ChannelHandler` is usually done via `EmbeddedChannel`.

So this is exactly what we do now to test our custom codec, which includes an encoder and decoder.

Let's us start with the encoder again. Listing 14.5 shows the simple written unit test.

Listing 14.5 – MemcachedRequestEncoderTest class

```
public class MemcachedRequestEncoderTest {

    @Test
    public void testMemcachedRequestEncoder() {
        MemcachedRequest request =
                new MemcachedRequest(Opcode.SET, "key1", "value1");      #1

        EmbeddedChannel channel = new EmbeddedChannel(
                new MemcachedRequestEncoder());                          #2
        Assert.assertTrue(channel.writeOutbound(request));               #3

        ByteBuf encoded = (ByteBuf) channel.readOutbound();

        Assert.assertNotNull(encoded);                                   #4
        Assert.assertEquals(request.magic(), encoded.readByte());        #5
        Assert.assertEquals(request.opCode(), encoded.readByte());       #6
        Assert.assertEquals(4, encoded.readShort());                     #7
        Assert.assertEquals((byte) 0x08, encoded.readByte());            #8
        Assert.assertEquals((byte) 0, encoded.readByte());               #9
        Assert.assertEquals(0, encoded.readShort());                    #10
        Assert.assertEquals(4 + 6 + 8, encoded.readInt());              #11
        Assert.assertEquals(request.id(), encoded.readInt());           #12
        Assert.assertEquals(request.cas(), encoded.readLong());         #13
        Assert.assertEquals(request.flags(), encoded.readInt());        #14
        Assert.assertEquals(request.expires(), encoded.readInt());      #15

        byte[] data = new byte[encoded.readableBytes()];                #16
        encoded.readBytes(data);
        Assert.assertArrayEquals((request.key() + request.body())
                .getBytes(CharsetUtil.UTF_8), data);
        Assert.assertFalse(encoded.isReadable());                       #17

        Assert.assertFalse(channel.finish());
        Assert.assertNull(channel.readInbound());
    }
}
```

#1 Create a new MemcachedRequest which should be encoded to a ByteBuf
#2 Create a new EmbeddedChannel which holds the MemcachedRequestEncoder to test
#3 Write the request to the channel and assert if it produced an encoded message
#4 Check if the ByteBuf is null
#5 Assert that the magic was correctly written into the ByteBuf
#6 Assert that the opCode (SET) was written correctly
#7 Check for the correct written length of the key
#8 Check if this request had extras included and so they was written
#9 Check if the data type was written
#10 Check if the reserved data was insert
#11 Check fo the total body size which is key.length + body.length + extras
#12 Check for correctly written id
#13 Check for correctly written Compare and Swap (CAS)
#14 Check for correct flags
#15 Check if expire was set
#16 Check if key and body are correct.

After tests for the encoder are in place, it's time to take care of the tests for the decoder. As we not know if the bytes will come in all in once or fragmented we need take special care to test both cases.

Listing 14.6 shows the tests in detail.

Listing 14.6 – MemcachedRequestEncoderTest class

```
public class MemcachedResponseDecoderTest {

    @Test
    public void testMemcachedResponseDecoder() {
        EmbeddedChannel channel = new EmbeddedChannel(
                new MemcachedResponseDecoder());                          #1

        byte magic = 1;
        byte opCode = Opcode.SET;
        byte dataType = 0;

        byte[] key = "Key1".getBytes(CharsetUtil.US_ASCII);
        byte[] body = "Value".getBytes(CharsetUtil.US_ASCII);
        int id = (int) System.currentTimeMillis();
        long cas = System.currentTimeMillis();

        ByteBuf buffer = Unpooled.buffer();                               #2
        buffer.writeByte(magic);
        buffer.writeByte(opCode);
        buffer.writeShort(key.length);
        buffer.writeByte(0);
        buffer.writeByte(dataType);
        buffer.writeShort(Status.KEY_EXISTS);
        buffer.writeInt(body.length + key.length);
        buffer.writeInt(id);
        buffer.writeLong(cas);
        buffer.writeBytes(key);
        buffer.writeBytes(body);

        Assert.assertTrue(channel.writeInbound(buffer));                  #3

        MemcachedResponse response = (MemcachedResponse) channel.readInbound();
        assertResponse(response, magic, opCode, dataType,
                Status.KEY_EXISTS, 0, 0, id, cas, key, body);             #4
    }

    @Test
    public void testMemcachedResponseDecoderFragments() {
        EmbeddedChannel channel = new EmbeddedChannel(
                new MemcachedResponseDecoder());                          #5

        byte magic = 1;
        byte opCode = Opcode.SET;
        byte dataType = 0;

        byte[] key = "Key1".getBytes(CharsetUtil.US_ASCII);
        byte[] body = "Value".getBytes(CharsetUtil.US_ASCII);
        int id = (int) System.currentTimeMillis();
        long cas = System.currentTimeMillis();

        ByteBuf buffer = Unpooled.buffer();                               #6
```

```
        buffer.writeByte(magic);
        buffer.writeByte(opCode);
        buffer.writeShort(key.length);
        buffer.writeByte(0);
        buffer.writeByte(dataType);
        buffer.writeShort(Status.KEY_EXISTS);
        buffer.writeInt(body.length + key.length);
        buffer.writeInt(id);
        buffer.writeLong(cas);
        buffer.writeBytes(key);
        buffer.writeBytes(body);

        ByteBuf fragment1 = buffer.readBytes(8);                        #7
        ByteBuf fragment2 = buffer.readBytes(24);
        ByteBuf fragment3 = buffer;

        Assert.assertFalse(channel.writeInbound(fragment1));             #8
        Assert.assertFalse(channel.writeInbound(fragment2));             #9
        Assert.assertTrue(channel.writeInbound(fragment3));             #10

        MemcachedResponse response = (MemcachedResponse) channel.readInbound();
        assertResponse(response, magic, opCode, dataType,
                Status.KEY_EXISTS, 0, 0, id, cas, key, body);           #11
    }

    private static void assertResponse(MemcachedResponse response, byte magic, byte
      opCode, byte dataType, short status, int expires, int flags, int id, long cas,
      byte[] key, byte[] body) {
        Assert.assertEquals(magic, response.magic());
        Assert.assertArrayEquals(key, response.key().getBytes(CharsetUtil.US_ASCII));
        Assert.assertEquals(opCode, response.opCode());
        Assert.assertEquals(dataType, response.dataType());
        Assert.assertEquals(status, response.status());
        Assert.assertEquals(cas, response.cas());
        Assert.assertEquals(expires, response.expires());
        Assert.assertEquals(flags, response.flags());
        Assert.assertArrayEquals(body,
      response.data().getBytes(CharsetUtil.US_ASCII));
        Assert.assertEquals(id, response.id());
    }
}
```

#1 Create a new EmbeddedChannel which holds the MemcachedResponseDecoder to test
#2 Create a new Buffer and write data to it which match the structure of the binary protocol
#3 Write the buffer to the EmbeddedChannel and check if a new MemcachedResponse was created by assert the return value
#4 Assert the MemcachedResponse with the expected values
#5 Create a new EmbeddedChannel which holds the MemcachedResponseDecoder to test
#6 Create a new Buffer and write data to it which match the structure of the binary protocol
#7 Split the Buffer into 3 fragments
#8 Write the first fragment to the EmbeddedChannel and check that no new MemcachedResponse was created, as not all data is ready yet
#9 Write the second fragment to the EmbeddedChannel and check that no new MemcachedResponse was created, as not all data is ready yet
#10 Write the last fragment to the EmbeddedChannel and check that a new MemcachedResponse was created as we finally received all data.
#11 Assert the MemcachedResponse with the expected values

It's important to note that the shown tests don't provide full coverage but mainly "act" as example how you would make use of the `EmbeddedChannel` to test your custom written codec. How "complex" your tests need to be mainly depends on the implementation itself and so it's impossible to test what exactly you should test.

Anyway here are some things you should generally take care of:

- Testing with fragmented and non fragmented data
- Test validation if received data / send data if needed

14.6 Summary

After reading this chapter you should be able to create your own codec for your "favorite" protocol. This includes writing the encoder and decoder, which convert from bytes to your POJO and vise-versa. It shows how you can use a protocol specification and extract the needed information for the implementation.

Beside this it shows you how you complete your work by write unit tests for the encoder and decoder and so make sure everything works as expected without the need to run a full Memcached Server. This allows easy integration of tests into the build-system.

In the next chapter we will have a deeper look into the thread model Netty uses and how it is abstracted away. Understanding the thread model itself will allow you to make the best use of Netty and the features it provides.

15

EventLoop and Thread-Model

This chapter covers

- Details about the thread-model
- EventLoop
- Concurrency
- Task execution
- Task scheduling

A thread model defines how the application or framework executes your code, so it's important to choose the right thread model for the application/framework. Netty comes with an easy but powerful thread model that helps you simplify your code. All of your `ChannelHandlers`, which contain your business logic, are guaranteed to be executed by the same `Thread` for a specific `Channel`. This doesn't mean Netty fails to use multithreading, but it does pin each `Channel` to one `Thread`. This design works very well for non-blocking IO-Operations.

After reading this chapter you'll have a deep understanding of Netty's thread model and why the Netty team chose it over other models. With this information you'll be able to get the best performance out of an application that uses Netty under the covers. You will learn from the Netty team's experience; why the current model was adopted in favor of the previous one and how it makes Netty even easier to use and more powerful

This chapter assumes the following:

- You understand what threads are used for and have experience working with them.

  If that's not the case, please take time to gain this knowledge to make sure everything is clear to you. A good reference is *Java Concurrency in Practice* by Brian Goetz.

- You understand multi-threaded applications and their designs.

  This also includes what it takes to keep them thread-safe while trying to get the best performance out of them.

While we make these assumptions, if you're new to multi-threaded applications, having a general understanding of the core concepts should be enough for this chapter to provide some enlightening insights for you.

15.1 Thread model overview

In this section you'll get a brief introduction to thread models in general, how Netty uses the specific thread model now, and what thread models Netty used in the past. You'll be able to understand all the pros and cons of the different thread models.

A thread-model specifies how code is executed and gives the developer information on how his code will be executed. This is important as it allows the developer to know in advanced how he needs to guard his code from side effects by concurrent execution. Without this

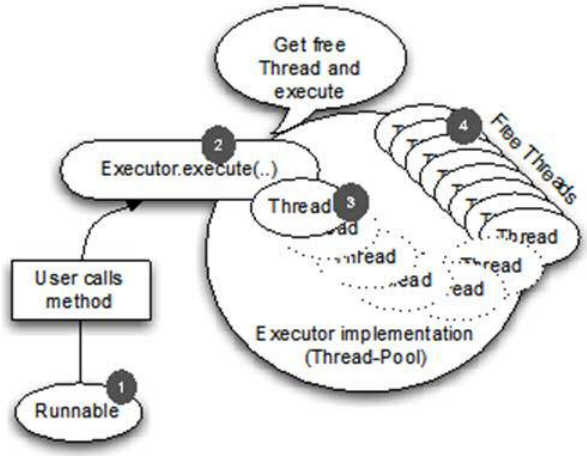
knowledge the best a developer could do is to place a bet and hope to get lucky in the end, but this will bite back in almost all cases.

Before going into more detail, let's get a better understanding of the topic by revisiting what most applications do these days.

Most modern applications use more than one thread to dispatch work and therefore let the application use all of the system's available resources in an efficient manner. This makes a lot of sense as most hardware comes with more then one core or even more then one CPU these days. If everything would only be executed in one `Thread` it would be impossible to make proper use of the provided resource. To fix this many applications execute the running code in multiple `Threads`. In the early days of Java, this was done by simply creating new `Threads` on demand and starting them when work was required to be done in parallel.

But it soon turned out that this isn't perfect, because creating `Threads` and recycling them comes with overhead. In Java 5 we finally got so called thread pools, which often caches Threads, and so eliminate the overhead of creation and recycling. These pools are provided by the `Executor` interface. Java 5 provided many useful implementations, which vary significantly in their internals, but the idea is the same on all of them. They create `Threads` and reuse them when a task is submitted for execution. This helps keep the overhead of creating and recycling threads to a minimum.

Figure 15.1 shows the how a thread pool is used to execute a task. It submits a task that will be executed by one free thread and once it's complete it frees up the thread again.



#1 Runnable which represents the task to execute. This could be anything from a database call to file system cleanup
#2 Previous runnable gets hand-over to the thread pool
#3 A free Thread is picked and the task is executed with it. When a thread has completed running it is put back into the list of free threads to be reused when a new task needs to be run
#4 Threads that execute tasks

Figure 15.1 Executor execution logic

This fixes the overhead of `Thread` creation and recycling by not requiring new `Threads` to be created and destroyed with each new task.

But using multiple `Threads` comes with the cost of managing resources and, as a side effect, introduces too much context switching. This gets worse with the number of threads that run and the number of tasks to execute increases. While using multiple threads may not seem like a problem at the beginning, it can hit you hard once you put real workload on the system.

In addition to these technical limitations and problems, other problems can occur that are more related to maintaining your application/framework in the future or during the lifetime of the project. It's valid to say that the complexity of your application rises depending on how parallel it is. To state it simply: Writing a multithreaded application is a hard job! What can we do to solve this problem? You need multiple `Threads` to scale for real-world scenarios; that's a fact. Let's see how Netty solves this problem.

15.2 The EventLoop

The event loop does exactly what its name says. It runs events in a loop until it's terminated. This fits well in the design of network frameworks, as they need to run events for a specific connection in a loop when these occur. This isn't something new that Netty invented; other frameworks and implementations have been doing this for ages.

Listing 14.1 shows the typical logic of an `EventLoop` that behaves the same way as explained. Be aware this is more to illustrate the idea then to show the actual implementation of Netty itself.

Listing 14.1 Execute task in `EventLoop`

```
while (!terminated) {
    List<Runnable> readyEvents = blockUntilEventsReady();        #1
    for (Runnable ev: readyEvents) {
          ev.run();                                               #2
    }
}
```

#1 Block until we have events that are ready to run
#2 Loop over all of the events and run them

The event-loop itself is represented by the `EventLoop` interface in Netty. As an `EventLoop` extends from `EventExecutor`, which in turn extends from `ScheduledExecutorService`, it's possible to also directly hand over a task that will be executed by the `EventLoop`. Beside this, it is also possible to schedule tasks for later execution, just like you can do this with any `ScheduledExecutorService` implementation.

The class/interface hierarchy of `EventExecutor` is shown in figure 15.2 (only shown in one depth).

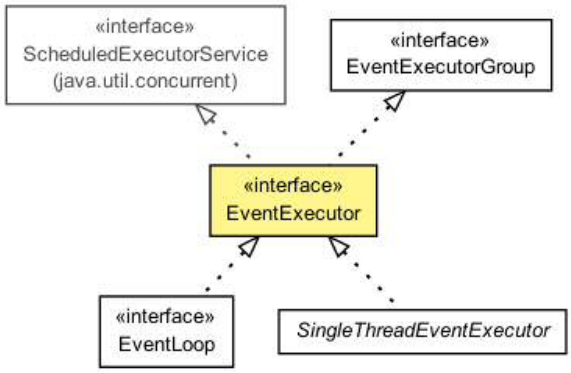


Figure 15.2 EventLoop class hierarchy

An `EventLoop` is powered by exactly one `Thread` that never changes. To make proper use of resources Netty uses multiple `EventLoop's`, depending on the configuration and the available cores.

Event / Task execution order

One important detail about the execution order of events and tasks is that the events / tasks are executed in a `FIFO` (First-in-First-Out) order. This is needed as otherwise events could be processed out of order and so the bytes that are processed would not be guaranteed to be in the right order. This would lead to corruption and so is not allowed by design.

15.2.1 I/O and Event handling in Netty 4

Netty uses I/O events, which are triggered by various IO operations on the transport itself. These I/O operations are i.e. part of the network API that's provided by Java and the underlying operating system.

One difference here is that some operations (and so events) are triggered by the transport implementation of Netty itself and some by the user. For example a „read event" is typically triggered by the transport itself once there was some data read. In contrast a write event is most of the times triggered by the user itself, i.e. when calling `Channel.write(...)`.

