Netty IN ACTION

Norman Maurer
Marvin Allen Wolfthal

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Netty in Action
Version 10

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 0.

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 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.

[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.

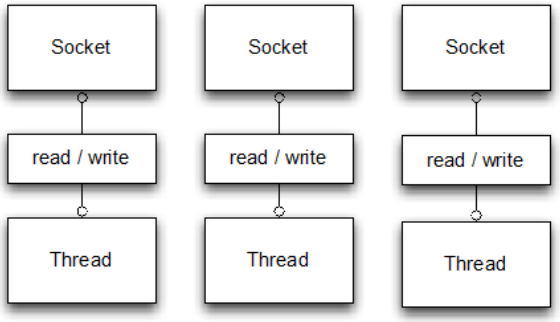


Figure 1.1 Blocking I/O

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.

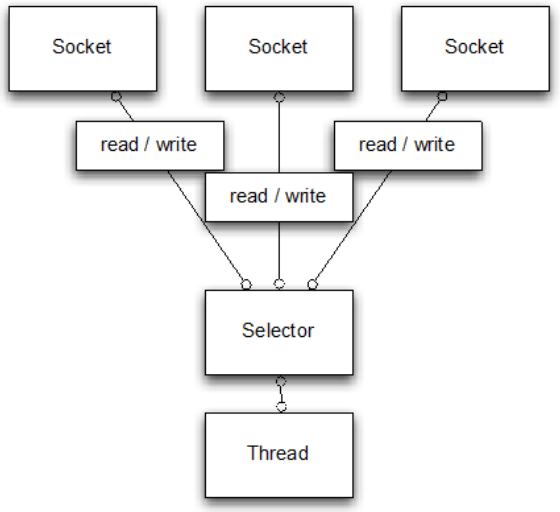


Figure 1.2 Nonblocking I/O

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

[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

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 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.

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

---

[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.

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 1.2.4). Once such a callback is triggered the event can be handled by an implementation of `interface ChannelHandler`. 0 shows such a `ChannelHandler`. Once a new connection has been established the callback `channelActive()` will be called and will print out a message.

Listing 1.2 ChannelHandler triggered by a callback

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

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

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."

0 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.

Listing 1.3 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

0 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.

Listing 1.4 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.

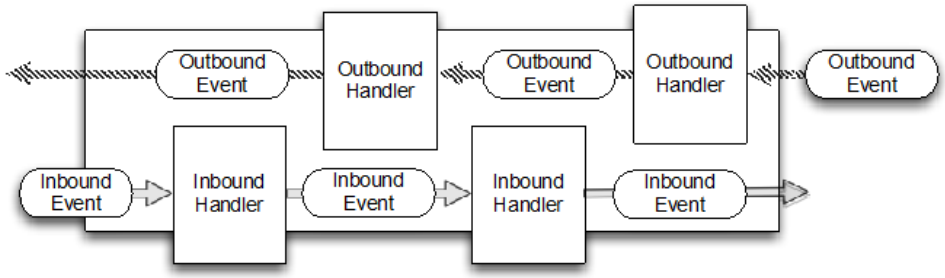


Figure 1.3 Event Flow

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://maven.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.

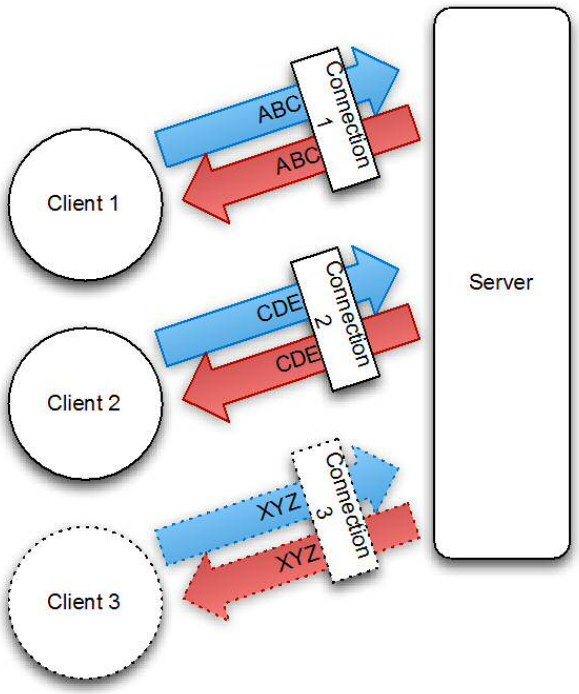


Figure 2.1.Echo client / server

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.writeAndFlush(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.

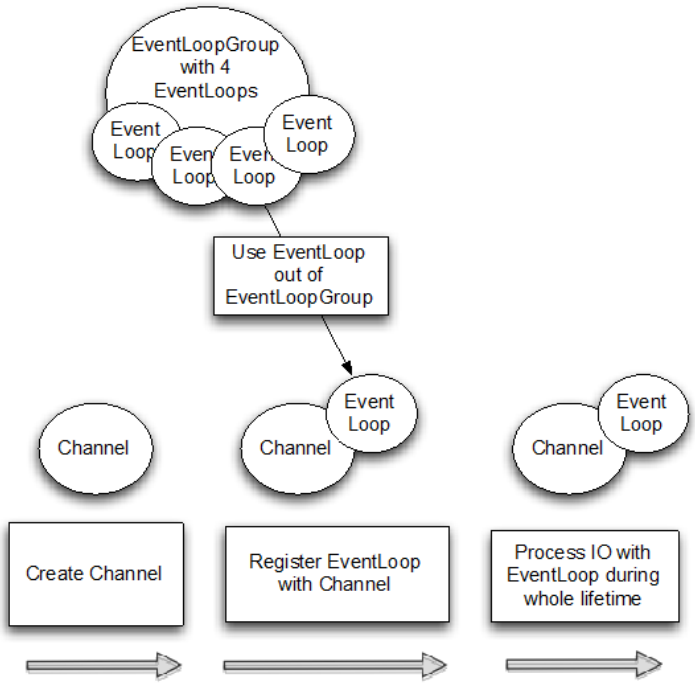


Figure 3.1

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.

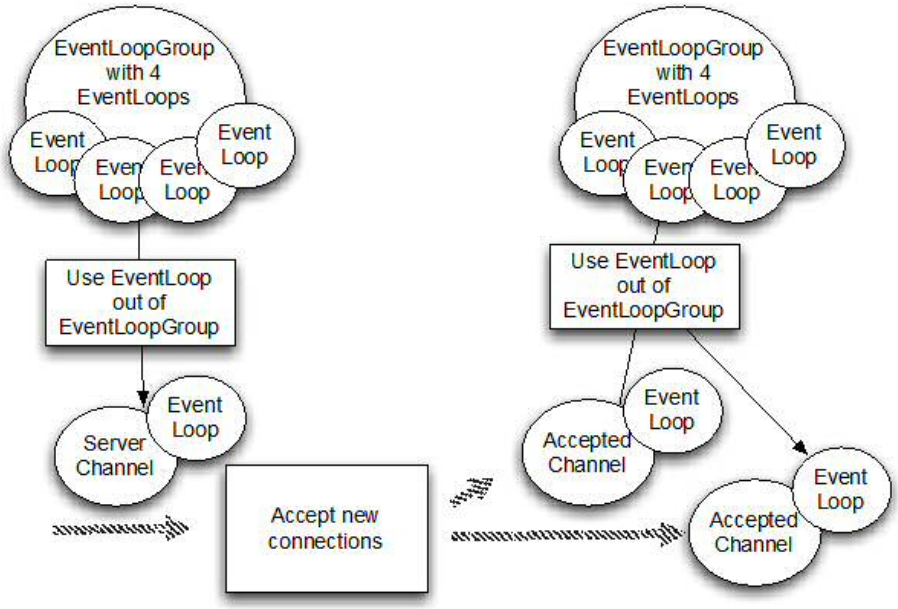


Figure 3.2 Server with two EventLoopGroups

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`.

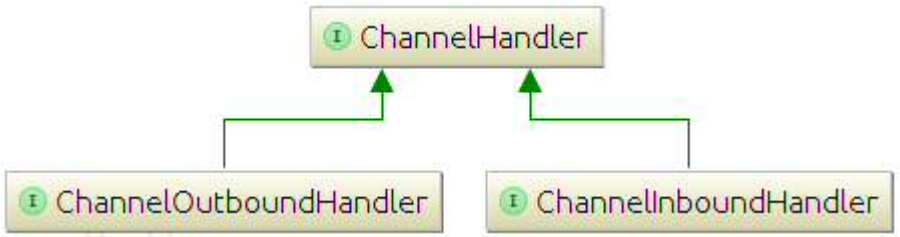


Figure 3.3 ChannelHandler class hierarchy

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`.

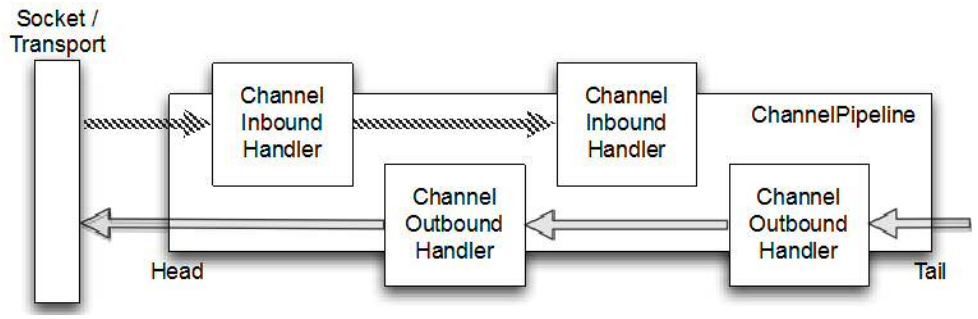


Figure 3.4 ChannelPipeline with inbound and outbound ChannelHandlers

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 a variety of transports, their use cases and APIs:

- NIO
- OIO
- Local
- Embedded

Network applications provide a communications channel for people and systems, but of course they also move a lot of data from one place to another. How this is done concretely depends largely on the network transport, but what gets transferred is always the same: bytes over the wire. The concept of a transport helps us to abstract away the underlying mechanics of data transfer. All users need to know is that bytes are sent and received.

If you've ever done network programming in Java you may have discovered at some point that you had to support many more concurrent connections than expected. If you then tried to switch from a blocking to a non-blocking transport, you may have encountered problems because the network APIs exposed by Java to handle the two cases are quite different.

Netty layers a unified API over its transport implementations, making such a conversion far simpler than you can achieve with the JDK. The resulting code code will be uncontaminated by implementation dependencies and you won't need to perform extensive refactoring of your entire code base. In short, you can spend your time doing something productive.

In this chapter we'll study this unified API, contrasting it with the JDK and demonstrating its far greater ease of use. We'll explain the different transport implementations that are bundled with Netty and the use cases appropriate to each. After absorbing this information, you'll be able to choose the best option for your application

The only prerequisite for this chapter is knowledge of the Java programming language. Experience with network frameworks or network programming is a plus but not a requirement.

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, we'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 off we'll present blocking (OIO) and asynchronous (NIO) versions of the application that use only the JDK APIs. Listing 4.1 shows the blocking implementation. If you've ever experienced the joy of network programming with Java this code will be very familiar.

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 {
                for (;;) {
                        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 the server to the specified port.**
2. **Accept a connection.**
3. **Create a new `Thread` to handle the connection.**
4. **Write message to the connected client.**
5. **Close the connection once the message is written and flushed.**
6. **Start the `Thread`.**

This code works fine and writes “Hi!” to the remote peer. After a while you may notice that it isn't scaling very well to the tens of thousands of concurrent incoming connections so you decide to convert to asynchronous networking. You soon discover that the asynchronous API is completely different, obliging you to rewrite your application completely.

The non-blocking version is shown in Listing 4.2.

Listing 4.2 Asynchronous networking without Netty

```
public class PlainNioServer {
      public void serve(int port) throws IOException {
```

```
        ServerSocketChannel serverChannel = ServerSocketChannel.open();
        serverChannel.configureBlocking(false);
        ServerSocket ss = serverChannel.socket();
        InetSocketAddress address = new InetSocketAddress(port);
        ss.bind(address);                                                    //1
        Selector selector = Selector.open();                                 //2
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);            //3
        final ByteBuffer msg = ByteBuffer.wrap("Hi!\r\n".getBytes());
        for (;;) {
                try {
                        selector.select();                                   //4
                } catch (IOException ex) {
                        ex.printStackTrace();
                        // handle exception
                        break;
                }
                Set<SelectionKey> readyKeys = selector.selectedKeys();       //5
                Iterator<SelectionKey> 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();
                                client.configureBlocking(false);

                                client.register(selector, SelectionKey.OP_WRITE |
                                        SelectionKey.OP_READ, msg.duplicate()); //7
                                System.out.println(
                                        "Accepted connection from " + client);
                                }
                                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) {
                                        // ignore on close
                                }
                        }
                }
        }
      }
}
```

1. **Bind the server to the selected port.**
2. **Open the selector for handling channels.**

3. **Register the `ServerSocket` with the `Selector` and specify that it is interested in accepting connections.**
4. **Wait for new events to process. This will block until an event is incoming.**
5. **Obtain all `SelectionKey` instances that received events.**
6. **Check if the event is a new connection ready to be accepted.**
7. **Accept client and register it with the selector.**
8. **Check if the socket is ready for writing data.**
9. **Write data to the connected client. If the network is saturated, this loop will write data when the connection is writable, until the buffer is empty.**
10. **Close the connection.**

As you can see, this code is completely different from the preceding version, although it does exactly the same thing. If reimplementing this simple application for non-blocking I/O requires a complete rewrite, consider the level of effort that would be required to port something truly complex.

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

4.1.2 Using I/O and NIO with Netty

We'll start by writing another blocking version of the application, but this time using the Netty framework, as shown in Listing 4.3.

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 a `ServerBootstrap`.
2. Use OioEventLoopGroup to 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

Next we will implement the same logic with non-blocking I/O using Netty.

4.1.3 Non-blocking Netty version

The code in the following listing is virtually identical to that in Listing 4.3. A change of two lines (highlighted) is all that is required to switch from OIO (blocking) transport to NIO (non-blocking).