What exactly needs to be done once an event is handled depends on the nature of the event. Often it will read or transfer data from the network stack into your application. Other times it will do the same in the other direction, for example, transferring data from your application into the network stack (kernel) to send it over the remote peer. But it's not limited to this type of transaction; the important thing is that the logic used is generic and flexible enough to handle all kinds of use cases.

One important thing about I/O and Event handling in Netty 4 is that each of these I/O operations and events are always handled by the EventLoop itself and so by the Thread that is assigned to the EventLoop.

It should be noted that the thread model (abstracted via the `EventLoop`) described was not always used by Netty. In the next section you'll learn about the thread model used in Netty 3. This will help you to understand why the new thread model is now used and so replaced the old model, which is still in use by Netty 3.

15.2.2 I/O operations in Netty 3

In previous releases the thread model was different. Netty guaranteed only that inbound (previously called upstream) events were executed in the I/O `Thread` (the I/O `Thread` was what now is called `EventLoop` in Netty 4). All outbound (previously called downstream) events were handled by the calling `Thread`, which may be the I/O `Thread` but also any other `Thread`. This sounded like a good idea at first but turned out to be error-prone as handling of outbound events in the `ChannelHandlers` needed carefully synchronization, as it wasn't guaranteed that only one `Thread` operated on them at the same time. This could happen if you fired downstream events at the same time for one `Channel`, for example, call `Channel.write(..)` in different threads.

In addition to the burden placed on you to synchronize `ChannelHandlers`, another problematic side effect of this thread model is that you may need to fire an inbound (upstream) event as a result of an outbound (downstream) event. This is true, for example, if your `Channel.write(..)` operation causes an exception. In that case, an `exceptionCaught` must be generated and fired. This doesn't sound like a problem at first glance, but knowing that `exceptionCaught` is an inbound (upstream) event by design may give you an idea where the problem is. The problem is, in fact, that you now have a situation where the code is executed in your calling `Thread` but the `exceptionCaught` event must be handed over to the worker thread for execution and so a context switch is needed again.

In contrast the new thread model of Netty 4 not has the problems stated above, as everything is executed in the same `EventLoop` and so in the same `Thread`. This eliminates all the need of synchronization in the `ChannelHandler` and makes it much easier for the user to reason about the execution.

Now that you know how you can execute tasks in the `EventLoop`, it's time to have a quick look at various Netty internals that use this feature.

15.2.3 Netty's thread model internals

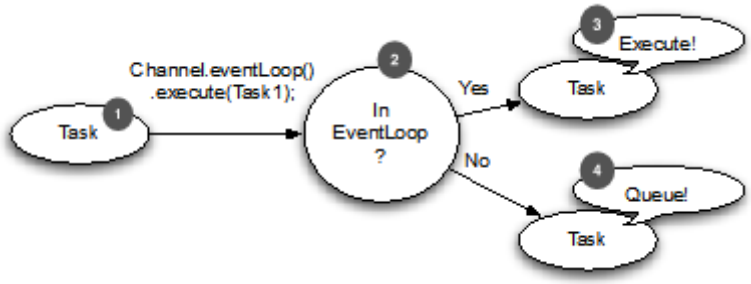
The trick that's used inside Netty to make its thread model perform so well is that it checks to see if the executing `Thread` is the one that's assigned to the actual `Channel` (and `EventLoop`). The `EventLoop` is responsible for handling all events for a `Channel` during its lifetime.

If the calling `Thread` is the same as the one of the `EventLoop`, the code block in question is executed. If the `Thread` is different, it schedules a task and puts it in an internal queue, used by the `EventLoop`, for later execution. The `EventLoop` will notice that there is a queued

task to process and automatically pick it up. This allows you to directly interact with the `Channel` from any `Thread`, while still being sure that all `ChannelHandlers` don't need to worry about concurrent access.

It is important to know that each `EventLoop` has it's own task/event queue and so is not affected by other `EventLoops` when processing the queue.

Figure 15.5 shows the execution logic that's used in the `EventLoop` to schedule tasks to fit Netty's thread model



#1 The task that should be executed in the EventLoop
#2 After the task is passed to the execute methods, a check is performed to detect if the calling thread is the same as the one that's assigned to the EventLoop
#3 The thread is the same, so you're in the EventLoop, which means the task can get executed directly
#4The thread is not the same as the one that's assigned to the EventLoop. Queue the task to get executed once the EventLoop processes its events again

Figure 15.5 EventLoop execution logic / flow

That said, because of the design it's important to ensure that you never put any long-running tasks in the execution queue, because once the task is executed and run it will effectively block any other task from executing on the same thread. How much this affects the overall system depends on the implementation of the `EventLoop` that's used in the specific transport implementation.

As switching between transports is possible without any changes in your code base, it's important to remember that the Golden Rule always applies here: Never block the I/O thread. If you must do blocking calls (or execute tasks that can take long periods to complete), use a dedicated `EventExecutor` for your `ChannelHandler` as explained in Chapter 6.

The next section will show one more feature that's often needed in applications. This is the need to schedule tasks for later execution or for periodic execution. Java has out-of-the-box solutions for this job, but Netty provides you with several advanced implementations that you'll learn about next

15.3 Scheduling tasks for later execution

Every once in a while, you need to schedule a task for later execution. Maybe you want to register a task that gets fired after a client is connected for five minutes. A common use case

is to send an "Are you alive?" message to the remote peer to see if it's still there. If it fails to respond, you know it's not connected anymore, and you can close the channel (connection) and release the resources.

The next section will show how you can schedule tasks for later execution in Netty using its powerful `EventLoop` implementation. It also gives you a quick introduction to task scheduling with the core Java API to make it easier for you to compare the built API with the one that comes with Netty. In addition to these items, you'll get more details about the internals of Netty's implementation and understand what advantages and what limitations it provides.

15.3.1 Scheduling tasks with plain Java API

To schedule a task you typically use the provided `ScheduledExecutorService` implementations that ship with the JDK. This wasn't always true. Before Java 5 you used a timer, which has the exact same limitations as normal threads.

Table 15.1 shows the utility methods that you can use to create an instance of `ScheduledExecutorService`.

Table 15.1 java.util.concurrent.Executors—Static methods to create a ScheduledExecutorService

Methods	Description
`newScheduledThreadPool(int corePoolSize)`	Creates a new `ScheduledThreadExecutorService` that can schedule commands to run after a delay or to execute periodically. It will use `corePoolSize` to calculate the number of threads.
`newScheduledThreadPool(int corePoolSize, ThreadFactorythreadFactory)`	
`newSingleThreadScheduledExecutor()`	Creates a new `ScheduledThreadExecutorService` that can schedule commands to run after a delay or to execute periodically. It will use one thread to execute the scheduled tasks.
`newSingleThreadScheduledExecutor (ThreadFactorythreadFactory)`	

After looking at the table 15.1 you'll notice that not many choices exist, but these are enough for most use cases. Now see how `ScheduledExecutorService` is used in listing 15.4 to schedule a task to run after a 60-second delay.

Listing 15.4 Schedule task with a `ScheduledExecutorService`

```
ScheduledExecutorService executor = Executors
.newScheduledThreadPool(10);                                    #1

ScheduledFuture<?> future = executor.schedule(
new Runnable() {                                                #2
    @Override
public void run() {
```

```
System.out.println("Now it is 60 seconds later");            #3
    }
}, 60, TimeUnit.SECONDS);                                     #4
...
...
executor.shutdown();                                          #5
```

#1 Create a new ScheduledExecutorService which uses 10 threads
#2 Create a new runnable to schedule for later execution
#3 This will run later
#4 Schedule task to run 60 seconds from now
#5 Shut down ScheduledExecutorService to release resources once task complete

As you can see, using `ScheduledExecutorService` is straightforward.

Now that you know how to use the classes of the JDK, you'll learn how you can use Netty's API to do the same, but in a more efficient way.

15.3.2 Scheduling tasks using EventLoop

Using the provided `ScheduledExecutorService` implementation may have worked well for you in the past, but it comes with limitations, such as the fact that tasks will be executed in an extra thread. This boils down to heavy resource usage if you schedule many tasks in an aggressive manner. This heavy resource usage isn't acceptable for a high-performance networking framework like Netty. What do you do if you must schedule tasks for later execution but still need to scale? Fortunately everything you need is already provided for free and part of the core API.

Netty solves this by allowing you to schedule tasks using the `EventLoop` that's assigned to the channel, as shown in listing 15.5.

Listing 15.5 Schedule task with `EventLoop`

```
Channel ch = ...
ScheduledFuture<?> future = ch.eventLoop().schedule(
new Runnable() {                                              #1
    @Override
public void run() {
System.out.println("Now its 60 seconds later");              #2
}
}, 60, TimeUnit.SECONDS);                                     #3
```

#1 Create a new runnable to schedule for later execution
#2 This will run later
#3 Schedule task to run 60 seconds from now

After the 60 seconds passes, it will get executed by the `EventLoop` that's assigned to the channel.

As you learned earlier in the chapter, `EventLoop` extends `ScheduledExecutorService` and provides you with all the same methods that you learned to love in the past when using executors.

You can do other things like schedule a task that gets executed every X seconds. To schedule a task that will run every 60 seconds, use the code shown in listing 15.6.

Listing 15.6 Schedule a fixed task with the `EventLoop`

```
Channel ch = ...
ScheduledFuture<?> future = ch.eventLoop()
.scheduleAtFixedRate(new Runnable() {                    #1
    @Override
public void run() {
System.out.println("Run every 60 seconds");              #2
        }
}, 60, 60, TimeUnit.Seconds);                            #3
```

#1 Create new runnable to schedule for later execution
#2 This will run until the ScheduledFuture is canceled
#3 Schedule in 60 seconds and every 60 seconds

To cancel the execution, use the returned `ScheduledFuture` that's returned for every asynchronous operation. The `ScheduledFuture` provides a method for canceling a scheduled task or to check the state of it.

One simple cancel operation would look like the following example.

```
ScheduledFuture<?> future = ch.eventLoop()
.scheduleAtFixedRate(..);                                #1
// Some other code that runs...
future.cancel(false);                                    #2
```

#1 Schedule task and obtain the returned ScheduledFuture
#2 Cancel the task, which prevents it from running again

The complete list of all the operations can be found in the `ScheduledExecutorService` javadocs.

Now that you know what you can do with it, you may start to wonder how the implementation of the `ScheduledExecutorService` is different in Netty, and why it scales so well compared to other implementations of it.

15.3.3 Scheduling implementation internals

The actual implementation in Netty is based on the paper "Hashed and hierarchical timing wheels: Data structures to efficiently implement timer facility" by George Varghese. This kind of implementation only guarantees an approximated execution, which means that the execution of the task may not be 100% exactly on time. This has proven to be a tolerable limitation in practice and does not affect most applications at all. It's just something to remember if you need to schedule tasks, because it may not be the perfect fit if you need 100% exact, on-time execution.

To better understand how this works, think of it this way:

1. You schedule a task with a given delay.

2. The task gets inserted into the Schedule-Task-Queue of the `EventLoop`.
3. The `EventLoop` checks on every run if tasks need to get executed now.
4. If there's a task, it will execute it right away and remove it from the queue.
5. It waits again for the next run and starts over again with step 4.

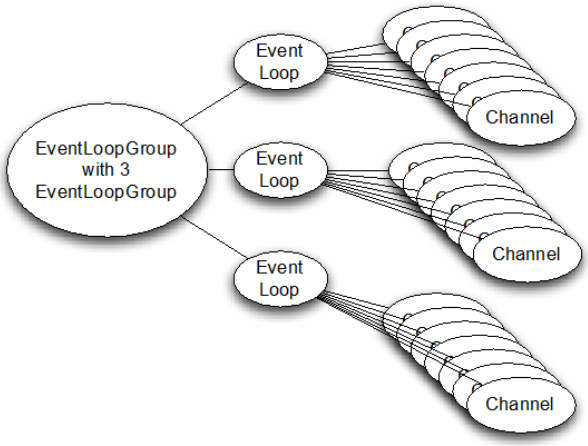
Because of this implementation the scheduled execution may be not 100 % acurate. Which means it may not be 100 % exact, this is fine for most use-cases given that it allows for almost no overhead within Netty.

But what if you need more accurate execution? It's easy. You'll need to use another implementation of `ScheduledExecutorService` that's not part of Netty. Just remember that if you don't follow Netty's thread model protocol, you'll need to synchronize the concurrent access on your own. Do this only if you must.

15.4 I/O EventLoop / Thread allocation in detail

Netty uses an EventLoopGroup that contains EventLoop's that will serve the I/O and events for a Channel. The way EventLoops are created / assigned varies upon the transport implementation. An asynchronous implementation uses only a few EventLoops (and so Threads) that are shared between the Channels. This allows a minimal number of Threads to serve many Channels, eliminating the need to have one dedicated Thread for each of them.

Figure 15.7 shows how the `EventLoopGroup` is used



#1 All EventLoops get allocated out of this EventLoopGroup. Here it will use three EventLoops instances
#2 This EventLoop handles all events and tasks for all the channels assigned to it. Each EventLoop is tied to one Thread
#3 Channels that are bound to the EventLoop and so all operations are always executed by the same thread during the life-time of the Channel. A Channel belongs to a connection

Figure 15.7 Thread allocation for nonblocking transports (such as NIO and AIO)

As you can see, figure 15.7 uses an `EventLoopGroup` with a fixed size of three `EventLoops` (each powered by one Thread). The `EventLoops` (and so the Threads) are allocated directly once the `EventLoopGroup` is created. This is done to make sure resources are available when needed.

These three `EventLoop` instances will get assigned to each newly created `Channel`. This is done through the `EventLoopGroup` implementation, which manage the `EventLoop` instances.. The actual implementation will take care of distributing the created `Channels` evenly on all `EventLoop` instances (and so on different `Threads`). This distribution is done in a round-robin fashion by default but may change in further releases of Netty.

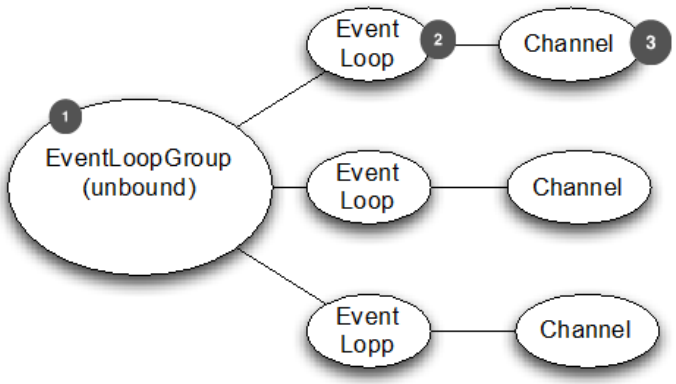
Once a `Channel` is assigned to an `EventLoop` it will use this `EventLoop` throughout its lifetime and so the `Thread` that powers the `EventLoop`. You can, and should, depend on this, as it makes sure that you don't need to worry at all about synchronization (which includes thread-safety, visibility and synchronization) in your `ChannelHandler` implementations.

But this also affects the usage of i.e `ThreadLocal`, which is often used by Applications. Because an `EventLoop` usually powers more then one Channel the `ThreadLocal` will be the same for all of the Channel that are assigned to the `EventLoop`. So it is a bad fit for state tracking etc. What it can be still very useful for is to share "heavy/expensive objects" between Channels that not need to keep any state at all and so can be used for each event without the need to depend on the previous state of the `ThreadLocal`.

EventLoop and Channel

We should note that it is possible in Netty 4 to deregister a Channel from its `EventLoop` and register it with a different `EventLoop` later. This functionality was @deprecated now as it did not work out very well in practice.

The semantic is a bit different for other transports, such as the shipped OIO (Old Blocking I/O) transport, as you can see in figure 14.8.



#1 All EventLoops get allocated out of this EventLoopGroup. Each new channel will get a new EventLoop
#2 EventLoop allocated for the channel will execute all events and tasks
#3 Channel bound to the EventLoop. A channel belongs to a connection

Figure 15.8 Thread allocation of blocking transports (such as OIO)

As you may notice here, one `EventLoop` (and so `Thread`) is created per `Channel`. You may be used to this from developing a network application that's based on regular blocking I/O when using classes out of the java.io.* package. But even if the semantics change in this case, one thing still stays the same. Each `Channel's` I/O will only be handled by one `Thread` at one time, which is the one `Thread` that powers the `EventLoop` of the `Channel`. You can depend on this hard rule; it's what makes writing code via the Netty framework so easy compared to other user network frameworks out there.