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 ChannelInboundHandlerAdapter() {              //4
                        @Override
                        public void channelActive(
                           ChannelHandlerContext ctx) throws Exception {
                              ctx.writeAndFlush(buf.duplicate())       //5
                                 .addListener(
                                    ChannelFutureListener.CLOSE);
                        }
                     });
               }
            });
         ChannelFuture f = b.bind().sync();                            //6
         f.channel().closeFuture().sync();
      } finally {
         group.shutdownGracefully().sync();                            //7
      }
   }
}
```

1. Create ServerBootstrap.
2. Use `NioEventLoopGroup` forl nonblocking mode.
3. Specify `ChannelInitializer` to be called for each accepted connection.
4. Add `ChannelStateHandlerAdapter` to receive events and process them.

5. **Write message to client and add `ChannelFutureListener` to close the connection once the message is written.**
6. **Bind server to accept connections.**
7. **Release all resources.**

Since Netty exposes the same API for every transport implementation, your code is practically unchanged, whichever you choose. In all cases the implementation is defined in terms of the interfaces `Channel`, `ChannelPipeline` and `ChannelHandler`.

Now that you've seen some of the benefits of using Netty-based transports, let's take a closer look at the transport API itself.

4.2 Transport API

At the heart of the transport API is the `Channel` interface, which is used for all outbound operations. The hierarchy of this interface is shown in Figure 4.1.

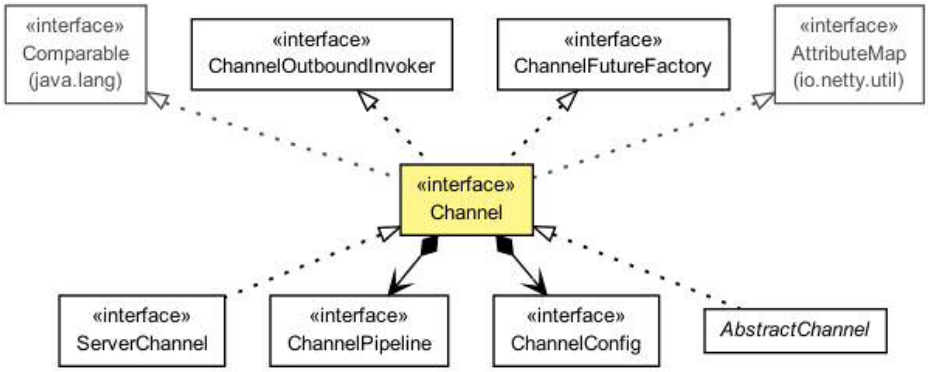


Figure 4.1 Channel interface hierarchy

The figure shows that a `Channel` has a `ChannelPipeline` and a `ChannelConfig` assigned to it. The `ChannelConfig` holds all of the configuration settings for the `Channel` and allows for updating them on the fly. Since a transport may have configuration settings that are valid only for that implementation, the transport may implement a subtype of `ChannelConfig`. (You can find detailed information in the javadocs of the specific `ChannelConfig` implementation.)

The `ChannelPipeline` holds all of the `ChannelHandler` instances that will be applied to inbound and outbound data and events. These `ChannelHandler`s implement the application's logic both for reacting to state changes and data transformation.

Typical uses for `ChannelHandlers`:

- Transform data from one format to another
- Notification of exceptions
- Notification of a `Channel` becoming active or inactive
- Notification when a `Channel` is registered/deregistered from an `EventLoop`
- Notification about about user-defined events

Intercepting Filter

The `ChannelPipeline` implements a common design pattern, "Intercepting Filters". UNIX pipes are another example: commands are chained together, the output of one command connecting to the input of the next in line.

You can also modify the `ChannelPipeline` on the fly by adding or removing `ChannelHandler` instances as needed. This capability of Netty can be exploited to build highly flexible applications. For example, you could support the STARTTLS protocol on demand simply by adding an appropriate `ChannelHandler` (the `SslHandler`) to the `ChannelPipeline` when the protocol was requested.

In addition to accessing the assigned `ChannelPipeline` and `ChannelConfig`, you can make use of the `Channel` itself. The most important methods provided by the `Channel` API are listed in Table 4.1.

Table 4.1 Channel main 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 true if the `Channel` is active. The meaning of "active" may depend on the underlying transport. For example, a `Socket` transport is active once connected to the remote peer, while a `Datagram` transport would be active once it is open.
`localAddress()`	Returns the local `SocketAddress`
`remoteAddress()`	Returns the remote `SocketAddress`
`write(…)`	Writes data to the remote peer. This data is passed to the `ChannelPipeline` and queued until it is flushed.
`flush()`	Flushes the previously written data to the underlying transport, for example a `Socket`.
`writeAndFlush(…)`	A convenience method for calling `write(…)` followed by `flush()`

Further on we'll discuss the uses of all these features in detail. For now, keep in mind that the broad range of functionality offered by Netty always relies on a small number of interfaces. This means that you can make significant modifcations to application logic without wholesale refactoring of your code base.

For example, let's look at the common task of writing data and flushing it to the remote peer. Listing 4.5 illustrates the use of `Channel.writeAndFlush(...)` for this purpose.

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 the data to write**
2. **Write the data and flush it**
3. **Add `ChannelFutureListener` to receive notification after write completes**
4. **Write operation completes without error**
5. **Write operation completed with error**

All the methods of `Channel` are safe to use in a multithreaded enviroment. Thus it is safe to store a reference to a `Channel` in your application and use it whenever the need arises to write something to the remote peer, even when many threads are in use. Listing 4.6 shows a simple example of writing with multiple threads. Note that the messages are guaranteed to be written in the order in which the write method is invoked.

Listing 4.6 Using the channel from many threads

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

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

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

1. **Create a `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 one thread**
5. **Hand over another write task to executor for execution in another thread**

4.3 Included transports

Netty comes bundled with a several transports that are ready for use. Not all of them support all protocols. This means the transport you select needs to be compatible with underlying protocol employed by your application. In this section we'll discuss the relationships between transports and protocols.

Table 4.1 shows all of the transports that are provided by Netty.

Table 4.1 Provided transports

Name	Package	Description
NIO	`io.netty.channel.socket.nio`	Uses the `java.nio.channels` package as a foundation - uses a selector-based approach.
OIO	`io.netty.channel.socket.oio`	Uses the `java.net.package` as a foundation - 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 true network- based Transport. This can be quite useful for testing your `ChannelHandler` implementations.

Let's start by examining the most widely-used transport implementation, NIO.

4.3.1 NIO – Nonblocking I/O

NIO provides a fully asynchronous implementation of all I/O operations. At it's base is the selector approach that has been available in since the NIO subsystem was introduced in JDK 1.4.

The basic concept of the selector is to register to receive notification when the state of a `Channel` changes. The possible state changes are:

- A new `Channel` was accepted and is ready.
- A `Channel` connection was completed.
- A `Channel` has data that is ready for reading.
- A `Channel` is available for writing data.

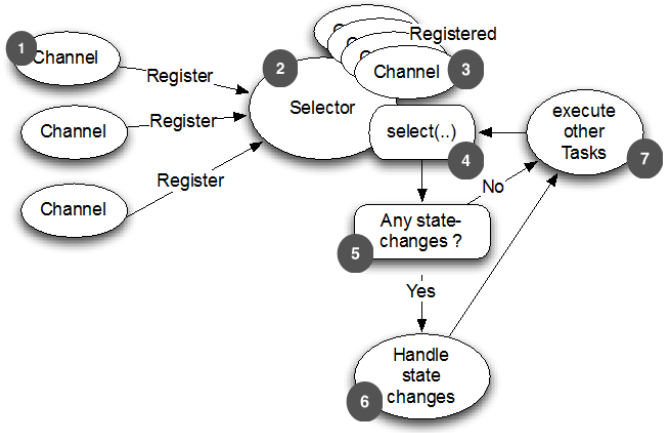
The application can then react to these state changes, after which the selector is reset to be notified of the next change of state. This is done with a thread that checks for updates and, if there are any, dispatches them accordingly.

The values shown in Table 4.2 are the bit patterms defined by the `SelectionKey` class. These patterns are combined to define the set of state changes about which the application desires to be notified.

Table 4.2 Selection operation bit-set

Name	Description
`OP_ACCEPT`	Requests notification when a new connection is accepted and a `Channel` is created.
`OP_CONNECT`	Requests notification when a connection is established.
`OP_READ`	Requests notification when data are ready to be read from the `Channel`.
`OP_WRITE`	Requests notification when it is possible to write more data to the `Channel`. Most of the time this is possible, but at times the socket buffer is completely filled. This usually happens when you write data faster then the remote peer can handle it.

While Netty's NIO transport uses this model internally to receive and send data, the API exposed to the user completely hides the internal implementation. (As mentioned previously, this API is common to all transports.) Figure 4.2 shows the process flow.



1. **New channel that registers WITH selector**
2. **Selector handles notification of state changes**
3. **Previously registered channels**
4. **The Selector.select() method blocks until new state changes are received or a configured timeout has elapsed**
5. **Check if there were state changes**
6. **Handle all state changes**
7. **Execute other tasks in the same thread in which selector operates**

Figure 4.2 Selecting and Processing State Changes

A feature that is currently available only with NIO transport is called "zero-file-copy", which allows you to quickly and efficiently transfer content from a file system by moving data to the network stack without copying from kernel space to user space. This can make a big difference in protocols such as FTP or HTTP.

However, not all operating systems support this feature. Furthermore, you can't use it with file systems that implement data encryption or compression - only transferring the raw content of a file is supported. On the other hand, transferring files that are already encrypted is perfectly valid.

Next we'll discuss is OIO, which provides a blocking transport.

4.3.2 OIO – Old blocking I/O

In Netty, the OIO transport represents a compromise. It is accessed via Netty's common API but is not asynchronous, being built on the blocking implementation of `java.net`. Anyone following the discussion to this point might think that this protocol doesn't have much to offer. But it does have its valid uses.

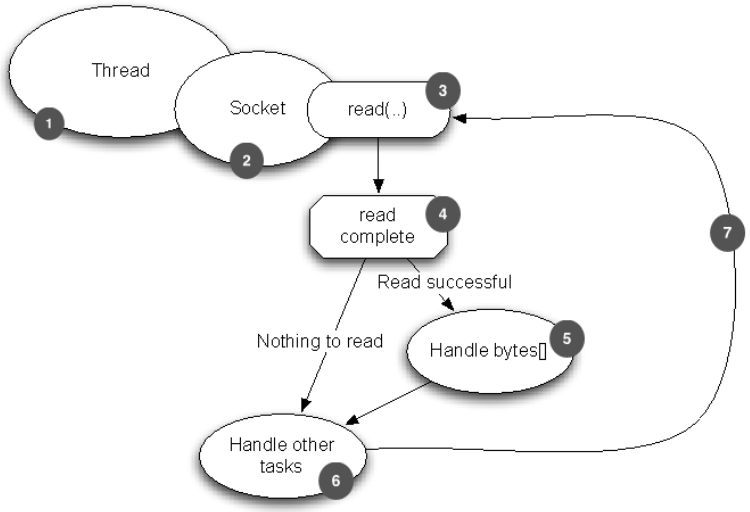
Suppose you need to port some legacy code that uses libraries that do blocking calls (such as JDBC[6]). It may not be feasible to convert the logic to non-blocking. Instead, you could use the OIO transport in the short term and port it later to one of the pure asynchronous transports. Let's see how it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

In the `java.net` API you usually have one thread that accepts new connections arriving at the listening `ServerSocket` and creating a new thread to handle the traffic on the new `Socket`. This is required since every I/O operation on a specific socket may block at any time. Handling multiple sockets with a single thread could easily lead to a blocking operation on one socket tying up all the others as well.

Given this, you may wonder how Netty can support NIO with the same API used for asynchronous transports. Here Netty makes use of the `SO_TIMEOUT` flag that can be set on a `Socket`. This timeout specifies the maximum number of milliseconds to wait for an I/O operation to complete. If the operation fails to complete within the specified interval, a `SocketTimeoutException` is thrown. Netty catches this exception and continues the processing loop. On the next `EventLoop` run, it tries again. This is in fact the only way an asynchronous framework like Netty can support OIO. The problem with this approach is the cost of filling in the `stack trace when a SocketTimeoutException` is thrown.

Figure 4.3 illustrates the logic we have described.

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



1. **Thread is allocated to Socket**
2. **Socket connected to remote peer**
3. **Read operation (may block)**
4. **Read complete**
5. **Handle readable bytes**
6. **Execute other tasks submitted belonging on socket**
7. **Try to read again**

Figure 4.3 OIO-Processing logic

Our discussion up to here covers the two most frequently used transports in Netty. In the next sections we'll study the others.

4.3.3 Local Transport for Communication within a JVM

Netty provides a "local" transport for asynchronous communication between clients and servers running in the same Java Virtual Machine. This transport supports the API common to all Netty transport implementations.

In this transport the `SocketAddress` associated with a server `Channel` is not "bound" to a physical network address, rather it is stored in a registry for as long as the server is running and deregistered when the `Channel` is closed. Since the transport does not accept "real" network traffic, it can't interoperate with other transport implementations. Therefore, a client that wishes to connect to a server that uses local transport must use it as well. Apart from this limitation, its use is identical to that of other transports.

4.3.4 Embedded Transport

Netty also provides a transport that allows you to embed `ChannelHandler` instances in other `ChannelHandlers` and use them like helper classes, adding flexibility to the ways in which you can interact with your `ChannelHandler`s.

This embedding technique is typically used for testing `ChannelHandler` implementations, but it can also be employed to add functionality to an existing `ChannelHandler` without requiring code changes. The key to the embedded transport is a concrete `Channel` implementation not surprisingly called "`EmbeddedChannel`".

Chapter 10 describes the use of `EmbeddedChannel` to test `ChannelHandler`s.

4.4 Transport Use Cases

Now that we've seen all the transports in detail, let's consider the factors that go into choosing one over another. As mentioned previously, not all transports support all core protocols, which may limit your choices. Table 4.3 shows the matrix of transports and protocols.

Table 4.3 Transport support by network protocol

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

* Currently supported only on Linux. The table reflects the protocols supported at the time of publication.

Enabling SCTP on Linux

Note that SCTP requires kernel support as well as installation of the user libraries.

Example: for Ubuntu, use the following command:

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

For Fedora, use `yum`:

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

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

While only SCTP[7] has these special requirements, there are configuration recommendations for specific transports. Consider also that a server platform will probably need to support a higher number of concurrent connections than that of a client.

Here are the use cases that you are likely to encounter.

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

NON-BLOCKING CODE-BASE

If you don't have blocking calls in your code base - or can limit them - it is always a good idea to use NIO. While NIO is intended to handle many concurrent connections it also works out quite well with a smaller number, especially given the way it shares threads across connections.

BLOCKING CODE BASE

If your code base relies heavily on blocking I/O and your applications have a corresponding design, you are likely to encounter problems with blocking operations if you try to convert directly to Netty's NIO transport. Rather than rewriting your code to accomplish this, consider a phased migration: start with with OIO and move to NIO once you have revised your code.

COMMUNICATIONS WITHIN THE SAME JVM

Communications within the same JVM with no need to expose a service over the network present the perfect use case for local transport. This will eliminate all the overhead of real network operations while still employing your Netty code base.

If the need arises later to expose the service over the network you will need only to replace the transport with NIO or OIO. You may also find it useful to add an extra encoder and decoder to convert Java objects to and from `ByteBuf`.

TESTING YOUR CHANNELHANDLER IMPLEMENTATIONS

If you want to write unit tests for your `ChannelHandler` implementations, consider using the embedded transport. This will make it easy to test your code without having to create many mock objects. Your classes will still conform to the common API event flow, guaranteeing that the `ChannelHandler` will work correctly with "live" transports. You will find more information about testing `ChannelHandler`s in Chapter 10.

Table 4.4 summarizes the use cases we have examined.

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 `ChannelHandler` implementations	Embedded

4.5 Summary

In this chapter we studied transports, their implemention and use, and how Netty presents them to the developer.

We went through the transports that ship with Netty and explained their behavior. We also looked at their minimum requirements, as not all transports work with the same Java version or may be usable only on specific operating systems. Finally, we examined matching transports to specific use cases.

In the next chapter we will focus on `ByteBuf` and `ByteBufHolder`, Netty's data containers. We'll show how to use them and how to get the best performance from them.

5

Buffers

This chapter covers

- Byte containers and allocators

As we pointed out earlier, the fundamental unit of network data is always the `byte`. Java NIO provides the `ByteBuffer` as its byte container, but this class is overly complex and somewhat cumbersome to use.

Netty's alternative to `ByteBuffer` is `ByteBuf`,a powerful implementation that addresses the limitations of the JDK API and provides a better tool for network application developers. But `ByteBuf` doesn't merely expose methods for manipulating a sequence of bytes; it was specifically designed with the semantics of Netty's `ChannelPipeline` in mind.

In this chapter, we'll illustrate the superior functionality and flexibility of `ByteBuf` as compared to the JDK API. This will also give us a better understanding of Netty's approach to data handling. Finally, the discussion will lay the foundation for later chapters, since `ByteBuf` is a fundamental element of the Netty framework, along with `ChannelPipeline` and `ChannelHandler`, classes we'll be examining in detail in Chapter 6.

5.1 Buffer API

Netty's buffer API is exposed through:

- `ByteBuf`
- `ByteBufHolder`

Netty uses reference-counting[8] to determine when to release a `ByteBuf` or `ByteBufHolder` and any associated resources, thereby enabling the use of pooling and other techniques to improve performance and reduce memory consumption. This feature doesn't require any specific action on the part of the developer; we mention it to highlight the fact that it is a good practice to release pooled resources as early as possible and especially when using `ByteBuf` and `ByteBufHolder`.

These are some of the advantages of the `ByteBuf` API:

- It is extensible to user-defined buffer types.
- Transparent zero-copy is achieved by a built-in composite buffer type.
- Capacity is expanded on demand, as with the JDK `StringBuilder`.
- There is no need to call `flip()` to switch between reader and writer modes.
- Reader and writer employ distinct indices.
- Supports method chaining.
- Supports reference counting.
- Supports pooling.

[8] See section 5.6.

We'll explore these features after we have examined `ByteBuf` and `ByteBufHolder` in more detail.

5.2 ByteBuf - The byte data container

Since all network communications involve the movement of sequences of bytes, an efficient, convenient, and easy-to-use data structure is clearly a necessity. Netty's `ByteBuf` implementation meets - and exceeds - these requirements.

`ByteBuf` provides simple access to its data, facilitated by the use of distinct indices for reading and writing. For example, you can read the same sequence of bytes repeatedly either by adjusting the reader index and then rereading or by using one of the `get()` methods that accept an index argument.

5.2.1 How it works

When you write to a `ByteBuf` its `writerIndex` is incremented by the corresponding number of bytes. Similarly, when you read from it the `readerIndex` is incremented. If you read bytes until both indices are at the same position, the `ByteBuf` becomes unreadable. Subsequent read operations will trigger the same `IndexOutOfBoundsException` you encounter when you attempt to access data beyond the end of an array.

Calling any of the `ByteBuf` methods whose names begin with `"read"` or `"write"` will advance the corresponding index. On the other hand, operations that `"set"` and `"get"` bytes do *not* move the indices; they operate on the relative index that that is passed as an argument to the method.

The maximum capacity of a `ByteBuf` can be specified, this limiting its capacity. Attempting to move the write index past this value will trigger an exception. (The default size limit is `Integer.MAX_VALUE`.)

Figure 5.1 shows the layout and state of an empty `ByteBuf`.

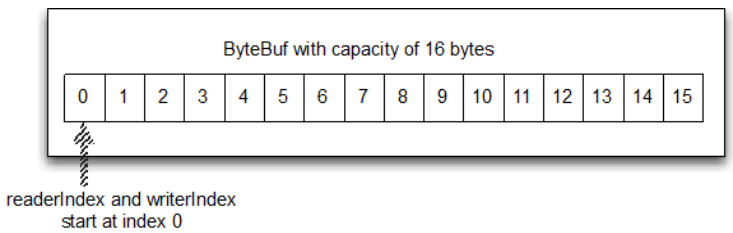


Figure 5.1 A 16-byte ByteBuf with its indices set to 0

5.2.2 ByteBuf Usage Patterns

In working with Netty you will encounter several common patterns built around `ByteBuf`. As we examine them, it will help to keep Figure 5.1 in mind: an array of bytes with indices to control read and write access.

HEAP BUFFERS

The most frequently used `ByteBuf` pattern stores the data in the heap space of the JVM. Referred to as a "backing array," this provides fast allocation and deallocation in situations where pooling is not in use. This approach, shown in Listing 5.1, is well-suited to cases where you have to handle legacy data.

Listing 5.1 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.someMethod(array, offset, length);                    //5
}
```