15.5 Limitations of the Thread-Model

At the current date there are two limitations which will be addressed in Netty 5, that are a result of the current thread model(Netty 4). This section will explain these limitations and how the Team plans to fix them as part of Netty 5.

15.5.1 Choosing the next EventLoop

In Netty 4 a round-robin like algorithm is used to choose the next EventLoop for a `Channel` that was created. This works well on the start but after a while it may happen that some `EventLoop's` have less `Channels` assigned then others. This is even more likely if the `EventLoopGroup` (that contains the `EventLoops`) handle different protocols or different kind of connections like long-living and short-living connections.

A fix may sound easy at first glance, as you could just choose the `EventLoop` that has the smallest `Channel` count. But taking into account the number of `Channels` on the `EventLoop` is not enough for a proper fix, as even if one `EventLoop` holds more `Channels` then another does

not mean automatically that this `EventLoop` is busier. Here many things need to be considered. This includes:

- Number of queued tasks
- Last execution times
- How saturated the network stack is

The problem is tracked as part of issue 1230[36] and currently targeted for Netty 5.0.0.Final but may be backported to Netty 4 if it can be implemented without any API breakage.

15.5.2 Re-distribute Channels across the EventLoop

The other problem is how to re-distribute `Channels` across the `EventLoops` and so better make use of the resources. This is especially important for long-living connections, as chances are "better" here that some `EventLoops` will get more busy then others over the time.

To fix this the Netty Team is currently working on introduce a `reregister(....)` method which allows to move a Channel to an other `EventLoop` and so move over it's handling to another Thread. But this is not as easy as it sounds, as it brings a few problems with them. First of all it must to be ensured that all events / operations that were triggered while the „old" `EventLoop` was registered are executed while still be on the „old" `EventLoop`. Failing to do so would break the thread model, which guarantees that each `ChannelHandler` is only processed by one `Thread`. Another issue is how to make sure that we not encounter any visibility issues when move from one `Thread` to another.

This problem is currently tracked as issue 1799[37] and targeted for Netty 5.0.0.Final. It is not likely that it will be ported to Netty 4.x as it will need a small API breakage to expose the needed operations on the `Channel` interface.

15.6 Summary

In this chapter you learned which thread model Netty uses. You learned the pros and cons of using thread models and how they simplify your life when working with Netty.

In addition to the inner workings, you also gained insight on how you can execute your own tasks in the `EventLoop` (I/O `Thread`) the same way that Netty does. You learned how to schedule tasks for later execution and how this feature is implemented to scale even if you schedule a large number of tasks. You also learned how to verify whether a task has executed and how to cancel it.

You now know what thread model previous versions of Netty used, and you got more background information about why the new thread model is more powerful than the old one it replaced.

[36] https://github.com/netty/netty/issues/1230
[37] https://github.com/netty/netty/issues/1799

All of this should give you a deeper understanding of Netty's thread model, thereby helping you maximize your application's performance while minimizing the code needed. For more information about thread pools and concurrent access, please refer to the book *Java Concurrency in Practice* by Brian Goetz. His book will give you a deeper understanding of even the most complex applications that have to handle multithreaded use case scenarios.

16

Case Studies, Part 1: Droplr, Firebase, and Urban Airship

In this chapter we'll take a look at

- `Case-Studies`
- `Real-World Use-Cases`

In previous Chapters you learned a lot about the details of Netty and how you can make use of them. This is great but often it is quite helpful once you see how others have empower-ed their applications to make use of Netty.

This chapter is all about this. In this Chapter we cover 3 different Companies that make heavy use for their internal infrastructure. You will get an insight in how Netty was used to solve those problems and may be able to make use of it to solve your own.

The Companies are:

- Droplr
- Firebase
- Urban Airship

Every Case-Study was written by the people that were part of the effort to use Netty within those companies. The Case-Studies are ordered by the name of the Companies and so the order does not reflect any relevance here.

16.1 Droplr - Building Mobile Services

Written by Bruno de Carvalho; Lead Architect @ Droplr

At Droplr we use Netty at the heart of our infrastructure, in everything ranging from our API servers to auxiliary services.

This is a case study on how we moved from a monolithic and sluggish LAMP application to a modern high-performance and horizontally distributed infrastructure, implemented atop of Netty.

16.1.1 How it all started

When I joined the team a couple of years ago we were running a LAMP application that served both as the front-end for users and as an API for the client applications — among which, my reverse engineered third-party Windows client, *windroplr*.

Windroplr went on to become *Droplr for Windows* and I, being mostly an infrastructure guy, eventually got a new challenge: completely rethink Droplr's infrastructure.

By then Droplr had already established itself as a working concept so the goals were pretty standard for a 2.0 version:

- Break the monolithic stack into multiple horizontally scalable components;
- Add redundancy to avoid downtime;
- Create a clean API for clients;
- Make it all run on HTTPS.

Josh and Levi, the founders, asked me to "make it fast, whatever it takes".

I knew those words meant more than just making it *slightly faster* or even *a lot faster*. "Whatever it takes" meant *a-full-order-of-magnitude faster*. And I knew then that Netty would eventually play an important role in this endeavor.

16.1.2 How Droplr works

Droplr has an extremely simple workflow; drag a file to the app's menu bar icon, Droplr uploads the file and when the upload completes Droplr copies a short URL to the file — the *drop* — to the clipboard.

That's just it. Frictionless, instant sharing.

Behind the scenes, *drop* metadata is stored in a database — creation date, name, number of downloads, etc. — and the files on Amazon S3.

16.1.3 Creating a faster upload expirience

The upload flow for Droplr's first version was woefully naive:

1. Receive upload
2. Upload to S3
3. Create thumbnails if it's an image
4. Reply to client applications

A closer look at this flow quickly reveals two choke points on steps 2 and 3. No matter how fast the upload from the client to our servers was, the creation of a *drop* would always go through an annoying hiatus after the actual upload completed, until the successful response was received — because the file would still need to be uploaded to S3 and have its thumbs generated.

The larger the file, the longer the hiatus. For very large files the connection would eventually timeout just waiting for the OK from the server — back then Droplr could only offer uploads of up to 32MB per file because of this very problem.

There were two distinct approaches to cut down upload times:

- Option A, the optimistic-and-apparently-simpler approach:
- Fully receive the file;
- Save to local filesystem and immediately return success to client;
- Schedule an upload to S3 some time in the future.

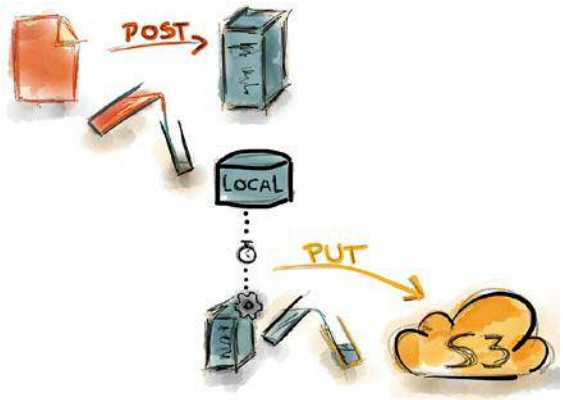


Figure 16.1 Option A, the optimistic-and-apparently-simpler approach

- Option B, the safe-but-complex approach:
- Pipe the upload from the client directly to S3, in real-time (streaming).

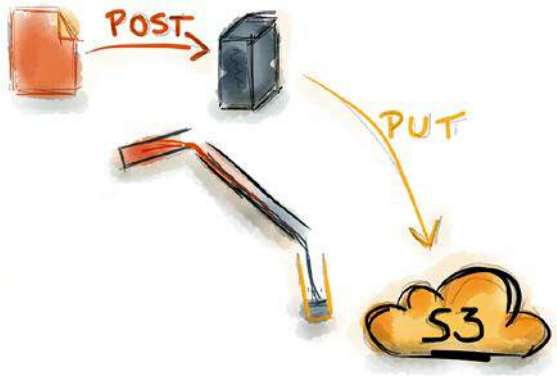


Figure 16.2 Option A, the optimistic-and-apparently-simpler approach

THE OPTIMISTIC-AND-APPARENTLY-SIMPLER APPROACH

Returning a short URL after receiving the file creates expectation — one could even go as far as call it *an implicit contract* — that the file is immediately available at that URL. Being unable to guarantee that the second stage of the upload would *eventually* succeed (i.e. actually pushing the file to S3), the user could end up with a broken link, a link he could have posted on Twitter or sent to an important client. Unnacceptable, even if it happens to one in every hundred thousand uploads.

Our current numbers show that we have an upload fail rate slightly below 0.01% (1 in every 10000), the vast majority being connection timeouts between client and server before the upload actually completes.

We could try and work around it by serving the file from the machine that received it until it finally got pushed to S3 but this is approach is in itself a can of worms:

- If the machine fails before a batch of files is completely uploaded to S3, the files would be forever lost;
- Synchronization issues across the cluster (i.e. *"where is the file for this drop?"*);
- Extra, complex logic to deal with edge cases — that keeps creating more edge cases.

Thinking through all the pitfalls with every workaround, you quickly realize that it's a classic hydra problem — for each head you chop off, two more appear in its place.

THE SAFE-BUT-COMPLEX APPROACH

The other option required low-level control over the whole process. In essence, I had to be able to:Open a connection to S3 while receiving the upload from the client;

- Pipe data from the client connection to the S3 connection;
- Buffer and throttle both connections;
- Buffering is required to keep a steady flow between both *client-to-server* and *server-to-S3* legs of the upload.
- Throttling, on the other hand, is required to prevent explosive memory consumption in case the *server-to-S3* leg of the upload becomes slower than the client-to-server leg.
- Cleanly roll everything back on both ends in case things went wrong.

Seems conceptually simple but it's hardly something your average webserver can offer. Especially when you consider that in order to throttle a TCP connection, you need low-level access to its socket.

It also introduced a completely new challenge that would ultimately end up shaping our final architecture: deferred thumbnail creation.

This meant that whatever the technology stack the platform would end up being built upon, it had to offer not only a few basic things like incredible performance and stability but it would also the flexibility to go *bare metal* (read *down to the bytes*) if required.

16.1.4 The Technology Stack

When kickstarting a new project for a webserver, you'll end up asking yourself:

> Ok, so what's the framework the cool kids are using these days?
>
> I did too.

Going with Netty wasn't a *no-brainer*; I explored plenty of frameworks with three factors, that I considered to be paramount, in mind:

1. It had to to be **fast** — I wasn't about to replace a low-performance stack with another low-performance stack;
2. It had to **scale** — whether it had 1 connection or 10000 connections, each server instance would have to be able to sustain throughput without crashing or leaking memory over time;

3. It had to offer me **low-level data control** — read bytes, trigger TCP congestion control, the works.

Factors 1 and 2 pretty much excluded any non-compiled language. I am a sucker for Ruby and love lightweight frameworks like Sinatra and Padrino but I knew the kind of performance I was looking for could not be achieved by building on these blocks.

Factor 2, on its own, meant that whatever the solution was, it could not rely on blocking I/O — after reading this book you certainly understand why non-blocking I/O was the only option.

Factor 3 was trickier. It meant finding the perfect balance between a framework that would offer low-level control of the data it received but at the same time would be fast to develop with and build upon — this is where language, documentation, community and other success stories come into play.

At this point I had a strong feeling Netty was my weapon of choice.

THE BASICS: A SERVER AND A PIPELINE

The server is merely a `ServerBootstrap` built with a `NioServerSocketChannelFactory`, configured with a few common handlers and a HTTP request controller at the end as shown in Listing 16.1

Listing 16.1 Setup the ChannelPipeline

```
pipelineFactory = new ChannelPipelineFactory() {
    @Override public ChannelPipeline getPipeline()
            throws Exception {
        ChannelPipeline pipeline = Channels.pipeline();
        pipeline.addLast("idleStateHandler", new IdleStateHandler(...)); #1
        pipeline.addLast("httpServerCodec", new HttpServerCodec());      #2
        pipeline.addLast("requestController",
                new RequestController(...));                             #3
        return pipeline;
    }
};
```

#1 An instance of IdleStateHandler, to shut down inactive connections;
#2 An instance of HttpServerCodec, to convert incoming bytes into HttpRequest instances and outgoing HttpResponse into bytes;
#3 An instance of RequestController, the only piece of custom DropIr code in the pipeline, which is responsible for initial request validations and, if all is well, routing the request to the appropriate request handler.

The request controller is probably the most complex part of the whole webserver. A new request controller is created for every connection that opens — a *client* — and lives for as long as that connection is open and performing requests.

It is responsible for:

- Handling load peaks
- HTTP pipeline management
- Setup a context for request handling

- Spawning new request handlers
- Feeding request handlers
- Handle internal and external errors

Listing 16.2 does a quick run-down of the relevant parts of the `RequestController`:

Listing 16.2 Setup the ChannelPipeline

```
public class RequestController
        extends IdleStateAwareChannelUpstreamHandler {

    @Override public void channelIdle(ChannelHandlerContext ctx, IdleStateEvent e)
            throws Exception {
        // Shut down connection to client and roll everything back.
    }

    @Override public void channelConnected(ChannelHandlerContext ctx,
      ChannelStateEvent e)
            throws Exception {
        if (!acquireConnectionSlot()) {
            // Maximum number of allowed server connections reached, respond with
            // 503 service unavailable and shutdown connection.
        } else {
            // Setup the connection's request pipeline.
        }
    }

    @Override public void messageReceived(ChannelHandlerContext ctx, MessageEvent e)
        throws Exception {
    if (isDone()) return;

    if (e.getMessage() instanceof HttpRequest) {
        handleHttpRequest((HttpRequest) e.getMessage());                #1
    } else if (e.getMessage() instanceof HttpChunk) {
        handleHttpChunk((HttpChunk) e.getMessage());                    #2
    }
}
```

#1 This is where the gist of Droplr's server request validation
#2 If there is an active handler for the current request and it accepts chunks, pass on the chunk.

As explained in this book, you should never execute non-CPU bound code on Netty's I/O threads — you'll be *stealing* away precious resources from Netty and thus affecting the server's throughput.

For this reason, both the `HttpRequest` and `HttpChunk` may hand off the execution to the request handler by switching over to a different thread — this happens when the request handlers are not CPU-bound, either because they access the database or perform any logic that is not confined to local memory or CPU.

When thread switching occurs, it is imperative that all the blocks of code execute in serial fashion; otherwise we'd risk, e.g. for an upload, having a `HttpChunk` *n-1* being processed after `HttpChunk` *n* and thus corrupting the body of the file (we'd be swapping how bytes were laid out in the uploaded file.)

To cope with this, I created a custom thread pool executor that ensures all tasks sharing a common identifier will be executed serially.

From here on, the data (requests and chunks) ventures out of the Netty realm and Droplr's.

I'll briefly explain how the request handlers are built for the sake of shedding some light on the bridge between the `RequestController` — which lives in Netty-land — and the handlers — Droplr-land. Who knows, maybe this will help you architect your own server!

THE REQUEST HANDLERS

Request handlers are what provide Droplr's functionality. They are the endpoints behind URIs such as `/account` or `/drops`. They are the logic cores, the server's interpreters to clients' requests.

Request handler implementations are where the framework actually becomes Droplr's API server.

THE PARENT INTERFACE

Each request handler, directly or through a sub-class hierarchy, is a realization of the the interface `RequestHandler`.

In its essence, the `RequestHandler` interface represents a stateless handler for requests (instances of `HttpRequest`) and chunks (instances of `HttpChunk`). It's an extremely simple interface with a couple of methods to help the request controller perform and/or decide how to perform its duties, such as:

- Is the request handler stateful or stateless? — i.e. does it need to be cloned from a prototype or can the prototype be used to handle the request;
- Is the request handler CPU or non-CPU bound? — i.e. can it execute on Netty's worker threads or should it be executed in a separate thread pool;
- Create a new freshly uninitialized
- Rollback current changes;
- Cleanup any used resources.

This interface is all the Request Controller knows about actions. Through its very clear and concise interface the controller can interact both with stateful, stateless, CPU and non-CPU-bound handlers (or combinations of the previous) in a completely isolated and implementation-agnostic fashion.