1. **Check whether `ByteBuf` has a backing array.**
2. **If so get a reference to the array.**
3. **Calculate the offset of the first byte.**
4. **Get the number of readable bytes.**
5. **Call your method using array, offset and length as parameters.**

Notes:

- Attempting to access a backing array when `hasArray()` returns `false` will trigger an `UnsupportedOperationException`.
- This pattern is similar to those that use the JDK's `ByteBuffer`.

DIRECT BUFFERS

"Direct buffer" is another `ByteBuf` pattern. All memory allocation of objects takes place on the heap, right? Well, not always. The `ByteBuffer` class that was introduced with NIO in JDK 1.4 allows a JVM implementation to allocate memory via native calls, the aim being

> to avoid copying the buffer's content to (or from) an intermediate buffer before (or after) each invocation of one of the underlying operating system's native I/O operations.
>
> . . .
>
> The contents of direct buffers may reside outside of the normal garbage-collected heap[9].

This explains why "direct buffers" are ideal for data transfer over a socket. If your data are contained in a heap-allocated buffer, the JVM will in fact copy your buffer to a direct buffer internally before sending it over the socket.

[9] http://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html.

The main disadvantage of direct buffers is that they are somewhat more expensive to allocate and release than heap-based buffers. Another drawback may surface if you are working with legacy code; since the data are not on the heap you may have to make a copy, as shown in Listing 5.2.

Listing 5.2 Direct buffer data access

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

}
```

1 Check if `ByteBuf` not backed by an array. If not this is a direct buffer.
2 Get the number of readable bytes.
3 Allocate a new array to hold `length` bytes.
4 Copy bytes into the array.
5 Call some method with array, offset and length parameters.

Clearly, this involves a bit more work than using a backing array. Therefore, if you know in advance that the data in the container will be accessed as an array you may prefer to use heap memory.

COMPOSITE BUFFERS

The third and final pattern we'll discuss uses a "composite buffer", which presents an aggregate view of multiple `ByteBuf`s. In this case you can add and delete `ByteBuf` instances as needed, a feature entirely absent from the JDK's `ByteBuffer` implementation.

Netty implements this pattern with a subclass of `ByteBuf`, `CompositeByteBuf`, which is a virtual representation of multiple buffers as a single, merged buffer.

Warning!

Since the `ByteBuf` instances in a `CompositeByteBuf` may include both direct and non-direct allocations, `hasArray()` will always returns `false`.

To illustrate, let's consider a message composed of two parts, header and body, to be transmitted via HTTP protocol. The two parts are produced by different application modules and assembled when the message is sent out. The application has the option of reusing the same message body for multiple messages, in which case a new header is created for each transmission.

Since we don't want to reallocate both buffers for each message, this case is a perfect fit for a `CompositeByteBuf`, which eliminates unnecessary copying while providing the familiar `ByteBuf` API. Figure 5.2 shows the message layout.

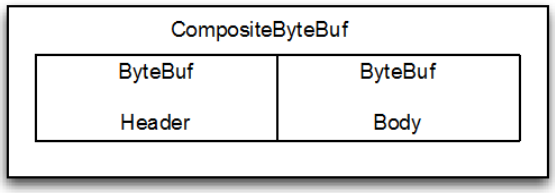


Figure 5.2 `CompositeByteBuf` holding a header and body.

Listing 5.3 shows an implementation using the JDK's `ByteBuffer`. An array of two `ByteBuffers` is created to hold the message components and a third one is created to hold a copy of all the data.

Listing 5.3 Composite buffer pattern using `ByteBuffer`

```
// Use an array to hold the message parts
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();
```

This approach is obviously inefficent; the allocation and copy operations are not optimal and manipulating the array makes the code awkward.

Listing 5.4 shows an improved version using `CompositeByteBuf`.

Listing 5.4 Composite buffer pattern using `CompositeByteBuf`

```
CompositeByteBuf messageBuf = ...;
ByteBuf headerBuf = ...; // can be backing or direct
ByteBuf bodyBuf = ...;   // can be backing or direct

messageBuf.addComponent(headerBuf, bodyBuf);                          //1
.....
messageBuf.removeComponent(0); // replace the header                  //2

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

1. **Append `ByteBuf` instances to the `CompositeByteBuf`.**
2. **Remove `ByteBuf` at index 1 .**
3. **Loop over all the `ByteBuf` instances.**

Listing 5.4 shows that you can simply treat a `CompositeByteBuf` as an iterable collection. Since `CompositeByteBuf` doesn't allow access to a backing array, data access, as shown in Listing 5.5, resembles the direct buffer pattern.

Listing 5.5 Access data

```
CompositeByteBuf compBuf = ...;
int length = compBuf.readableBytes();                                    //1
byte[] array = new byte[length];                                         //2
compBuf.getBytes(array);                                                 //3
yourImpl.someMethod(array, 0, array.length);                             //4
```

1. Get the number of readable bytes.
2. Allocate a new array with length of readable bytes.
3. Read bytes into the array.
4. Use the array with offset and length parameters.

Netty tries to optimize socket I/O operations that employ `CompositeByteBuf`, eliminating when possible the performance and memory usage penalties that are incurred with the JDK's buffer implementation[10]. While this optimization is performed in the Netty core code and is therefore not exposed, one should be aware of its impact.

CompositeByteBuf API

`CompositeByteBuf` offers a great deal of added functionality beyond the methods it inherits from `ByteBuf`. Please refer to the Netty Javadocs for a full listing of the `API`.

5.3 Byte-level Operations

Beyond the basic read and write operations, `ByteBuf` provides numerous methods for modifying the data it contains. In the next sections we'll discuss the most important of these.

5.3.1 Random access indexing

Just as in an ordinary Java byte array, `ByteBuf` indexing is zero-based: the index of the first byte is 0 and that of the last byte is always capacity – 1. Listing 5.7 shows that the encapsulation of its storage mechanisms by `ByteBuf` makes it very simple to iterate over its contents.

Listing 5.7 Access data

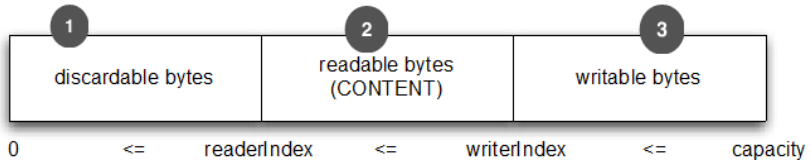
```
ByteBuf buffer = ...;
 for (int i = 0; i < buffer.capacity(); i++) {
     byte b = buffer.getByte(i);
     System.out.println((char) b);
}
```

[10] This applies particularly to the technique known as "Scatter/Gather I/O", defined as "a method of input and output where a single system call writes to a vector of buffers from a single data stream, or, alternatively, reads into a vector of buffers from a single data stream." Love, Robert. *Linux system programming*. Beijing: O'Reilly, 2007.

Note that accessing the contained data using one of the methods that takes an index argument does not alter the value of either `readerIndex` or `writerIndex`. Either can moved manually if necesssary by calling `readerIndex(index)` or `writerIndex(index)`.

5.3.2 Sequential access indexing

`While ByteBuf` has both reader and writer indices, the JDK's `ByteBuffer` uses only one. This is why you have to call `flip()` to switch between read and write modes. Figure 5.3 shows how its two indices partition a `ByteBuf` into three areas.



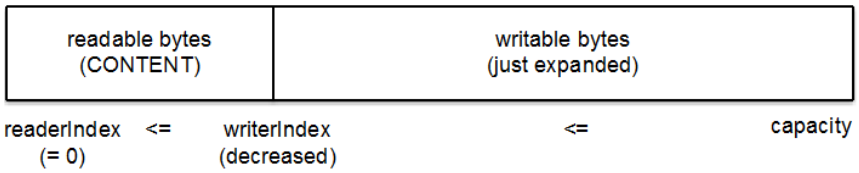
1. **Bytes that can be discarded since they have already been read**
2. **Bytes that have not yet been read: "readable bytes"**
3. **Space where more bytes can be added: "writeable bytes"**

Figure 5.3 ByteBuf internal segmentation

5.3.3 Discardable bytes

The segment labeled "discardable bytes" contains bytes that have already been read. They can be discarded and the space reclaimed by calling `discardReadBytes()`. The initial size of this segment, stored in `readerIndex`, `is 0 and` increases as `"read"` operations are executed (`"get"` operations do not move the `readerIndex`).

Figure 5.4 below shows the result of calling `discardReadBytes()` on the buffer shown in Figure 5.3. You can see that the space in the discardable bytes segment has become available for writing. Note that there is no guarantee about the contents of the writable segment after `discardReadBytes()` has been called.



1. **Bytes that have not yet been read (`readerIndex` is now 0).**
2. **Free space, augmented by the space that was reclaimed.**

Figure 5.4 ByteBuf after discarding read bytes.

While you may be tempted to call `discardReadBytes() frequently to` maximize the writable segment, please be aware that this will most likely cause memory copying, since the readable bytes (marked "CONTENT" in the figures) have to be moved to the start of the

buffer. So it is advisable to use this technique only when there is a clear-cut requirement for it, such as an immediate need to free up memory.

5.3.4 Readable bytes

The "readable bytes" segment of a `ByteBuf` stores the actual data. The default value of a newly allocated, wrapped, or copied buffer's `readerIndex` is 0. Any operation whose name starts with `"read"` or `"skip"` will retrieve or skip the data at the current `readerIndex` and increase it by the number of bytes read.

If the read operation called is one that specifies a `ByteBuf` argument as a write target and does not have a destination index argument, the destination buffer's `writerIndex` will be increased as well. Example:

```
readBytes(ByteBuf dest);
```

If an attempt is made to read from the buffer when readable bytes have have been exhausted, an `IndexOutOfBoundsException` is raised. Listing 5.8 shows how to read all readable bytes.

Listing 5.8 Read all 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 area of memory with undefined contents, ready for writing. The default value of a newly allocated buffer's `writerIndex` is 0. Any operation whose name starts with `"write"` will start writing data at the current `writerIndex`, increasing it by the number of bytes written. If the target of the write operation is also a `ByteBuf` and no source index is specified, the source buffer's `readerIndex` will be increased by the same amount. Example:

```
writeBytes(ByteBuf dest);
```

If an attempt is made to write beyond the target's capacity is an `IndexOutOfBoundException` will be raised.

The following listing shows an example that fills the buffer with random integer values until it runs out of space. The method `writableBytes()` is used here to determine whether there is sufficient space in the buffer.

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 Index management

The JDK `InputStream` defines the methods `mark(int readlimit)` and `reset()`. These are used to mark the current position in the stream and reset the stream to that position, respectively.

Similarly, you can set and reposition the `ByteBuf` `readerIndex` and `writerIndex` by calling `markReaderIndex()`, `markWriterIndex()`, `resetReaderIndex()` and `resetWriterIndex()`. These are similar to the `InputStream` calls, except that there is no `readlimit parameter to specify when the mark becomes invalid.`

You can also move the indices to specified positions by calling `readerIndex(int)` or `writerIndex(int)`. Attempting to set either index to an invalid position will cause an `IndexOutOfBoundsException`.

You can set both `readerIndex` and `writerIndex` to 0 by calling `clear()`. Note that this does not clear the contents of memory. Let's look at how it works. (Figure 5.5 repeats Figure 5.3 above.)

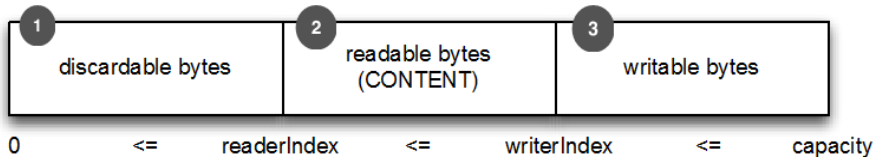
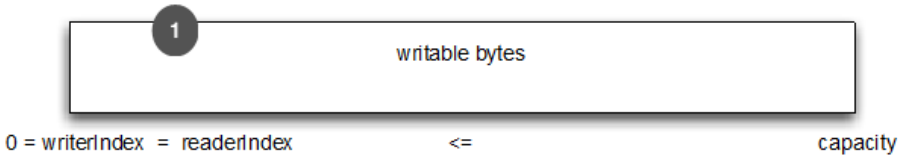


Figure 5.5 Before `clear()` is called

As before, it contains three segments. Figure 5.6 shows the `ByteBuf` after `clear()` is called.



Segment 1 is now as large as the capacity of the ByteBuf, so all the space is writable

Figure 5.6 After `clear()` is called

Calling `clear()` is much less expensive than `discardReadBytes()` because it merely resets the indices and doesn't copy any memory.

5.3.7 Search operations

There are several ways to determine the index of a specified value in the buffer. The simplest cases use the `indexOf()` methods. More complex searches can be executed with methods that take a `ByteBufProcessor` argument. This interface defines a single method, `boolean process(byte value)`, which reports whether or not the input value is the one being sought.

`ByteBufProcessor` defines numerous convenience implementations targeting common values. For example, suppose your application needs to integrate with so-called "Flash sockets",[11] which employ NULL-terminated content. Calling

```
forEachByte( ByteBufProcessor.FIND_NUL )
```

consumes the Flash data with reduced coding effort and increased efficiency, since fewer "bounds checks" are executed during processing.

Listing 5.10 shows an example of searching for a carriage return character, `'\r'`.

Listing 5.10 Using ByteBufProcessor to find '\r'

```
ByteBuf buffer = ...;
int index = buffer.forEachByte(ByteBufProcessor.FIND_CR);
```

5.3.8 Derived buffers

A "derived buffer" is a "view" of a `ByteBuf` that represents its contents in a specialized way. Such views are created by the methods `duplicate()`, `slice()`, `slice(int, int)`, `readOnly()`, and `order(ByteOrder)`. Each of these returns a new `ByteBuf` instance with its own reader, writer and marker indices. However, the internal data storage is shared just as in an NIO `ByteBuffer`. While this makes a derived buffer inexpensive to create, modifying its contents modifies those of the "source" instance as well.

ByteBuf copying

If a fresh copy of an existing buffer is required, use `copy()` or `copy(int, int)`. Unlike a derived buffer, the `ByteBuf` returned by this call has an independent copy of the data.

Use `slice(int, int)` if you need to operate on a segment of the data. Listing 5.11 shows how to work with a slice of a `ByteBuf`.

Listing 5.11 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 a `ByteBuf` which holds bytes for given string.**

[11] http://help.adobe.com/en_US/as3/dev/WSb2ba3b1aad8a27b0-181c51321220efd9d1c-8000.html

2. **Create a new slice of the `ByteBuf` starting at index 0 and ending at index 14.**
3. **This will print "Netty in Action".**
4. **Update the byte at index 0.**
5. **The assertion succeeds since the data are shared and modifications made to one will be visible in the other as well.**

Now let's see how to a *copy* of a `ByteBuf` segment differs from a *slice*.

Listing 5.12 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 a `ByteBuf` which holds bytes for given string.**
2. **Create a copy of a segment of the `ByteBuf` starting at index 0 and ending at index 14.**
3. **This will print "Netty in Action".**
4. **Update the byte at index 0.**
5. **The assertion succeeds since the data are not shared and modifications made to one will not be affect the other.**

The code is almost identical but the effect of a modification on the derived `ByteBuf` is different. Therefore, use a slice whenever possible to avoid the cost of copying memory.

5.3.9 Read/write operations

There are two categories of read/write operations:

- `get()`/`set()` operations that start at a given index and leave it unchanged
- `read()`/`write()` operations that start at a given index and adjust it by the number of bytes accessed.

Table 5.1 lists the most frequently used `get()` methods. For a complete list, please refer to the API docs.

Table 5.1 `get()` operations

Name	Description
`getBoolean(int)`	Return the Boolean value at the given index.
`getByte(int)` `getUnsignedByte(int)`	Return the (unsigned) byte value at the given index.
`getMedium(int)` `getUnsignedMedium(int)`	Return the (unsigned) 24-bit medium value at the given index.

`getInt(int)` `getUnsignedInt(int)`	Return the (unsigned) int value at the given index.
`getLong(int)` `getUnsignedLong(int)`	Return the (unsigned) int value at the given index.
`getShort(int)` `getUnsignedShort(int)`	Return the (unsigned) int value at the given index.
`getBytes(int, ...)`	Return the (unsigned) int value at the given index.

For most of these operations there is a corresponding `set()` method. These are listed in Table 5.2.

Table 5.2 `set()` operations

Name	Description
`setBoolean(int, boolean)`	Set the Boolean value at the given index.
`setByte(int, int)`	Set byte value at the given index.
`setMedium(int, int)`	Set the 24-bit medium value at the given index.
`setInt(int, int)`	Set the int value at the given index.
`setLong(int, long)`	Set the long value at the given index.
`setShort(int, int)`	Set the short value at the given index.

Listing 5.13 illustrates the use of `get()` and `set()` methods.