HANDLER IMPLEMENTATIONS

Through a sub-class hierarchy that starts at `AbstractRequestHandler` — the simplest of the realizations of `RequestHandler` — and grows ever more specific in each step it takes down to the actual handlers that provide all of Droplr's functionality, this very broad interface eventually becomes the stateful non-CPU bound (executed in a non-IO-worker thread) `SimpleHandler` which is ideal for quick implementation of endpoints that do the typical tasks of reading in some JSON, hitting the database and then writing out some JSON.

THE UPLOAD REQUEST HANDLER

The upload request handler is the *crux* of the whole Droplr API server. It was the action that shaped the design of the **webserver** module — the *frameworky* part of the server — and is by far the most complex and tuned piece of code of the whole stack.

During uploads, the server has dual behavior:

- On one side it acts as a server for the API clients which are uploading the files;
- On the other side it acts as client to S3, to push the data it receives from the API clients.

To act as a client, the server uses a http client library[^footnote-http-client] that is also built with Netty. This asynchronous library exposes an interface that perfectly matches the needs of the server — begin executing a HTTP request and allow data to be fed to it as it becomes available — and greatly reduces the complexity of the client facade of the upload request handler.

[^footnote-http-client]: You can find the http client library at https://github.com/brunodecarvalho/http-client

16.1.5 Performance

After the initial version of the server was complete, I ran a batch of performance tests. The results were nothing short of mind blowing: after continuously increasing the load in disbelief, the new server peaked at 10~12x faster uploads over the old LAMP stack — a full order of magnitude faster! — and could handle over 1000x more concurrent uploads, in a total of nearly 10k concurrent uploads (running on a single EC2 large instance).

Factors that contributed to this were:

- It was running in a tuned JVM;
- It was running in a highly tuned custom stack, created specifically to address this problem, instead of an all-purpose web framework;
- The custom stack was built with Netty using NIO (selector based model) which meant it could scale to tens or even hundreds of thousands of concurrent connections, unlike the one-process-per-client LAMP stack;
- There was no longer the overhead of receiving a full file and then uploading it to S3 in two separate phases — the file was now piped/streamed directly to S3;
- Since the server was now streaming files:
- it was not spending time on I/O operations, writing to temporary files and later reading them in the second stage of the upload;
- it was using less memory for each upload, which means more parallel uploads could take place;
- Thumbnail generation became an asynchronous post-process.

16.1.6 Summary – Standing on the shoulders of Giants

All of this was only possible due to Netty's incredibly well-designed API and performant non-blocking I/O architecture.

Since the launch of Droplr 2.0 on December 2011, we've had virtually zero downtime at the API level. A couple months ago we interrupted a year-and-a-half clean run of 100% infrastructure uptime due to a scheduled fullstack upgrade (databases, OS, major server & daemons codebase upgrade) that took just under an hour.

The servers soldier on, day after day, taking hundreds — sometimes thousands — of concurrent requests per second, all the while keeping both memory and CPU usage to levels so low it's hard to believe it's actually doing such an incredible amount of work:

- CPU usage rarely ever goes above 5%;
- Memory footprint can't be accurately described as I start the process with 1GB of pre-allocated memory (however, I do configure the JVM to grow up to 2GB if necessary and not a single time over the past two years has it happened...)

Anyone can throw more machines at any given problem but Netty helped Droplr scale intelligently... And keep the server bills pretty low ;)

16.2 Firebase – A Real-Time Data Synchronization Service

Written by Sara Robinson, VP of Developer Happiness at Firebase; and Greg Soltis, VP of Cloud Architecture at Firebase.

Real-time updates are an integral part of the user experience in modern applications. As users come to expect this behavior, more and more applications are pushing data changes to users in real-time. Real-time data synchronization is difficult to achieve with the traditional three-tiered architecture which requires developers to manage their own ops, servers, and scaling. By maintaining a real-time bidirectional communication to the client, Firebase makes it easy for developers to achieve real-time synchronization in their applications without running their own servers.

Implementing this presented a difficult technical challenge, and Netty was the optimal solution in building the underlying framework for all network communications in Firebase. This chapter will provide an overview of Firebase's architecture, and then examine three ways Firebase uses Netty to power its real-time synchronization service:

1. Long Polling
2. HTTP 1.1 Keep-Alive and Pipelining
3. Control of SSL Handler

16.2.1 The Firebase Architecture

Firebase allows developers to get an application up and running using a two-tiered architecture. Developers simply include the Firebase library and write client-side code. The data is exposed to the developer's code as JSON and operates off of a local cache of the data. The library handles synchronizing this local cache with the master copy, which is stored on

Firebase's servers. Changes made to any piece of data are synchronized in real-time to potentially hundreds of thousands of clients connected to Firebase. The interaction between multiple clients across platforms and devices and Firebase is depicted in the Figure 18.3.

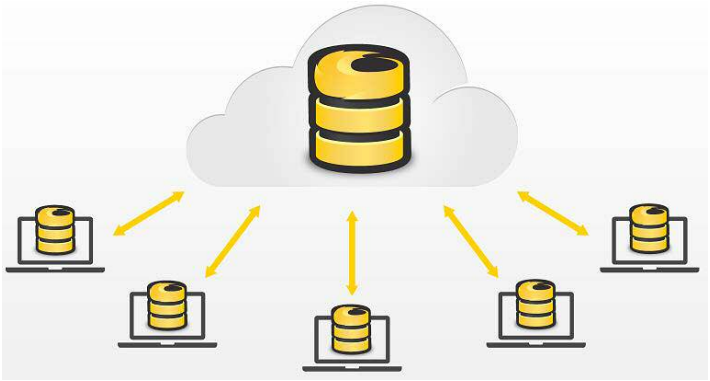


Figure 16.3 Firebase Architecture

Firebase servers take incoming data updates and immediately synchronize them to all of the connected clients that have registered interest in the changed data. To enable real-time notification of state changes, clients maintain an active connection to Firebase at all times. This connection may range from an abstraction over a single Netty channel to an abstraction over multiple channels or even multiple, concurrent abstractions if the client is in the middle of switching transport types.

Given that clients can connect to Firebase in a variety of ways, it is important to keep the connection code modular. Netty's Channel abstraction is a fantastic building block for integrating new transports into Firebase. In addition, the pipeline-and-handler pattern makes it simple to keep transport-specific details isolated and provide a common message stream abstraction to the application code. Similarly, this greatly simplifies adding support for new protocols. Firebase added support for a binary transport simply by adding a few new handlers to the pipeline.

Netty's pipeline and encoder classes encourage modularity by separating serialization and deserialization from application logic. By giving each handler a singular focus, a logical separation of concerns is achieved. Netty's speed, level of abstraction, and fine-grained control made it an excellent framework for implementing real-time connections between the client and server.

16.2.2 Long Polling

Firebase uses both long polling and websocket transports. The long polling transport is highly reliable across all browsers, networks, and carriers whereas the websocket-based transport is faster but not always available. Initially, Firebase connects using long-polling and then upgrades to websockets if possible. For the 5% of Firebase traffic that doesn't support

websockets, Firebase used Netty to implement a custom library for long polling tuned to be highly performant and responsive.

The Firebase application logic deals with bidirectional streams of messages with notifications when either side closes the stream. While this is relatively simple to implement on top of TCP or websockets, it presents a challenge when dealing with a long polling transport. The two properties that must be enforced to achieve this goal are:

1. Guaranteed in-order delivery of messages
2. Close notifications

GUARANTEED IN-ORDER DELIVERY OF MESSAGES

In-order delivery for long polling can be achieved by having only one single request outstanding at a given time. Since the client won't send another request until it receives a response from its last request, it can guarantee that its previous messages were received and that it is safe to send more. Similarly, on the server side, there won't be a new request outstanding until the client has received the previous response. Therefore, it is always safe to send everything that is buffered up in between. However, this leads to a major drawback. Using the single-request technique, both the client and server spend a significant amount of time buffering up messages. For example, if the client has new data to send but already has an outstanding request, it must wait for the server to respond before sending the new request. This could take a long time if there is no data available on the server.

A more performant solution is to tolerate more requests being in-flight concurrently. In practice, this can be achieved by swapping single-request for at-most-two-requests. The algorithm has two parts:

1. Whenever a client has new data to send, it sends a new request unless two are already in flight.
2. Whenever the server receives a request from a client, if it already has an open request from the client, it immediately responds to the first even if there is no data.

This has an important improvement over single-request: both the client and server's buffer time is bound to at most a single network round trip.

Of course, this increase in performance does not come without a price, as it results in a commensurate increase in code complexity. The long polling algorithm no longer guarantees in-order delivery, but a few ideas from TCP can ensure that messages are delivered in order. Each request sent by the client includes a serial number, incremented for each request. In addition, each request includes metadata about the number of messages in the payload. If a message spans multiple requests, the portion of the message contained in this payload is included in the metadata.

The server maintains a ring buffer of incoming message segments and processes them as soon as they are complete with no incomplete messages ahead of them. Downstream is easier since the long polling transport responds to an HTTP GET request and doesn't have the same restrictions on payload size. In this case, a serial number is included and is incremented once

for each response. The client can process all messages in the list as long as it has received all responses up to the given serial number. If it hasn't, it buffers the list until it receives the outstanding responses.

Close Notifications

The second property enforced in the long polling transport is close notifications. In this case, having the server aware that the transport has closed is significantly more important than having the client recognize the close. Firebase clients queue up operations to be run when a disconnect occurs, and those operations can have an impact on other, still-connected clients. So, it is important to know when a client has actually gone away. Implementing a server-initiated close is relatively simple and can be achieved by responding to the next request with a special protocol-level 'close' message.

Implementing client-side close notifications is trickier. The same 'close' notification can be used, but there are two things that can cause this to fail: the user can close the browser tab, or the network connection could disappear. The tab-closure case is handled with an iframe that fires a request containing the 'close' message on page unload. The second case is dealt with via a server-side timeout. It is important to pick your timeout values carefully, since the server is unable to distinguish a slow network from a disconnected client. That is to say, there is no way for the server to know that a request was actually delayed for a minute, rather than the client losing its network connection. It is important to choose an appropriate timeout that balances the cost of false positives (closing transports for clients on slow networks) against how quickly the application needs to be aware of disconnected clients.

The diagram below demonstrates how the Firebase long polling transport handles different types of requests:

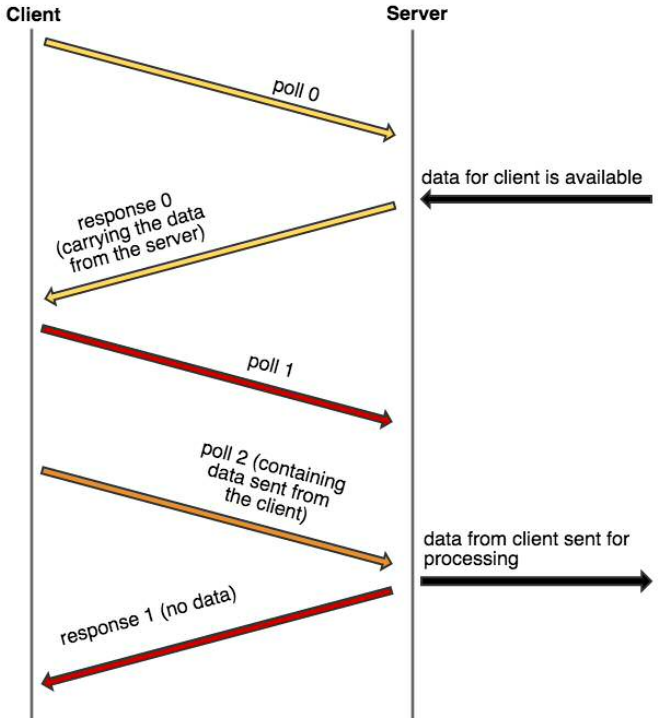


Figure 16.4 Long Polling

In this diagram, each long poll request indicates different types of scenarios. Initially, the client sends a poll (poll 0) to the server. Some time later, the server receives data from elsewhere in the system that is destined for this client, so it responds to poll 0 with the data. As soon as the poll returns, the client sends a new poll (poll 1), since it currently has none outstanding. A short time later, the client needs to send data to the server. Since it only has a single poll outstanding, it sends a new one (poll 2) that includes the data to be delivered. Per the protocol, as soon as the server has two simultaneous polls from the same client, it responds to the first one. In this case, the server has no data available for the client, so it sends back an empty response. The client also maintains a timeout, and will send a second poll when it fires even if it has no additional data to send. This insulates the system from failures due to browsers timing out slow requests.

16.2.3 HTTP 1.1 Keep-Alive and Pipelining

With HTTP 1.1 keep-alive, multiple requests can be sent on one connection to a server. This allows for pipelining--new requests can be sent without waiting for a response from the server. Implementing support for pipelining and keep-alive is typically straightforward, but gets significantly more complex when mixed with long polling.

If a long polling request is immediately followed by a REST request, there are some considerations that need to be taken into account to ensure the browser performs properly. A channel may mix asynchronous messages (long poll requests) with synchronous messages (REST requests). When a synchronous request comes in on one channel, Firebase must synchronously respond to all preceding requests in that channel in order. For example, if there is an outstanding long poll request, the long polling transport needs to respond with a no-op before handling the REST request.

The diagram below illustrates how Netty lets Firebase respond to multiple request types in one socket.

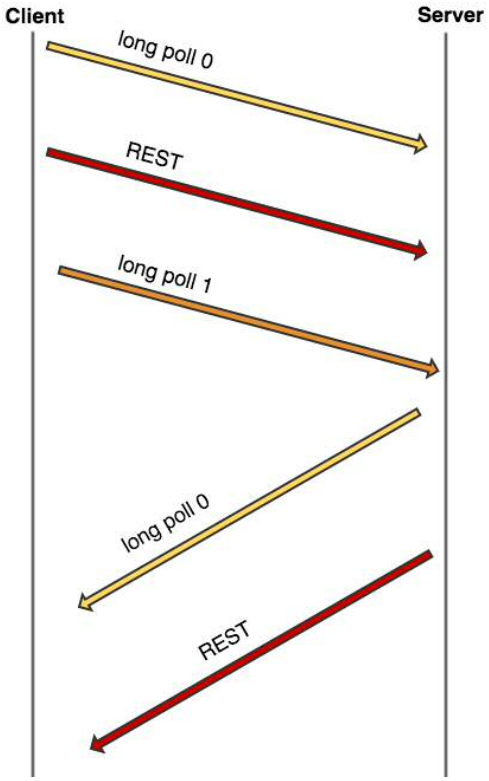


Figure 16.5 Network diagram

If the browser has more than one connection open and is using long polling, it will reuse the connection for messages from both of those tabs. Given long polling requests, this is difficult and requires proper management of a queue of HTTP requests. Long polling requests can be interrupted, but proxied requests cannot. Netty made serving multiple request types easy:

- Static HTML pages: cached content that can be returned with no processing; examples include a single page HTML app, robots.txt, and crossdomain.xml.
- REST requests: Firebase supports traditional GET, POST, PUT, DELETE, PATCH, and OPTIONS requests.
- Websockets: a bidirectional connection between a browser and a Firebase server with its own framing protocol.
- Long polling: these are similar to HTTP GET requests, but are treated differently by the application.
- Proxied requests: Some requests can't be handled by the server that receives them. In that case, Firebase proxies the request to the correct server in its cluster, so that end users don't have to worry about where data is located. These are like the REST requests, but the proxying server treats them differently.
- Raw bytes over SSL: a simple TCP socket running Firebase's own framing protocol and optimized handshaking.

Firebase uses Netty to set up its pipeline to decode an incoming request and then reconfigure the remainder of the pipeline appropriately. In some cases, like websockets and raw bytes, once a particular type of request has been assigned a channel, it will stay that way for its entire duration. In other cases, like the various HTTP requests, the assignment must be made on a per-message basis. The same channel could handle REST requests, long polling requests, and proxied requests.

16.2.4 Control of SSL handler

Netty's SslHandler class is an example of how Firebase uses Netty for fine-grained control of its network communications. When a traditional web stack uses an HTTP server like Apache or Nginx to pass requests to code, incoming SSL requests have already been decoded at the point when they are received by the application code. With a multi-tenant architecture, it is difficult to assign portions of the encrypted traffic to the tenant of the application using a specific service. This is complicated by the fact that multiple applications could use the same encrypted channel to talk to Firebase (for instance, the user might have two Firebase applications open in different tabs). To solve this, Firebase needed enough control in handling SSL requests before they were decoded.