Listing 5.13 `get()` and `set()` usage

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

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

buf.setByte(0, (byte)'B');                                              //4

System.out.println((char)buf.getByte(0));                               //5
assert readerIndex = buf.readerIndex();                                 //6
assert writerIndex = buf.writerIndex();
```

1. **Create a new `ByteBuf` to hold the bytes for the given `String`**
2. **Print the first char, 'N'**
3. **Store the current `readerIndex` and `writerIndex`**
4. **Update the byte at index 0 with the char 'B'**
5. **Print out the first char , now 'B'.**
6. **These assertions succeed because these operations never modify the indices.**

Now let's examine the `read()` operations, which act on the current `readerIndex` or `writerIndex`. These are used to read from the `ByteBuf` as if it were a stream. (The corresponding `write()` operations are used to "append" to a `ByteBuf`.)

Table 5.3 shows the most commonly used `read()` methods.

Table 5.3 `read()` operations

Name	Description
`readBoolean()`	Reads the Boolean value at the current `readerIndex` and increases the `readerIndex` by 1.
`readByte()` `readUnsignedByte()`	Reads the (unsigned) byte value at the current `readerIndex` and increases the `readerIndex` by 1.
`readMedium()` `readUnsignedMedium()`	Reads the (unsigned) 24-bit medium value at the current `readerIndex` and increases the `readerIndex` by 3.
`readInt()` `readUnsignedInt()`	Reads the (unsigned) int value at the current `readerIndex` and increases the `readerIndex` by 4.
`readLong()` `readUnsignedLong()`	Reads the (unsigned) int value at the current `readerIndex` and increases the `readerIndex` by 8.
`readShort()` `readUnsignedShort()`	Reads the (unsigned) int value at 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.

Almost every `read()` method has a corresponding `write()` method, as shown in Table 5.4. Note that the arguments to these methods are values to be written, not index values!

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.

The following listing shows these methods in use.

Listing 5.14 `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

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

1. **Create a new `ByteBuf` to hold the bytes for the given `String`.**
2. **Print the first char, 'N'.**
3. **Store the current `readerIndex`.**
4. **Store the currebt `writerIndex`.**
5. **Update the byte at index 0 with the char 'B'.**
6. **This assertion succeeds because `writeByte()` at 5. moved the `writerIndex`.**

5.3.10 More operations

Table 5.5 gives a quick look at additional useful operations provided by `ByteBuf`.

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 maximum 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`.

5.4 ByteBufHolder

We often encounter the need to store a variety of property values in addition to the actual data payload. An HTTP response is a good example; along with the content represented as bytes there are status code, cookies, and so on.

Netty provides `ByteBufHolder` to handle this common case. `ByteBufHolder` also provides support for advanced features of Netty such as buffer pooling, where the `ByteBuf` that holds the actual data can be borrowed from a pool and also released automatically if required.

`ByteBufHolder` has just a handful of methods. These support access to the underlying data and reference counting. Table 5.7 shows its methods (ignoring those it inherits from `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 in a `ByteBuf`, `ByteBufHolder` is a good choice.

5.5 ByteBuf allocation

This section describes the different ways in which `ByteBuf` instances can be managed.

5.5.1 On-demand: ByteBufAllocator

To reduce the overhead of allocating and deallocating memory, Netty supports pooling via the class `ByteBufAllocator, which can be used to allocate` instances of any of the `ByteBuf` varieties we have described. Whether to use pooling is an application-specific decision which does not, however, alter the `ByteBuf` API in any way.

Table 5.8 lists the operations provided by `ByteBufAllocator`.

Table 5.8 ByteBufAllocator methods

Name	Description

`buffer()`
`buffer(int);`
`buffer(int, int);`
	Return a `ByteBuf` with heap-based or direct data storage.

`heapBuffer()`
`heapBuffer(int)`
`heapBuffer(int, int)`
	Return a `ByteBuf` with heap-based storage.

`directBuffer()`
`directBuffer(int)`
`directBuffer(int, int)`
	Return a `ByteBuf` with direct storage.

`compositeBuffer()`
`compositeBuffer(int);`
`heapCompositeBuffer()`
`heapCompositeBuffer(int);`
`directCompositeBuffer()`
`directCompositeBuffer(int);`
	Return a `CompositeByteBuf` that can be expanded by adding heap-based or direct buffers.

`ioBuffer()`	Return a `ByteBuf` that will be used for I/O operations on a socket.

The `int` arguments accepted by some of these methods allow the user to specify the initial and maximum capacity values of the `ByteBuf`. You may recall that `ByteBuf` storage can expand until its maximum capacity is reached.

Getting a reference to the `ByteBufAllocator` is straightforward. You can obtain it either from the `Channel` (in theory, each `Channel` can have a distinct `ByteBufAllocator`) or through the `ChannelHandlerContext` that is bound to the `ChannelHandler` which implements your data-handling logic.

The following listing illustrates both ways of obtaining a `ByteBufAllocator`.

Listing 5.15 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 provides two implementations of `ByteBufAllocator`. One, `PooledByteBufAllocator`, pools `ByteBuf` instances to improve performance and minimize memory fragmentation. This implementation uses an efficent approach to memory allocation known as "jemalloc"[12] that

[12] http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf

has been adopted by a number of modern operating systems. The other implementation does not pool `ByteBuf` instances at all and returns a new instance every time.

Although Netty uses the `PooledByteBufAllocator` by default, this can be changed easily via the `ChannelConfig` API or by specifying a different allocator when bootstrapping your application. More details can be found in Chapter 9, "Bootstrapping Netty Applications".

5.5.2 Unpooled buffers

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`, which provides static helper methods to create unpooled `ByteBuf` instances. Table 5.9 lists the most important methods.

Table 5.9 `Unpooled` helper class

Name	Description

`buffer()`
`buffer(int)`
`buffer(int, int)`
	Returns an unpooled `ByteBuf` with heap-based storage

`directBuffer()`
`directBuffer(int)`
`directBuffer(int, int)`
	Returns an unpooled `ByteBuf` with direct storage

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

The `Unpooled` class also makes it easier to use the `ByteBuf` API in non-networking projects that can benefit from a high-performance extensible buffer API but do not need other parts of Netty.

5.5.3 ByteBufUtil

`The class ByteBufUtil` provides static helper methods for operating on a `ByteBuf`. Since this API is generic and unrelated to the use of pooling, these methods have been implemented outside the allocation classes.

The most valuable of these static methods is probably `hexdump()`, `which` prints hexadecumal representation of the contents of a `ByteBuf`. This can be useful in a variety of situations. A typical use is to log the contents of a `ByteBuf` for debugging purposes. A hex representation will generally provide a more usable log entry than would a direct representation of the byte values. Furthemore, the hex version can easily be converted back to the actual byte representation.

Another useful method is `boolean equals(ByteBuf, ByteBuf)`, which determines the equality of two `ByteBuf` instances. You may find that additional methods are particularly useful when you start to implement your own `ByteBuf` subclasses.

5.6 Reference counting

Netty 4 introduced reference counting for `ByteBuf` and `ByteBufHolder` (both of which implement the `ReferenceCounted` interface).

Reference counting itself is not complex; it involves tracking the number of references that apply to a specifed object. An instance of a class that implements `ReferenceCounted` will normally start out with an active reference count of 1. As long as the reference count is greater than 0 the object is guaranteed not to be released. When the number of active references decreases to 0, the instance will be released. Note that the semantics of "release" are specific to the implementation. At the very least, an object that has been released should no longer be available for use.

This technique is an essential part of how pooling implementations such as `PooledByteBufAllocator` reduce the overhead of memory allocation.

Listing 5.16 Reference counting

```
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.**

Listing 5.17 Release reference counted object

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

1. **Calling `release()` decerements the number of active references to the object. If this causes the reference count to reach 0 the object has been released and the method returns true.**

Trying to access reference-counted object that has been released will result in an `IllegalReferenceCountException`.

Note that a specific class can define its release counting "contract" in its own unique way. For example, one can envision a class that specifies that `release()` always sets the reference count to zero whatever its current value, thus invalidating all active references at once.

Who is responsible for release ?

In general, the party that the last to access an object is responsible for releasing it. In Chapter 6 we will explain the relevance of this concept to `ChannelHandler` and `ChannelPipeline`.

5.7 Summary

This chapter was devoted to Netty's data containers, based on `ByteBuf`. We started out by explaining the advantages of Netty's implementation over that provided by the JDK. We also highlighted the API's of the available variants and indicated which are best suited to specific use cases.

In the next chapter we will focus on `ChannelHandler`, which provides the vehicle for your data-processing logic. Since `ChannelHandler` makes heavy use of `ByteBuf`, we'll begin to see important pieces of the overall architecture of Netty falling into place.

6

ChannelHandler and ChannelPipeline

This chapter covers

- `Channel`
- `ChannelHandler`
- `ChannePipeline`
- `ChannelHandlerContext`

The `ByteBuf` we studied in the last chapter is the container Netty uses to "package" data. In this chapter we will examine how these containers move through an application, both incoming and outgoing, and how their contents are processed along the way.

Netty provides powerful support for the data-processing areas of application development. We have already had a first look at how `ChannelHandler`s can be chained together in a `ChannelPipeline` to structure processing steps in a flexible and modular fashion.

In this chapter and the following ones we will encounter a variety of use cases involving `ChannelHandler`, `ChannelPipeline` and an important related class, `ChannelHandlerContext`. We'll show how these fundamental components of the framework can help us to write clean and reusable processing implementations.

6.1 The ChannelHandler Family

Before we delve deeply into the internals of `ChannelHandler`, let's spend a few minutes on some of the underpinnings of this area of Netty's component model. This will provide a valuable background to our study of `ChannelHandler` and its subclasses.

6.1.1 The Channel Lifecycle

The `Channel` has a simple but powerful state model which is closely related to the `ChannelInboundHandler` API. The four `Channel` states are listed in Table 6.1.

Table 6.1 Channel lifeycle states

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

The normal lifecycle of a `Channel` is shown in Figure 6.1. As these state changes occur, corresponding events are generated, and these are communicated to `ChannelHandlers` in the `ChannelPipeline`, which can then act on them.

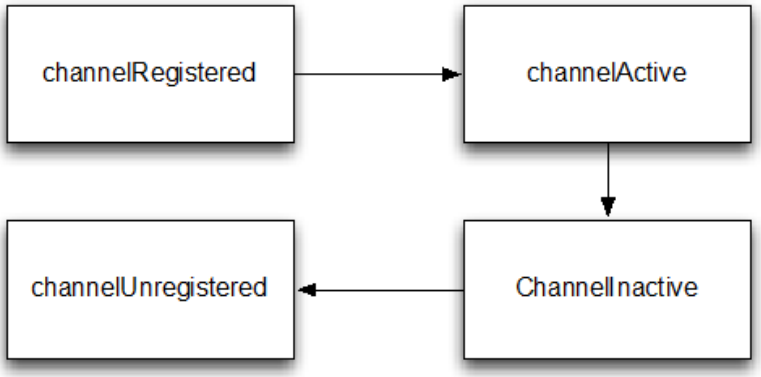


Figure 6.1 Channel State Model

6.1.2 The ChannelHandler Lifecycle

Now let's look at the lifecycle operations defined by the `ChannelHandler` interface, listed in Table 6.2. These are called after a `ChannelHandler` is added to or removed from a `ChannelPipeline`. Each of these methods takes a `ChannelHandlerContext` argument.