Firebase charges customers based on bandwidth. However, the account to be charged for a message is typically not available before the SSL decryption has been performed, since it is contained in the encrypted payload. Netty allows Firebase to intercept traffic at multiple points in the pipeline so they can start counting bytes as they come in off the wire. After the message has been decrypted and processed by Firebase's application logic, the byte count can be assigned to the appropriate account. In building this feature, Netty provided control for handling network communications at every layer of the protocol stack, and allowed for very accurate billing, throttling and rate limiting--all of which had significant business implications.

Netty made it possible to intercept all inbound and outbound messages and count bytes in a small amount of code:

Listing 16.3 Setup the ChannelPipeline

```
case class NamespaceTag(namespace: String)

class NamespaceBandwidthHandler extends ChannelDuplexHandler {
    private var rxBytes: Long = 0
    private var txBytes: Long = 0
    private var nsStats: Option[NamespaceStats] = None

    override def channelRead(ctx: ChannelHandlerContext, msg: Object) {
        msg match {
            case buf: ByteBuf => {
                rxBytes += buf.readableBytes(                              #1
                                    tryFlush(ctx)
            }
            case _ => { }
        }
        super.channelRead(ctx, msg)
    }

     override def write(ctx: ChannelHandlerContext, msg: Object,
             promise: ChannelPromise) {
         msg match {
             case buf: ByteBuf => {                                        #2
                 txBytes += buf.readableBytes()
                 tryFlush(ctx)
                 super.write(ctx, msg, promise)
             }
             case tag: NamespaceTag => {                                   #3
                 updateTag(tag.namespace, ctx)
             }
             case _ => {
                 super.write(ctx, msg, promise)
             }
         }
     }

     private def tryFlush(ctx: ChannelHandlerContext) {
         nsStats match {
             case Some(stats: NamespaceStats) => {                         #4
                 stats.logOutgoingBytes(txBytes.toInt)
                 txBytes = 0
                 stats.logIncomingBytes(rxBytes.toInt)
                 rxBytes = 0
             }
             case None => {
                 // no-op, we don't have a namespace
             }
         }
    }

    private def updateTag(ns: String, ctx: ChannelHandlerContext) {
         val (_, isLocalNamespace) = NamespaceOwnershipManager.getOwner(ns)
         if (isLocalNamespace) {
             nsStats = NamespaceStatsListManager.get(ns)
             tryFlush(ctx)
         } else {
             // Non-local namespace, just flush the bytes
```

```
            txBytes = 0                                                    #5
             rxBytes = 0
          }
      }
}
```

#1 Whenever a message comes in, count the number of bytes
#2 Whenever there is an outbound message, count those bytes as well
#3 If a tag is received, tie this channel to a particular account (a Namespace in the code), remember the account and assign the current byte counts to that account
#4 I If there is already a tag for the namespace the channel belongs to, assign the bytes to that account and reset the counters
#5 In some cases, the count is not applicable to this machine, so ignore it and reset the counters

16.2.5 Summary

Netty plays an indispensable role in the server architecture of Firebase's real-time data synchronization service. It allows support for a heterogeneous client ecosystem which includes a variety of browsers, along with clients that are completely controlled by Firebase. With Netty, Firebase can handle tens of thousands of messages per second on each server. Netty is especially awesome because:

- It's fast: it took only a few days to develop a prototype, and was never a production bottleneck.
- It handles server restarts extremely well: when new code is deployed there is a multiplicity of connections. During the reconnect, there is an amplification of activity and spike in the number of packets, all of which Netty handles smoothly.
- It is positioned well in the abstraction layer: Netty provides fine-grained control where necessary and allows for customization at every step of the control flow.
- It supports multiple protocols over the same port--HTTP, websockets, long polling, and standalone TCP.
- Its GitHub repo is top-notch: the build compiles and is easy to run locally, making it frictionless to develop against.
- It has a highly active community: the community is very responsive on issue maintenance, and seriously considers all feedback and pull requests. In addition, the team provides great and up-to-date example code. Netty is an excellent, well-maintained framework and has been essential in building and scaling Firebase's infrastructure. Real-time data synchronization in Firebase would not be possible without Netty's speed, control, abstraction and extraordinary team.

16.3 Urban Airship - Building Mobile Services

Written by Erik Onnen; Vice President of Architecture at Urban Airship

As smart phone usage grows across the globe at unprecedented rates, a number of service providers have emerged to assist developers and marketers towards the end of providing amazing end user experiences. Unlike their feature phone predecessors, these smart phones crave IP connectivity and seek it across a number of channels (3G, 4G, Wifi, WiMAX and

Bluetooth). As more and more of these devices access public networks via IP-based protocols, the challenges of scale, latency and throughput become more and more daunting for back end service providers.

Thankfully, Netty is well suited to many of the concerns faced by this thundering herd of always connected mobile devices. This chapter will detail several practical applications of Netty in scaling a mobile developer and marketer platform, Urban Airship.

16.3.1 Basics of Mobile Messaging

While marketers have long used SMS as a channel to reach mobile devices, a more recent functionality called Push Notifications is rapidly becoming the preferred mechanism for messaging smart phones. Push notifications commonly use the less expensive data channel and the price per message is a fraction of the cost of SMS. The throughput of push notifications is commonly two to three orders of magnitude higher than SMS making it an ideal channel for breaking news. Most importantly, push notifications give users device-driven control of the channel. If a user dislikes the messaging from an application, the user can disable notifications for an application or outright delete the application.

At a very high level, the interaction between a device and push notification behavior is similar to the depiction in Figure 18.5

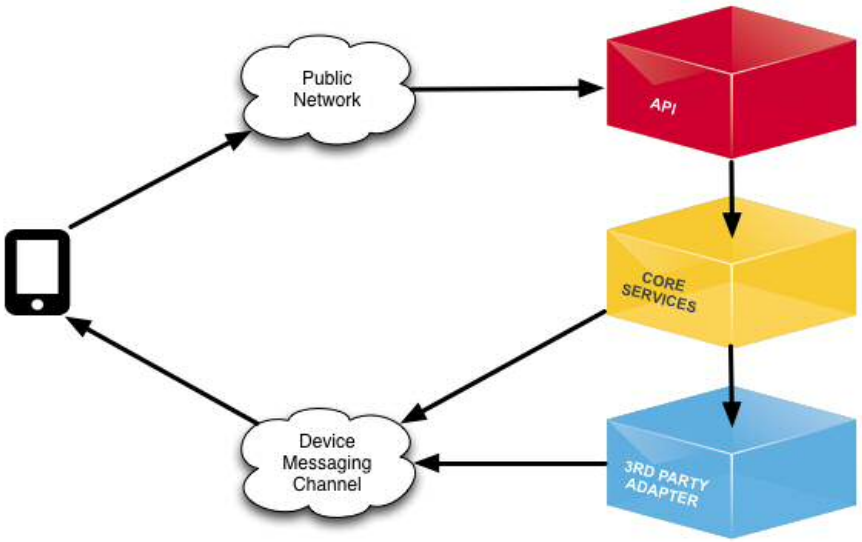


Figure 16.5 High-Level Mobile Messaging Platform Integration

At a high level, when an application developer desires to send push notifications to a device[38], the developer must plan to store information about the device and its application installation. Commonly, an application installation will execute code to retrieve a platform-specific identifier and report that identifier back to a centralized service where the identifier is persisted. Later, logic external to the application installation will initiate a request to deliver a message to the device.

nce an application installation has registered its identifier with a back end service, the delivery of a push message can in turn take two paths. In the first path, a message can be delivered directly to the application itself with the application maintaining a direct connection to a back end service. In the second and more common approach, an application will rely on a third party to deliver the message to the application on behalf of back end service. At Urban Airship, both approaches to delivering push notifications are used and both leverage Netty extensively.

16.3.2 Third-Party Delivery

In the case of 3rd party push delivery, every push notification platform provides a different API for developers to deliver messages to application installations. These APIs differ in terms of their protocol (binary vs. text), authentication (OAuth, X.509, etc.) and capabilities. Each approach has its own unique challenges for integration as well as achieving optimal throughput.

Despite the fact that the fundamental purpose of each of these providers is to deliver a notification to an application, each takes a different approach with significant implications to system integrators. For example, integration with Apple's APNS service is strictly a binary protocol where as other providers base their service on some form of HTTP but all with subtle variations that affect how to best achieve maximum throughput. Thankfully, Netty is an amazingly flexible tool and significantly helps smoothing over the differences between the various protocols.

The following sections will provide examples of how Urban Airship uses Netty to integrate with two of the listed providers.

16.3.3 Binary Protocol Example

Apple's APNS protocol is a binary protocol with a specific, network byte ordered payload. Sending an APNS notification involves the following sequence of events:

1. Connect a TCP socket to APNS (Apple Push Notification Service) servers over an SSLv3 connection, authenticated with an x509 certificate.

[38] Some mobile operating systems allow a form of push notifications called local notifications that would not follow this approach.

2. Format a binary representation of a push message structured according to the format defined by Apple[39].
3. Write the message to the socket.
4. Read from the socket if ready to determine any error codes associated with a sent message.
5. In the case of an error, reconnect the socket and repeat the previous sequence of activities.

As part of formatting the binary message, the producer of the message is required to generate an identifier that is opaque to the APNS system. In the event of an invalid message (formatting, incorrect size, incorrect device information), the identifier will be returned to the client in the error response message of step 4.

While at face value the protocol seems straight forward, there are several nuances to successfully addressing all of the concerns above, in particular on the JVM:

- The APNS specification dictates that certain payload values should be sent in big endian ordering (token length for example).
- Step three in the above sequence requires one of two solutions. Because the JVM will not allow reading from a closed socket, even if data exists in the output buffer, developers have the options of:
- After a write, performing a blocking read with a timeout on the socket. This has multiple disadvantages:
- The amount of time to block waiting for an error is non-deterministic. An error may occur in milliseconds or seconds.
- As socket objects cannot be shared across multiple threads, writes to the socket must immediately block waiting for errors. This has dramatic implications for throughput. If a single message is delivered in a socket write, no additional messages can go out on that socket until the read timeout has occurred. When delivering 10s of millions of messages, a three second delay between messages is not acceptable.
- Relying on a socket timeout is an expensive operation. It results in an exception being thrown and several unnecessary system calls.
- Using Asynchronous I/O. Under this model, both reads and writes do not block. This allows writers to continue sending messages to APNS while at the same time allowing the operating system to inform user code when data is ready to be read.

Netty makes addressing all of these concerns trivial while at the same time delivering amazing throughput.

First, Netty makes packing a binary APNS message with correct endian ordering trivial.

[39] http://bit.ly/189mmpG

Listing 16.4 ApnsMessage implementation

```
public final class ApnsMessage {
    private static final byte COMMAND = (byte) 1;                          #1
    public ByteBuf toBuffer() {
        short size = (short) (1 + // Command                               #2
                4 + // Identifier
                4 + // Expiry
                2 + // DT length header
                32 + //DS length
                2 + // body length header
                body.length);

        ByteBuf buf = Unpooled.buffer(size).order(ByteOrder.BIG_ENDIAN); #3
        buf.writeByte(COMMAND);
        buf.writeInt(identifier);                                          #4
        buf.writeInt(expiryTime);
        buf.writeShort((short) deviceToken.length);                        #5
        buf.writeBytes(deviceToken);
        buf.writeShort((short) body.length);
        buf.writeBytes(body);
        return buf;                                                        #6
    }
}
```

#1 An APNS message always starts with a command 1 byte in size so that value is coded as a constant.
#2 Messages are of varying sizes. To avoid unnecessary memory allocation and copying, the exact size of the message body is calculated before the ByteBuf is created.
#3 When creating the ByteBuf, it is sized exactly as large as needed and, the appropriate endianness for APNS is explicitly specified.
#4 Various values are inserted into the buffer from state maintained elsewhere in the class.
#5 The deviceToken field in this class (not shown) is a Java byte[]. The length property of a Java array is always an integer. In this case though, the APNS protocol requires a two byte value. In this case, the length of the payload has been validated elsewhere so casting to a short is safe at this location. Note that without explicitly constructing the ByteBuf to be big endian, subtle bugs could occur with values of types short and int.
#6 Note that when the buffer is ready, it is simply returned. Unlike the standard java.nio.ByteBuffer, it is not necessary to flip the buffer and worry about its position, Netty's ByteBuf handles read and write position management automatically.

In a small amount of code, Netty has made trivial the act of creating a properly formatted APNS message. Because this message is now packed into a `ByteBuf`, it can easily be written directly to a Channel connected to APNS when ready to send the message.

Connecting to APNS can be accomplished via multiple mechanisms but at its most basic, a `ChannelInitializer` that populates the `ChannelPipeline` with an `SslHandler` and a decoder is required.

Listing 16.5 Setup the ChannelPipeline

```
public final class ApnsClientPipelineInitializer
        extends ChannelInitializer<Channel> {
    private final SSLEngine clientEngine;

    public ApnsClientPipelineFactory(SSLEngine engine) {
        this.clientEngine = engine;                                        #1
```

```
    }

    @Override
    public void initChannel(Channel channel) throws Exception {
        final ChannelPipeline pipeline = channel.pipeline();              #2
        final SslHandler handler = new SslHandler(clientEngine);          #3
        handler.setEnableRenegotiation(true);                             #4
        pipeline.addLast("ssl", handler);
        pipeline.addLast("decoder", new ApnsResponseDecoder());           #5
    }
}
```

#1 To perform an X.509 authenticated request, a javax.net.ssl.SSLEngine object instance is required. Constructing this object is beyond the scope of this chapter but in general requires a certificate and secret key stored in a keystore object.
#2 Obtain the ChannelPipeline, which will hold the SslHandler and a handler to process disconnect responses from APNS.
#3 Construct a Netty SslHandler using the SSLEngine provided in the constructor.
#4 In particular, APNS will attempt to renegotiate an SSL connection shortly after connection so allowing renegotiation is required in this case.
#5 An extension of Netty's ByteToMessageDecoder, the ApnsResponseDecoder (not listed here) handles cases where APNS returns an error code and disconnects.

It is worth noting how easy Netty makes negotiating an X.509 authenticated connection in conjunction with Asynchronous I/O. In early prototypes of APNS code at Urban Airship without Netty, negotiating an asynchronous X.509 authenticated connection required over 80 lines of code and an executor thread pool simply to connect. Netty hides all the complexity of the SSL handshake, the authentication and most importantly encryption of cleartext bytes to cipher text and the key renegotiation that comes along with using SSL. These incredibly tedious, error prone and poorly documented APIs in the JDK are hidden behind three lines of Netty code.

At Urban Airship, Netty plays a roll in all connectivity to third party push notification services including APNS, Google's GCM and various other third party providers. In every case, Netty is flexible enough to allow explicit control over exactly how integration takes place from higher-level HTTP connectivity behavior down to basic socket level settings such as TCP keep-alive and socket buffer sizing.

16.3.4 Direct to Device Delivery

The previous section provides insight into how Urban Airship integrates with a third party for message delivery. In referring back to Figure 16.1, note that two paths exist for delivering messages to a device. In addition to delivering messages through a third party, Urban Airship also has experience serving directly as a channel for message delivery. In this capacity, individual devices connect directly to Urban Airship's infrastructure bypassing third party providers. This approach brings a distinctly different set of challenges, namely:

- Socket connections from mobile devices are often short lived – mobile devices frequently switch between different types of networks depending on various conditions. To back end providers of mobile services, devices constantly reconnect and experience

short but frequent periods of connectivity.

- Connectivity across platforms is irregular – from a network perspective, tablet devices tend to behave differently than mobile phones and mobile phones behave differently than desktop computers.
- Frequency of mobile phone updates to back end providers – mobile phones are increasingly used for daily tasks producing significant amounts of general network traffic but also analytics data for back end providers.
- Battery and bandwidth cannot be ignored – unlike a traditional desktop environment, mobile phones tend to operate on limited data plans. Service providers must honor the fact that end users have limited battery life and use expensive, limited bandwidth. Abuse of either will frequently result in the uninstallation of an application, the worst possible outcome for a mobile developer.
- Massive scale – as mobile device popularity increases, more application installations result in more connections to a mobile services infrastructure. Each of the previous elements in this list are further complicated by the sheer scale and growth of mobile devices.

Over time, Urban Airship learned several critical lessons as connections from mobile devices continued to grow:

- Diversity of mobile carriers can have a dramatic effect on device connectivity.
- Many carriers do not allow TCP keep-alive functionality. Given that, many carriers will aggressively cull idle TCP sessions.
- UDP is not a viable channel for messaging to mobile devices as it is disallowed by many carriers.
- The overhead of SSLv3 is an acute pain for short-lived connections.

Given the challenges of mobile growth and the lessons learned by Urban Airship, Netty was a natural fit for implementing a mobile messaging platform for several reasons highlighted in the following sections.

16.3.5 Netty excels at managing large numbers of concurrent connections

As mentioned in the previous section, Netty makes supporting asynchronous I/O on the JVM trivial. Because Netty operates on the JVM and because the JVM on Linux ultimately uses the Linux epoll facility to manage interest in socket file descriptors, Netty makes it possible to accommodate the rapid growth of mobile by allowing developers to easily accept large numbers of open sockets, close to 1 million TCP connections per single Linux process. At numbers of this scale, service providers can keep costs low, allowing a large number of

devices to connect to a single process on a physical server[40]. In controlled testing and with configuration options optimized to use small amounts of memory, a Netty based service was able to accommodate slightly less than 1 million connections (approximately 998,000). In this case, the limit was fundamentally the Linux kernel imposing a hard-coded limit of one million file handles per process. Had the JVM itself not held a number of sockets and file descriptors for JAR files, the server would likely have been able of handling even further connections, all on a 4GB heap. Leveraging this efficiency, Urban Airship has successfully sustained over 20 million persistent TCP socket connections to its infrastructure for message delivery, all on a handful of servers.

It is worth noting that while in practice a single Netty-based service is capable of handling nearly a million inbound TCP socket connections, doing so is not necessarily pragmatic or advised. As with all things in distributed computing, hosts will fail, processes will need to be restarted and unexpected behavior will occur. As a result of these realities, proper capacity planning means considering the consequences of a single process failing.

16.3.6 Summary - Beyond the Perimiter of the Firewall

The previous two sections of this chapter have demonstrated two every day usages of Netty at the perimeter of the Urban Airship network. While Netty works exceptionally well for these purposes, it has also found a home as scaffolding for many other components inside Urban Airship.

Internal RPC Framework

Netty has been the centerpiece of an internal RPC framework that has consistently evolved inside Urban Airship. Today, this framework processes 100s of 1000s of requests per second with very low latency and exceptional throughput. Nearly every API request fielded by Urban Airship processes through multiple back end services with Netty at the core of all of those services.

Load and Performance Testing

Netty has been used at Urban Airship for several different load and performance testing frameworks. For example, to simulate millions of device connections in testing the previously described device messaging service, Netty was used in conjunction with a Redis[41] instance to test end-to-end message throughput with a minimal client-side footprint.

[40] Note the distinction of a physical server in this case. While virtualization offers many benefits, leading cloud providers were regularly unable to accommodate more than 200-300 thousand concurrent TCP connections to a single virtual host. With connections at or above this scale, expect to use bare metal servers and expect to pay close attention to the NIC (Network Interface Card) vendor.

[41] http://redis.io/

Asynchronous Clients for Commonly Synchronous Protocols

For some internal usages, Urban Airship has been experimenting with Netty to create asynchronous clients for typically synchronous protocols including services like Apache Kafka[42] and Memcached[43]. Netty's flexibility easily allows crafting clients that are asynchronous in nature but can be converted back and forth between truly asynchronous or synchronous implementations without requiring upstream code changes.

All in all, Netty has been a cornerstone of Urban Airship as a service. The authors and community are fantastic and have produced a truly first class framework for anything requiring networking on the JVM.

16.4 Summary

In this chapter you were able to get some insight into real-world usage of Netty and how it helped companies to solve their problem and build up their stack. Hopefully this should give you some better idea how you can solve yours and use Netty in your next Project.

That said there are also other Companies that started to build OpenSource Projects on Top of Netty which itself are used to power their internal needs. In the next Chapter you will learn about two of those OpenSource Projects that were founded by Facebook and Twitter.

[42] http://kafka.apache.org/
[43] http://memcached.org/

17

Case Studies, Part 2: Facebook and Twitter

In this chapter we'll take a look at

- Case-Studies
- Real-World Use-Cases
- Build your own Project on top of Netty

In this chapter you will see how two of the most popular social networks world-wide are using Netty to solve their real-world problems. Both of them build their own services and frameworks on top of Netty which are completely different in terms of needs and usage.

This is only possible because of the very flexible and generic design of Netty. The details in this chapter will serve as insperation but also to show how some problems can be solved by using Netty. Learn from the expirience and issues of Facebook and Twitter, and understand how to solve them by make use of Netty. Both case studies are written by the engineers themselves who are responsible for the design and implementation of the various solutions.

Understand how their use of Netty and design choices allows them to scale and keep up with the needs of the future.

17.1 Netty at Facebook: Nifty and Swift

Written by Andrew Cox; Software Engineer at Facebook

At Facebook, we use Netty in several of our back-end services (for handling messaging traffic from mobile phone apps, for HTTP clients, etc.), but our fastest-growing usage is via two new frameworks we've developed for building Thrift services in Java: Nifty and Swift.

17.1.1 What is Thrift?

Thrift is a framework for building services and clients that communicate via remote procedure calls (RPC). It was originally developed at Facebook[44] to meet our requirements for building services that can handle certain types of interface mismatches between client and server. This comes in very handy here, since services and their clients usually can't all be upgraded simultaneously.

Another important feature of Thrift is that it is available for a wide variety of languages. This enables teams at Facebook to choose the right language for the job, without worrying about whether they will be able to find client code for interacting with other services. Thrift has grown to become one of the primary means by which our back-end services at Facebook communicate with one another, and it is also used for non-RPC serialization tasks, because it

[44] A now-ancient whitepaper from the original Thrift developers can be found here: http://thrift.apache.org/static/files/thrift-20070401.pdf

provides a common, compact storage format that can be read from a wide selection of languages for later processing.

Since it's development at Facebook, Thrift has been open-sourced as an Apache project[45], where it continues to grow to fill the needs of service developers, not only at Facebook but also at other companies – including Evernote and last.fm[46] – and on major open-source projects such as Apache Cassandra and HBase.

The major components of Thrift are:

- **Thrift Interface Definition Language (IDL)** – used to define your services and compose any custom types which your services will send and receive
- **Protocols** – control encoding/decoding elements of data into a common binary format (e.g. thrift binary protocol, or JSON)
- **Transports** – provide a common interface for reading/writing to different media (e.g. TCP socket, pipe, memory buffer)
- **Thrift compiler** – parses Thrift IDL files to generate stub code for the server and client interfaces, and serialization/deserialization code for the custom types defined in IDL
- **Server implementation** – handles accepting connections, reading requests from those connections, dispatching calls to an object that implements the interface, and sending the responses back to clients
- **Client implementation** – translates method calls into requests and sends them to the server

17.1.2 Improving the State of Java Thrift using Netty

The Apache distribution of Thrift has been ported to about twenty different languages, and there are also separate frameworks compatible with Thrift built for other languages (Twitter's Finagle for Scala is a great example). Several of these languages receive at least some usage at Facebook, but the most common used for writing Thrift services here at Facebook are C++ and Java.

When I arrived here at Facebook, we were already well underway with the development of a solid, high-performance, asynchronous Thrift implementation in C++, built around libevent. From libevent, we get cross-platform abstractions over the operating system APIs for asynchronous IO, but libevent isn't any easier to use than say, raw Java NIO. So we've also built some abstractions on top of that such as asynchronous message channels, and we make use of chained buffers from folly[47] to avoid copies as much as possible. This framework also has a client implementation that supports asynchronous calls with multiplexing, and a server implementation that supports asynchronous request handling (the server can start an

[45] Find more info at http://thrift.apache.org/
[46] Find more examples at http://thrift.apache.org/about/
[47] Folly is Facebook's open-source C++ common library: https://www.facebook.com/notes/facebook-engineering/folly-the-facebook-open-source-library/10150864656793920

asynchronous task to handle a request and return immediately, then invoke a callback or set a future later when the response is ready).

Meanwhile, our Java thrift framework received a lot less attention, and our load testing tools showed that Java performance lagged well behind C++. There were already Java Thrift frameworks built on NIO, and asynchronous NIO-based clients were available as well. But the clients did not support pipelining or multiplexing requests, and the servers did not support asynchronous request handling. Because of these missing features, Java Thrift service developers here at Facebook were running into problems that had been solved already in C++, and it became a source of frustration.

We could have built a similar custom framework on top of NIO and based our new Java Thrift implementation on that, as we had done for C++. But experience showed us that this was a *ton* of work to get right, and as it just so happened, the framework we needed was already out there, just waiting for us to make use of it: Netty.

We quickly put together a server implementation, and mashed the names of "Netty" and "Thrift" together to come up with "Nifty", the name for the new server. It was immediately impressive how less code was needed to get Nifty working, compared to everything we needed to achieve the same results in C++.

Next we put together a simple load tester Thrift server using Nifty, and used our load-testing tools to compare it to existing servers. The results were clear: Nifty simply clobbered the other NIO servers, and is in the same ballpark as our newest C++ Thrift server. Using Netty was definitely going to pay off!

17.1.3 Nifty Server Design

Nifty[48] is an open-source, Apache-licensed Thrift client/server implementation built on top of the Apache Thrift library. It is designed so that moving from any other Java Thrift server implementation should be painless: you can reuse the same Thrift IDL files, the same Thrift code generator (packaged with the Apache Thrift library), and the same service interface implementation, the only thing that really needs to change is your server startup code (Nifty setup follows a slightly different style than that of the traditional Thrift server implementations in Apache Thrift).

NIFTY ENCODER/DECODER

The default Nifty server handles either plain messages or framed messages (with a four-byte size prefix). It does this by using a custom Netty frame decoder that looks at the first few bytes to determine how to decode the rest. Then, when a complete message is found, the decoder wraps the message content along with a field that indicates the type of message. The server later refers to this field to encode the response in the same format.

[48] https://github.com/facebook/nifty

Nifty also supports plugging in your own custom codec. For example, we use a custom codec internally to read Facebook-specific header content, which we insert before each message (containing optional metadata, client capabilities, etc). The decoder could also easily be extended to handle other types of message transports, such as HTTP.

ORDERING RESPONSES ON THE SERVER

Initial versions of Java Thrift used OIO sockets, and servers maintained one thread per active connection. With this setup, each request is read, processed, and the response is sent all on the same thread, before the next response is read. This guarantees that responses will always be returned in the same order as corresponding requests arrived.

Newer asynchronous I/O server implementations were built which don't need a thread-per-connection, and these servers can handle more simultaneous connections, but clients still mainly used synchronous I/O, so the server could count on not receiving the next request until after it had sent the current response.

This request / execution flow is shown in Figure 17.1.

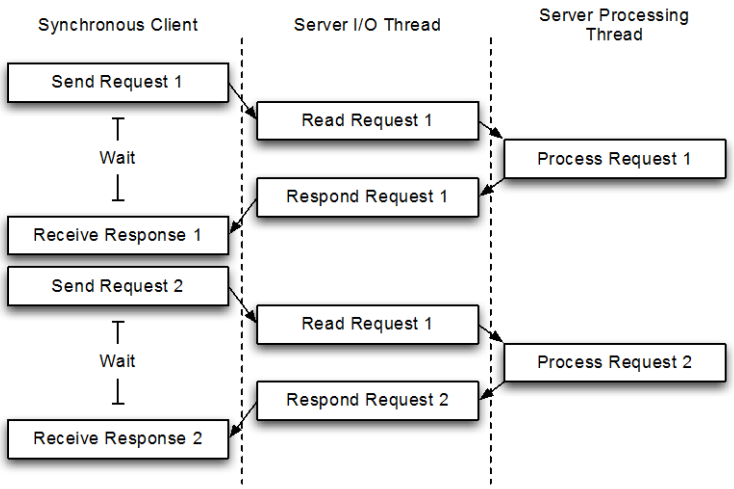


Figure 17.1 Request – Response flow

Initial pseudo-asynchronous usages of clients started happening when a few Thrift users took advantage of the fact that for method a generated client method foo(), send_foo() and recv_foo() were also exposed separately. This allows Thrift users to send several requests (whether on several clients, or on the same client) and then call the corresponding "receive" methods to start waiting for and collecting the results.

In this new scenario, the server may read multiple requests from a single client before it has finished processing the first. In an ideal world, we could assume all asynchronous thrift

clients that pipeline requests can handle the responses to those requests in whatever order they arrive. In the world we live in though, newer clients *can* handler this, but older asynchronous Thrift clients may write multiple requests, but must receive the responses in order.

This is the kind of problem solved by using the Netty 4 EventExecutor or OrderedMemoryAwareThreadPoolExcecutor in Netty 3.x, which guarantee sequential processing for all incoming messages on a connection, without forcing all of those messages to run on the same executor thread.

Figure 17.2 shows how pipelined requests are handled in the correct order which means the response for the first request will be returned and then the response for the second and so on...

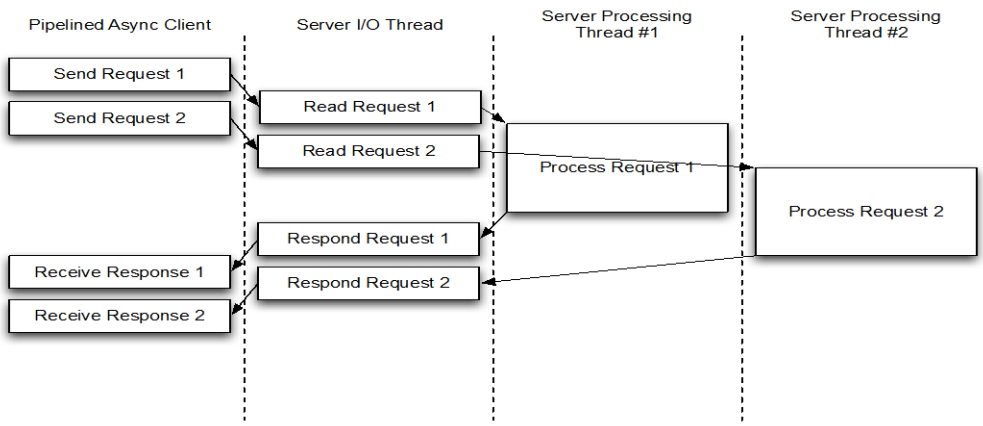


Figure 17.2 Request – Response flow

Nifty has special requirements though: we aim to serve each client with the best response ordering that it can handle. We would like to allow the handlers for multiple pipelined requests from a single connection to be processing in parallel, but then we can't control the order in which these handlers will finish.

So we instead use a solution that involves buffering responses: if the client requires in-order responses, we'll buffer later responses until all the earlier ones are also available, and then we'll send them together, in the required order. See Figure 17.3.

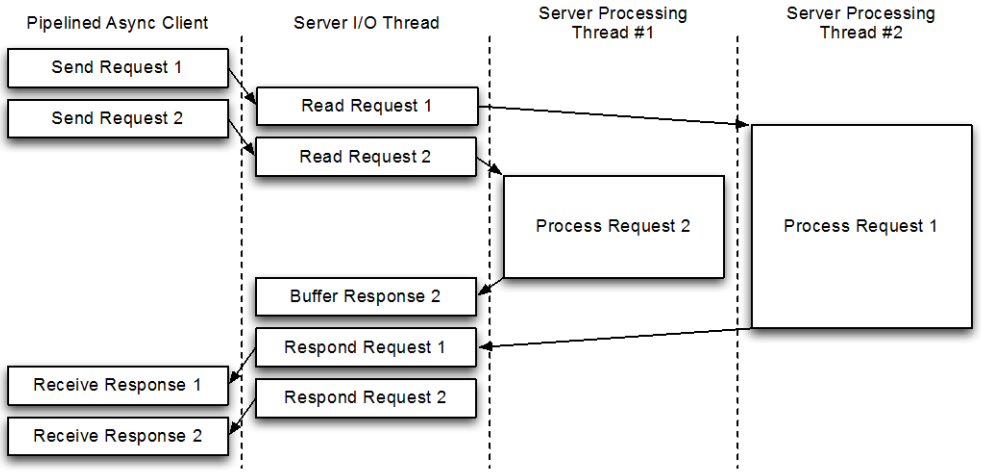


Figure 17.3 Request – Response flow

Of course, Nifty includes asynchronous channels (usable through Swift) that *do* support out-of-order responses. When using a custom transport that allows the client to notify the server of this client capability, the server is relieved of the burden of buffering responses, and will send them back in whatever order the requests finish.

17.1.4 Nifty Asynchronous Client Design

Nifty client development is mostly focused on asynchronous clients. Nifty actually does provide a Netty implementation of the Thrift's synchronous transport interface, but its usage is pretty limited because it doesn't provide much win over a standard socket transport from Thrift. Because of this the user should use the asynchronous clients whenever possible.

PIPELINING

The Thrift library has its own NIO-based asynchronous client implementation, but one feature we were missing from them was request pipelining. Pipelining is the ability to send multiple requests on the same connection, without waiting for a response. If the server has idle worker threads, it can process these requests in parallel, but even if all worker threads are busy, pipelining can still help in other ways. The server will spend less time waiting for something to read, and the client may be able to send multiple small requests together in a single TCP packet, thus better utilizing network bandwidth.

With Netty, pipelining just works. Netty does all the hard work of managing the state of the various NIO selection keys, and Nifty can focus on encoding requests and decoding responses.

MULTIPLEXING

As our infrastructure grows, we've started to see a *lot* of connections building up on our servers. Multiplexing – sharing connections for all the Thrift clients connecting from a single source – can help to mitigate this. But multiplexing over a client connection that requires ordered responses presents a problem: one client on the connection may incur extra latency because it's response must come after the responses for other requests sharing the connection.

The basic solution is pretty simple though: Thrift already sends a sequence identifier with every message, so to support out-of-order responses we just need the client channels to keep a map from sequence id to response handler, instead of a queue.

The catch is that in standard synchronous Thrift clients, the protocol is responsible for extracting the sequence identifier from the message, and the protocol calls the transport, but never the other way around.

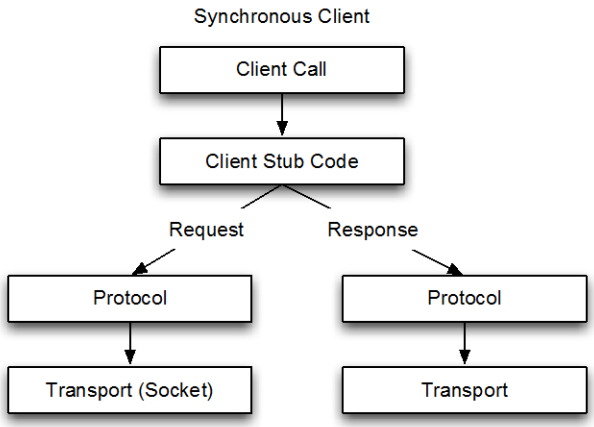


Figure 17.4 Multiplexing / Transport layers

That simple flow (shown above in Figure 17.4) works fine for a synchronous client, where the protocol can wait on the transport to actually receive the response, but for an asynchronous client, the control flow gets a bit more complicated. The client call is dispatched to the Swift library, which first asks the protocol to encode the request into a buffer, then passes that encoded request buffer to the Nifty channel to be written out. When the channel receives a response from the server, it notifies the Swift library, which again uses the protocol to decode the response buffer. This is the flow shown in Figure 17.5.

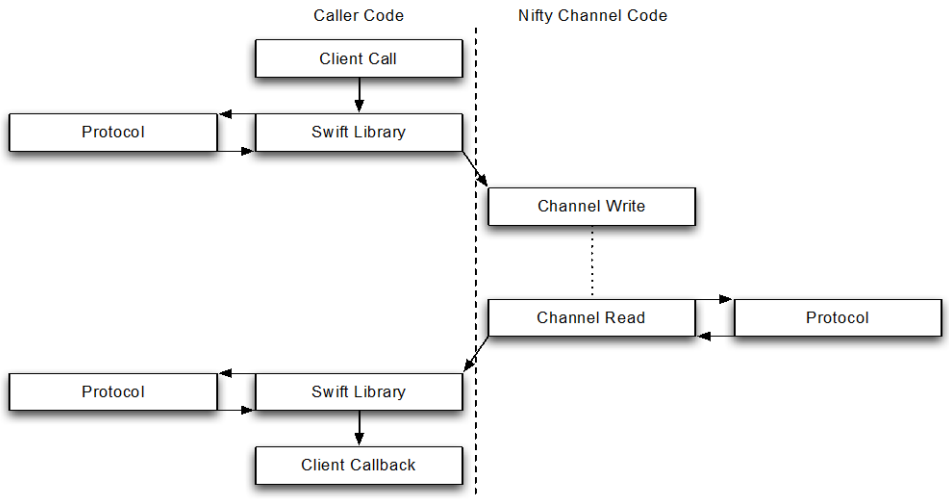


Figure 17.5 Dispatching

17.1.5 Swift: A faster way to build Java Thrift service

The other key part of our new Java Thrift framework is called Swift. It uses Nifty as its IO engine, but the service specifications can be represented directly in Java using annotations, giving Thrift service developers the ability to work purely in Java. When your service starts up, the Swift runtime gathers information about all the services and types through a combination of reflection and interpreting Swift annotations. From that information, it can build the same kind of model that the Thrfit compiler builds when parsing Thrift IDL files. Then, it uses this model to run the server and client directly (without any generated server or client stub code) by generating new classes from byte code used for serializing/deserializing the custom types.

Skipping the normal Thrift code generation also makes it easier to add new features without having to change the IDL compiler, so a lot of our new features (e.g. asynchronous clients) are supported in Swift first. If you're interested, take a look at the introductory information on Swift's github page[49].

17.1.6 Results

Performance Comparisons

One measurement of Thrift server performance is a benchmark of "no-ops". This benchmark uses long-running clients that continually make Thrift calls to a server that just sends an

[49] https://github.com/facebook/swift

empty response back. This measurement is not a realistic performance estimation of most actual Thrift services, however it is a good measure of the maximum potential of a Thrift service built, and improving this benchmark does generally mean a reduction in the amount of CPU used by the framework itself.

Table 17.1 Benchmark of different implementations

Thrift Server Implementation	No-op Requests/Second
`TNonblockingServer`	~68000
`TThreadedSelectorServer`	188000
`TThreadPoolServer`	867000
`DirectServer`	367000
`Nifty`	963000
`Previous libevent-based C++ Server`	895000
`Next-gen libevent-based C++ Server`	1150000

As shown above, Nifty outperforms all of the other NIO Thrift server implementations (TNonblockingServer, TThreadedSelectorServer, and THshaServer) on this benchmark. It even easily beats DirectServer (a pre-Nifty server implementation we used internally, based on plain NIO).

The only Java server we tested that can compete with Nifty is TThreadPoolServer. This server uses raw OIO, and runs each connection on a dedicated thread. This gives it an edge when handling a lower number of connections; however, you can easily run into scaling problems with OIO when your server needs to handle a very large number of simultaneous connections.

Nifty even beats the previous C++ server implementation that was most prominent at the time we started development on it, and while it falls a bit short compared to our next-gen C++ server framework, it is at least in the same ballpark.

EXAMPLE STABILITY ISSUES

Before Nifty, many of our major Java services at Facebook used a custom NIO-based Thrift server implementation that works similarly to Nifty. That implementation is an older codebase that had more time to mature, but because it's asynchronous I/O handling code was built from scratch, and because Nifty is built on the solid foundation of Netty's asynchronous I/O framework, it has had a lot fewer problems.

One of our custom message queuing services had been built using DirectServer, and started to suffer from a kind of socket leak. A lot of connections were sitting around in CLOSE_WAIT state, meaning the server had received a notification that the client closed the

socket, but the server never reciprocated by making its own call to close the socket. This leaves the socket in CLOSE_WAIT limbo.

The problem happened very slowly: across the entire pool of machines handling this service, there might be millions of requests per second, but usually only one socket on one server would enter this state in an hour. This meant that it wasn't an urgent issue since it took a long time before a server needed a restart at that rate, but it also complicated tracking down the cause. Extensive digging through the code didn't help much either: initially several places looked suspicious, but everything eventually checked out and we didn't locate the problem.

Eventually, we migrated the service onto Nifty. The conversion – including testing in a staging environment – took less than a day and the problem has since disappeared, and we haven't really seen any problems like this in Nifty.

This is just one example of the kind of subtle bug that can show up when using NIO directly, and it's similar to bugs we've had to solve in our C++ Thrift framework time and time again to stabilize it. But I think it's a great example of how using Netty has helped us take advantage of the years of stability fixes it has received.

IMPROVING TIMEOUT HANDLING FOR C++

Netty has also helped us indirectly, by lending suggestions for improvements to our C++ framework. An example of this is the hashed wheel timer. Our C++ framework uses timeout events from libevent to drive client and server timeouts, however adding separate timeouts for every request proves to be prohibitively expensive, so we had been using what we called "timeout sets". The idea here was that a client connection to a particular service usually has the same receive timeout for every call made from that client, so we'd maintain only one real timer event for a set of timeouts that share the same duration. Every new timeout was guaranteed to fire after existing timeouts scheduled for that set, so when each timeout expired or was cancelled, we would schedule only the next timeout.

However, our users occasionally wanted to supply per-call timeouts, with different timeout values for different requests on the same connection. In this scenario, the benefits of using a timeout set are lost, so we tried using individual timer events. We started to see performance problems when many timeouts were scheduled at once. We knew that Nifty doesn't run into this problem, despite the fact that it does not use timeout sets. Netty solves this problem with its HashedWheelTimer[50]. So, with inspiration from Netty, we put together a hashed wheel timer for our C++ Thrift framework as well, and it has resolved the performance issue with variable per-request timeouts.

[50] http://netty.io/4.0/api/io/netty/util/HashedWheelTimer.html

FUTURE IMPROVEMENTS ON NETTY 4

Nifty is currently running on Netty 3 which has been great for us so far, but we have a Netty 4 port ready which we'll be moving to very soon now that the v4 has been finalized. We are eagerly looking forward to some of the benefits the Netty 4 API will offer us.

One example of how we plan to make better use of v4 is achieving better control over which thread manages a given connection. We hope to use this feature to allow server handler methods to start asynchronous client calls from the same IO thread the server call is running on. This is something that specialized C++ servers are already able to take advantage of (for example a Thrift request router).

Extending from that example, we also look forward to being able to build better client connection pools that are able to migrate existing pooled connections to the desired IO worker thread, which is something that wasn't possible in v3.

17.1.7 Summary

With the help of Netty, we've been able to build a better Java server framework that nearly matches the performance of our fastest C++ Thrift server framework. We've migrated several of our existing major Java services onto Nifty already, solving some pesky stability and performance problems, and we've even started to feed back some ideas from Netty, and from the development of Nifty and Swift, into improving aspects of C++ Thrift.

On top of that Netty has been a pleasure to work with, and made a lot of new features like built-in SOCKS support for Thrift clients simple to add.

But we're not done yet. We've got plenty of performance tuning work to do, as well as plenty of other improvements planned for the future. So if you're interested in Thrift development using Java, be sure to keep an eye out!

17.2 Netty at Twitter: Finagle

Written by Jeff Smick; Software Engineer at Twitter

Finagle is Twitter's fault tolerant, protocol-agnostic RPC framework built atop Netty. All of the core services that make up Twitter's architecture are built on Finagle, from backends serving user information, tweets, and timelines to front end API endpoints handling HTTP requests.

17.2.1 Twitter's Growing Pains

Twitter was originally built as a monolithic Ruby on Rails application, semi-affectionately called "The Monorail." As Twitter started to experience massive growth, the Ruby runtime and Rails framework started to become a bottleneck. From a compute standpoint Ruby was relatively inefficient with resources. From a development standpoint The Monorail was becoming difficult to maintain. Modifications to code in one area would opaquely affect another area. Ownership of different aspects of the code was unclear. Small changes unrelated to core business objects required a full deploy. Core business objects didn't expose clear APIs, which increased the brittleness of internal structures and the likelihood of incidents.!

We decided to split the monorail into distinct services with clear owners and clear APIs allowing for faster iteration and easier maintenance. Each core business object would be maintained by a specific team and be served by its own service. There was precedent within the company for developing on the JVM, a few core services had already been moved out of the monorail and had been rebuilt in Scala. Our operations teams had a background in JVM services and knew how to operationalize them. Given that, we decided to build all new services on the JVM using either Java or Scala. Most services decided on Scala as their JVM language of choice.

17.2.2 The birth of Finagle

In order to build out this new architecture we needed a performant, fault tolerant, protocol agnostic, asynchronous RPC framework. Within a service oriented architecture, services spend most of their time waiting for responses from other upstream services. Using an asynchronous library allows services to concurrently process requests and take full advantage of the hardware. While Finagle could have been built directly on top of NIO, Netty had already solved many of the problems we would have encountered as well as provided a clean, clear API.

Twitter is built atop several open source protocols, primarily HTTP, Thrift, Memcached, MySQL, and Redis. Our network stack would need to be flexible enough that it could speak any of these protocols and extensible enough that we could easily add more. Netty isn't tied to any particular protocol. Adding to it is as simple as creating the appropriate ChannelHandlers. This extensibility has lead to many community driven protocol implementations including, SPDY[51], PostrgreSQL[52], WebSockets[53], IRC[54], and AWS[55].

Netty's connection management and protocol agnosticism provided an excellent base from which Finagle could be built. However we had a few other requirements Netty couldn't satisfy out of the box as those requirements are more „high-level". Clients need to connect to and load balance across a cluster of servers. All services need to export metrics (request rates, latencies, etc) that provide valuable insight for debugging service behavior. With a service oriented architecture a single request may go through dozens of services making debugging performance issues nearly impossible without a Dapper[56] inspired tracing framework[57]. Finagle was built to solve these problems.

[51] https://github.com/twitter/finagle/tree/master/finagle-spdy
[52] https://github.com/mairbek/finagle-postgres
[53] https://github.com/sprsquish/finagle-websocket
[54] https://github.com/sprsquish/finagle-irc
[55] https://github.com/sclasen/finagle-aws
[56] http://research.google.com/pubs/pub36356.html
[57] https://github.com/twitter/zipkin

17.2.3 How Finagle works

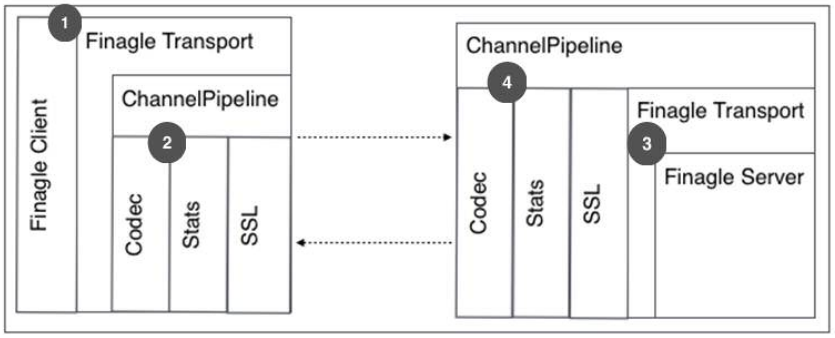
Internally Finagle is very modular. Components are written independently then stacked together. Each component can be swapped in or out depending on the provided configuration. For instance tracers all implement the same interface, thus a tracer can be created to send tracing data to a local file, hold it in memory and expose a read endpoint, or write out to the network.

At the bottom of a Finagle stack is a Transport. Transports are a representation of a stream of objects that can be asynchronously read from and written to. Transports are implemented as Netty ChannelHandlers and inserted into the end of a ChannelPipeline. Messages come in from the wire where Netty picks them up, runs them through the ChannelPipeline where they're interpreted by a codec then sent to the Finagle Transport. From there Finagle reads the message off the Transport and sends it through its own stack.

For client connections Finagle maintains a pool of transports that it can load balance across. Depending on the semantics of the provided connection pool Finagle will either request a new connection from Netty or re-use an existing one. When a new connection is requested a Netty ChannelPipeline is created based on the client's codec. Extra ChannelHandlers are added to the ChannelPipeline for stats, logging, and SSL. The connection is then handed to a channel transport which Finagle can write to and read from.

On the server side a Netty server is created then given a ChannelPipelineFactory that manages the codec, stats, timeouts, and logging. The last ChannelHandler in a server's ChannelPipeline is a Finagle bridge. The bridge will watch for new incoming connections and create a new Transport for each one. The Transport wraps the new channel before it's handed to a server implementation. Messages are then read out of the ChannelPipeline and sent to the implemented server instance.

Figure 17.6 show the relationship for the Finagle Client and Server.



#1 Finagle Client which is powered by the Finagle Transport. This Transport abstract Netty away from the user

#2 The actual ChannelPipeline of Netty that contains all the ChannelHandler implementations that do the actual work

#3 Finagle Server which is created for each connection and provided a transport to read from and write to

#4 The actual ChannelPipeline of Netty that contains all the ChannelHandler implementations that do the actual work

Figure 17.6 Netty usage

NETTY / FINAGLE BRIDGE

Listing 17.1 shows a static `ChannelFactory` with default options.

Listing 17.1 Setup the ChannelFactory

```
object Netty3Transporter {
    val channelFactory: ChannelFactory =
        new NioClientSocketChannelFactory(
            Executor, 1 /*# boss threads*/, WorkerPool, DefaultTimer
        ){
             // no-op; unreleasable
             override def releaseExternalResources() = ()
        }                                                              #1
    val defaultChannelOptions: Map[String, Object] = Map(
        "tcpNoDelay" -> java.lang.Boolean.TRUE,
        "reuseAddress" -> java.lang.Boolean.TRUE                       #2
      )
}
```

#1 Create a new ChannelFactory instance
#2 Set options which are used for new Channels.

This Bridges a netty channel with a Finagle transport (stats code has been removed here for brevity). When invoked via `apply` this will create a new `Channel` and Transport. A Future is returned that is fulfilled when the Channel has either connected or failed to connect.

Listing 17.2 shows the `ChannelConnector` which allows to connect a Channel to a remote host.

Listing 17.2 Connect to remote host

```
private[netty3] class ChannelConnector[In, Out](
    newChannel: () => Channel,
    newTransport: Channel => Transport[In, Out]
) extends (SocketAddress => Future[Transport[In, Out]]) {
    def apply(addr: SocketAddress): Future[Transport[In, Out]] = {
        require(addr != null)
        val ch = try newChannel() catch {
            case NonFatal(exc) => return Future.exception(exc)         #1
        }
        // Transport is now bound to the channel; this is done prior to
        // it being connected so we don't lose any messages.
        val transport = newTransport(ch)                               #2
        val connectFuture = ch.connect(addr)                           #3
        val promise = new Promise[Transport[In, Out]]                  #4
        promise setInterruptHandler { case _cause =>
           // Propagate cancellations onto the netty future.
           connectFuture.cancel()
        }
        connectFuture.addListener(new ChannelFutureListener {
```

```
            def operationComplete(f: ChannelFuture) {                    #5
                if (f.isSuccess) {
                    promise.setValue(transport)
                } else if (f.isCancelled) {
                    promise.setException(
                    WriteException(new CancelledConnectionException))
                } else {
                    promise.setException(WriteException(f.getCause))
                }
            }
        })
        promise onFailure { _ =>
            Channels.close(ch)
        }
    }
}
```

#1 Try to create a new Channel. If it fails wrap the exception in a Future and return immediately
#2 Create a new Transport with the Channel
#3 Connect the remote host asynchronously. The returned ChannelFuture will be notified once the connection completes
#4 Create a new Promise which will be notified once the connect attempt finished
#5 Handle the completion of the connectFuture by fulfilling the created promise

This factory is provided a `ChannelPipelineFactory`, a channel factory and transport factory. The factory is invoked via the `apply` method. Once invoked a new ChannelPipeline is created (`newPipeline`). That pipeline is used by the `ChannelFactory` to create a new Channel which is then configured with the provided options (`newConfiguredChannel`). The configured channel is passed to a ChannelConnector as an anonymous factory. The connector is invoked and `Future[Transport]` is returned.

Listing 17.3 shows the details.[58]

Listing 17.3 Netty3 based Transport

```
case class Netty3Transporter[In, Out](
    pipelineFactory: ChannelPipelineFactory,
    newChannel: ChannelPipeline => Channel =
        Netty3Transporter.channelFactory.newChannel(_),
    newTransport: Channel => Transport[In, Out] =
        new ChannelTransport[In, Out](_),
    // various timeout/ssl options
) extends (
    (SocketAddress, StatsReceiver) => Future[Transport[In, Out]]
){
    private def newPipeline(
        addr: SocketAddress,
        statsReceiver: StatsReceiver
    )={
        val pipeline = pipelineFactory.getPipeline()
        // add stats, timeouts, and ssl handlers
```

[58] https://github.com/twitter/finagle/blob/master/finagle-core/src/main/scala/com/twitter/finagle/netty3/client.scala

```
        pipeline                                                          #1
    }
    private def newConfiguredChannel(
        addr: SocketAddress,
        statsReceiver: StatsReceiver
    )={
        val ch = newChannel(newPipeline(addr, statsReceiver))
        ch.getConfig.setOptions(channelOptions.asJava)
        ch
    }
    def apply(
        addr: SocketAddress,
        statsReceiver: StatsReceiver
    ): Future[Transport[In, Out]] = {
        val conn = new ChannelConnector[In, Out](
            () => newConfiguredChannel(addr, statsReceiver),
            newTransport, statsReceiver)                                  #2
        conn(addr)
    }
}
```

#1 Create a new ChannelPipeline and add the needed handlers
#2 Create a new ChannelConnector which is used internally

Finagle servers use Listeners to bind themselves to a given address. In this case the listener is provided a ChannelPipelineFactory, a ChannelFactory, and various options (excluded here for brevity). Listen is invoked with an address to bind to and a Transport to communicate over. A Netty ServerBootstrap is created and configured. Then an anonymous ServerBridge factory is created and passed to a ChannelPipelineFactory which is given to the bootstrapped server. Finally the server is bound to the given address.

Listing 17.4 shows the Netty bases implementation of a the `Listener`.

Listing 17.4 Netty based Listener

```
case class Netty3Listener[In, Out](
    pipelineFactory: ChannelPipelineFactory,
    channelFactory: ServerChannelFactory
    bootstrapOptions: Map[String, Object], ... // stats/timeouts/ssl config
) extends Listener[In, Out] {
    def newServerPipelineFactory(
        statsReceiver: StatsReceiver, newBridge: () => ChannelHandler
    ) = new ChannelPipelineFactory {                                      #1
        def getPipeline() = {
            val pipeline = pipelineFactory.getPipeline()
            ... // add stats/timeouts/ssl
            pipeline.addLast("finagleBridge", newBridge())                #2
            pipeline
        }
    }
    def listen(addr: SocketAddress)(
        serveTransport: Transport[In, Out] => Unit
    ): ListeningServer =
        new ListeningServer with CloseAwaitably {
            val newBridge = () => new ServerBridge(serveTransport, ...)
            val bootstrap = new ServerBootstrap(channelFactory)
            bootstrap.setOptions(bootstrapOptions.asJava)
```

```
            bootstrap.setPipelineFactory(
                newServerPipelineFactory(scopedStatsReceiver, newBridge))
            val ch = bootstrap.bind(addr)
        }
}   }
```

#1 Create a new ChannelPipelineFactory
#2 Add the Bridge into the ChannelPipeline

When a new channel is opened this bridge creates a new ChannelTransport and hands it back to the Finagle server. Listing 17.5 shows the code needed.[59]

Listing 17.5 Bridge Netty and Finagle

```
class ServerBridge[In, Out](
    serveTransport: Transport[In, Out] => Unit,
) extends SimpleChannelHandler {
    override def channelOpen(
        ctx: ChannelHandlerContext,
        e: ChannelStateEvent
    ){
        val channel = e.getChannel
        val transport = new ChannelTransport[In, Out](channel)           #1
        serveTransport(transport)
        super.channelOpen(ctx, e)
    }
    override def exceptionCaught(
        ctx: ChannelHandlerContext,
        e: ExceptionEvent
    ) { // log exception and close channel }
}
```

#1 Is called once a new Channel was open and create a new ChannelTransport to bridge to Finagle

17.3 Finagle's Abstraction

Finagle's core concept is a simple function (functional programming is the key here) from Request to Future of Response.

```
type Service[Req, Rep] = Req => Future[Rep]
```

This simplicity allows for very powerful composition. Service is a symmetric API representing both the client and the server. Servers implement the service interface. The server can be used concretely for testing or Finagle can expose it on a network interface. Clients are provided an implemented service that is either virtual or a concrete representation of a remote server.

For example, we can create a simple HTTP server by implementing a service that takes an HttpReq and returns a Future[HttpRep] representing an eventual response.

[59] https://github.com/twitter/finagle/blob/master/finagle-core/src/main/scala/com/twitter/finagle/netty3/server.scala

```
val s: Service[HttpReq, HttpRep] = new Service[HttpReq, HttpRep] {
    def apply(req: HttpReq): Future[HttpRep] =
        Future.value(HttpRep(Status.OK, req.body))
}
Http.serve(":80", s)
```

A client is then provided a symmetric representation of that service.

```
val client: Service[HttpReq, HttpRep] =  Http.newService("twitter.com:80")
val f: Future[HttpRep] = client(HttpReq("/"))
f map { rep => processResponse(rep) }
```

This example exposes the server on port 80 of all interfaces and consumes from twitter.com port 80. However we can also choose not to expose the server and instead use it directly.

```
server(HttpReq("/")) map { rep => processResponse(rep) }
```

Here the client code behaves the same way but doesn't require a network connection. This makes testing clients and servers very simple and straightforward.

Clients and servers provide application specific functionality. However there is a need for application agnostic functionality as well. Timeouts, authentication, and statics are a few examples. Filters provide an abstraction for implementing application agnostic functionality.

Filters receive a request and a service with which it is composed.

```
type Filter[Req, Rep] = (Req, Service[Req, Rep]) => Future[Rep]
```

Filters can be chained together before being applied to a service.

```
recordHandletime andThen
traceRequest andThen
collectJvmStats andThen
myService
```

This allows for clean abstractions of logic and good separation of concerns. Internally, Finagle heavily uses filters. Filters help to enhance modularity and reusability. They've proved valuable for testing as they can be unit tested in isolation with minimal mocking.

Filters can modify both the data and type of requests and responses. Figure 17.7 shows a request making its way through a filter chain into a service and back out.

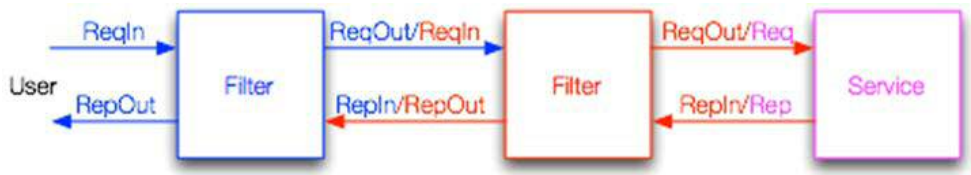


Figure 17.7 Reqest / Response flow

We might use type modification for implementing authentication.

```
val auth: Filter[HttpReq, AuthHttpReq, HttpRes, HttpRes] =
    { (req, svc) => authReq(req) flatMap { authReq => svc(authReq) } }

val authedService: Service[AuthHttpReq, HttpRes] = ...
val service: Service[HttpReq, HttpRes] =
    auth andThen authedService
```

Here we have a service that requires and AuthHttpReq. To satisfy the requirement a filter is created that can receive an HttpReq and authenticate it. The filter is then composed with the service yielding a new service that can take an HttpReq and produce an HttpRes. This allows us to test the authenticating filter in isolation of the service.

17.3.1 Failure Management

We operate in an environment of failure; hardware will fail, networks will become congested, network links fail. Libraries capable of extremely high throughput and extremely low latency are meaningless if the systems they're running on or are communicating with fail. To that end, Finagle is setup to manage failures in principled way. It trades some throughput and latency for better failure management.

Finagle can balance load across a cluster of hosts implicitly using latency as a heuristic. Finagle clients locally track load on every host it knows about. It does so by counting the number of outstanding requests being dispatched to a single host. Given that, Finagle will dispatch new requests to hosts with the lowest load and implicitly the lowest latency.

Failed requests will cause Finagle to close the connection to the failing host and remove it from the load balancer. In the background Finagle will continuously try to reconnect. The host will be re-added to the load balancer only after Finagle can re-establish a connection. Service owners are then free to shutdown individual hosts without negatively impacting downstream clients.

17.3.2 Composing Services

Finagle's service as a function philosophy allows for simple, but expressive code. For example a user making a request for their home timeline touches a number of services. The core of which are the authentication service, timeline service, and tweet service. These relationships can be expressed succinctly.

Listing 17.6 Composing services via Finagle

```
val timelineSvc = Thrift.newIface[TimelineService](...)                  #1
val tweetSvc = Thrift.newIface[TweetService](...)
val authSvc = Thrift.newIface[AuthService](...)

val authFilter = Filter.mk[Req, AuthReq, Res, Res] { (req, svc) =>       #2
    authSvc.authenticate(req) flatMap svc(_)
}

val apiService = Service.mk[AuthReq, Res] { req =>
```

```
        timelineSvc(req.userId) flatMap {tl =>
            val tweets = tl map tweetSvc.getById(_)
            Future.collect(tweets) map tweetsToJson(_)
        }
    }                                                                    #3

    Http.serve(":80", authFilter andThen apiService)                     #4
```

#1 Create a client for each service
#2 Create new Filter to authenticate incoming requests
#3 Create a service to convert an authenticated timeline request to a json response
#4 Start a new HTTP server on port 80 using the authenticating filter and our service

Here we create clients for the timeline service, tweet service, and authentication service. A filter is created for authenticating raw requests. Finally our service is implemented, combined with the auth filter and exposed on port 80.

When a request is received the auth filter will attempt to authenticate it. A failure will be returned immediately without ever affecting the core service. Upon successful authentication the AuthReq will be sent to the api service. The service will use the attached userId to lookup the user's timeline via the timeline service. A list of tweet ids is returned then iterated over. Each id is then used to request the associated tweet. Finally the list of tweet requests is collected and converted into a JSON response.

As you can see, the flow of data is defined and we leave the concurrency to Finagle. We don't have to manage thread pools or worry about race conditions. The code is clear and safe.

17.4 The Future: Netty

We've been working closely with the Netty maintainers to improve on parts of Netty that both Finagle and the wider community can benefit from[60]. Recently the internal structure of Finagle has been updated to be more modular, paving the way for an upgrade to Netty 4.

17.4.1 Conclusion

Finagle has yielded excellent results. We've managed to dramatically increase the amount of traffic we can serve while reducing latencies and hardware requirements. For instance after moving our API endpoints from the Ruby stack on to Finagle we saw p99 latencies drop from hundreds of milliseconds to tens while reducing the number of machines required from triple digits to single digits. Our new stack has enabled us to reach new records in throughput. As of this writing our record tweets per second is 143,199[61]. That number would have been unthinkable on our old architecture.

Finagle was born out of a need to set Twitter up to scale out to billions of users across the entire globe at a time when keeping the service up for just a few million was a daunting task.

[60] https://blog.twitter.com/2013/netty-4-at-twitter-reduced-gc-overhead
[61] https://blog.twitter.com/2013/new-tweets-per-second-record-and-how

Using Netty as a base we were able quickly design and build Finagle to manage our scaling challenges. Finagle and Netty handle every request Twitter sees.

17.5 Summary

In this chapter you got some insight about how big companies like Facebook and Twitter build software on top of Netty which are not only mission-critical but also need to give them the right amount of performance and flexibility.

Facebooks Nifty project gives insight how they was able to replace an existing Thrift implementation with their own which is written on top of Netty. For this they wrote their own protocol encoders and decoders by making use of Netty's features.

Twitters Finagle shows how you can build your own framework on top of Netty and offer more „high-level" features like load-balancing, failover and more. All of thise while still gives a high-degree of performane by making use of Netty in it's core.

Hopefully this chapter will serve you as source of insperation but also as source of informations which you will be able to make use of in your next-gen project.

This chapter should give you an idea how you can build your next generation framework on top of Netty and make use of it's performance and flexibility. No need to handle all the low-level networking API by yourself when using Netty.