Table 6.2 ChannelHandler lifecycle methods

Type	Description
`handlerAdded`	Called when a `ChannelHandler` is added to a `ChannelPipeline`
`handlerRemoved`	Called when a `ChannelHandler` is removed from a `ChannelPipeline`
`exceptionCaught`	Called If an error happens during processing in the `ChannelPipeline`.

6.1.3 ChannelHandler Subinterfaces

Netty defines these two important subinterfaces of `ChannelHandler`:

- `ChannelInboundHandler` - processes inbound data and state changes of all kinds.
- `ChannelOutboundHandler` - processes outbound data and allows interception of all kinds of operations.

In the next sections we'll discuss these subinterfaces in detail.

ChannelHandler Adapters

Netty provides a simple skeleton implementation of `ChannelHandler that has` bodies for all of the declared method signatures. The methods of this class, `ChannelHandlerAdapter`, merely forward events to the next `ChannelHandler` in the pipeline until the end of the pipeline is reached. This class also serves as the base for `ChannelInboundHandlerAdapter` and `ChannelOutboundHandlerAdapter`.

All three adapter classes are intended to serve as starting points for your own implementations; you can extend them, overriding only the methods you need to customize.

6.1.4 ChannelInboundHandler

The lifecycle methods of `ChannelInboundHandler, listed in` Table 6.3, are called when data are received or when the state of the associated `Channel` changes. As we mentioned earlier, these methods map closely to the `Channel` lifecycle.

Table 6.3 ChannelInboundHandler methods

Type	Description
`channelRegistered`	Invoked when a `Channel` is registered to its `EventLoop` and is able to handle I/O.
`channelUnregistered`	Invoked when a `Channel` is deregistered from its `EventLoop` and cannot handle any I/O.
`channelActive`	Invoked when a `Channel` is active; the `Channel` is connected/bound and ready.
`channelInactive`	Invoked when a `Channel` leaves active state and is no longer connected to its remote peer.
`channelReadComplete`	Invoked when a read operation on the `Channel` has completed.
`channelRead`	Invoked if data are read from the `Channel`.
`channelWritabilityChanged`	Invoked when the writability state of the Channel changes. The user can ensure writes are not done too fast (with risk of an `OutOfMemoryError`) or can resume writes when the Channel becomes writable again. `Channel.isWritable()` can be used to detect the actual writability of the channel. The threshold for writability can be set via `Channel.config().setWriteHighWaterMark()` and `Channel.config().setWriteLowWaterMark()`.
`userEventTriggered(…)`	Invoked when a user calls `Channel.fireUserEventTriggered(…)` to pass a pojo through the `ChannelPipeline`. This can be used to pass user specific events through the `ChannelPipeline` and so allow handling those events.

Please note that a `ChannelInboundHandler` implementation that overrides `channelRead()` to process incoming messages is responsible for releasing resources. Netty uses pooled resources for `ByteBuf`, so failing to release resources will lead to a memory leak.

The correct approach is shown in Listing 6.1.

Listing 6.1 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 passing it to `ReferenceCountUtil.release()`**

Netty logs unreleased resources with a `WARN`-level log entry, making it fairly simple to find offending instances in the code. However, since managing resources by hand can be cumbersome, you can simplify matters by using `SimpleChannelInboundHandler`. Listing 6.2 gives a variation of Listing 6.1 that illustrates this.

Listing 6.2 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. **No need for any explicit release of resources**

Please note: since `SimpleChannelInboundHandler` releases resources automatically, *do not* store references to any messages for later use, as these will become invalid.

For a more detailed discussion please see Section **Error! Reference source not found.**.

6.1.5 ChannelOutboundHandler

Up to now we have been looking at implementations for processing inbound operations and data. `ChannelOutboundHandler` provides methods that are called when outbound operations are requested. These are the methods invoked by `Channel`, `ChannelPipeline`, and `ChannelHandlerContext`.

A powerful aspect of `ChannelOutboundHandler's` role is its ability to defer an operation or event on request. This enables some sophisticated approaches to request handling. For

example, you can defer flush operations if writing to the remote peer is suspended and pick them up at a later time.

Table 6.4 shows all of the methods defined by `ChannelOutboundHandler`. (Those inherited from `ChannelHandler` are not listed.)

Table 6.4 ChannelOutboundHandler methods

Type	Description
`bind`	Invoked on request to bind the `Channel` to a local address
`connect`	Invoked on request to connect the `Channel` to the remote peer
`disconnect`	Invoked on request to disconnect the `Channel` from the remote peer
`close`	Invoked on request to close the `Channel`
`deregister`	Invoked on request to deregister the `Channel` from its `EventLoop`
`read`	Invoked on request to read more data from the `Channel`
`flush`	Invoked on request to flush queued data to the remote peer through the `Channel`
`write`	Invoked on request to write data through the `Channel` to the remote peer

Almost 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`.

ChannelPromise vs. ChannelFuture

A `ChannelPromise` is a special `ChannelFuture` that allows you to either fail or success the `ChannelPromise` and so the operation to which it belongs. So whenever you call for example `Channel.write(…)` a new ChannelPromise will be created and passed throught the `ChannelPipeline`. The write itself will just return a `ChannelFuture` and so only allow you to get notified once the operation completes. Netty itself will use the `ChannelPromise` to be able to notify the returned `ChannelFuture`, which in fact is most of the times just the `ChannelPromise` itself (`ChannelPromise` extends `ChannelFuture`).

As we mentioned earlier, `ChannelOutboundHandlerAdapter` provides a skeleton implementation for `ChannelOutboundHandler` with base implementations of all of its methods. These simply forward the event to the next `ChannelOutboundHandler` in the pipeline by calling the equivalent method on the associated `ChannelHandlerContext`. It remains for you only to implement the methods you're interested in.

6.1.6 Resource Management

Whenever you act on data via `ChannelInboundHandler.channelRead(...)` or `ChannelOutboundHandler.write(...)` it is important to understand how you need to handle resources to ensure there are no resource leaks.

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.

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 for the 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.ja
va:697)

     io.netty.handler.codec.xml.XmlFrameDecoderTest.testDecodeWithXml(XmlFrameDeco
derTest.java:157)

     io.netty.handler.codec.xml.XmlFrameDecoderTest.testDecodeWithTwoMessages(XmlF
rameDecoderTest.java:133)

     ...
```

LEAK DETECTION LEVELS

Netty currently defines four leak detection levels that can be enabled when needed. Table 6.5 gives an overview.

Table 6.5 Leak detection levels

Level	Description
DISABLED	Disables Leak detection completely. While this even eliminates the 1 % overhead you should only do this after extensive testing.
SIMPLE	Tells if a leak was found or not. Again uses the sampling rate of 1%, 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. This it has a massive impact on performance. Use this only in the debugging phase.

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

For example:

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

With this in mind let us have a look what needs to be done to prevent such leaks in your `ChannelInboundHandler.channelRead(...)` and `ChannelOutboundHandler.write(...)` implementations.

if you handle a `channelRead(...)` operation and consume (not pass it to the next `ChannelInboundHandler` via `ChannelHandlerContext.fireChannelRead(...)`) a message you are responsible for releasing it as shown in Listing 6.3.

Listing 6.3 Handler that consume inbound data

```
@Sharable
public class DiscardInboundHandler
        extends ChannelInboundHandlerAdapter {                          //1
    @Override
    public void channelRead(ChannelHandlerContext ctx,
            Object msg) {
        ReferenceCountUtil.release(msg);                                //2
    }
}
```

1. **Extend ChannelInboundHandlerAdapter**
2. **Release resource by using ReferenceCountUtil.release(…)**

So remember, whenever you consume a message you are responsible to release it!

SimpleChannelInboundHandler – consuming inbound messages the easy way

As consuming inbound data and releasing it is such a common task Netty provides you with a special `ChannelInboundHandler` implementation called `SimpleChannelInboundHandler`. This implementation will automatically release a message once it was consumed by the user via the `channelRead0(…)` method.

if you handle a write operation and discard a message you are responsible for releasing it. Now let's look at how you could make use of this in practice. Listing 6.4 shows an implementation that discards all written data.

Listing 6.4 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.

So let us summarize this; It is the responsibility of the user to call `ReferenceCountUtil.release(message)` if a message is consumed / discarded 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.

6.2 ChannelPipeline

If we think of a `ChannelPipeline` simply as a series of `ChannelHandler` instances that intercept the inbound and outbound events that flow through a `Channel`, then it is easy to see how the interaction of these `ChannelHandler`s can provide the core of an application's data and event processing logic.

Every new `Channel` that is created is assigned a new `ChannelPipeline`. This association is permanent; the `Channel` can neither attach another `ChannelPipeline` nor detach the current one. This is a fixed aspect of Netty's component lifeycle and requires no action on the part of the developer.

Depending on its origin, an event will be handled by either a `ChannelInboundHandler` or a `ChannelOutboundHandler`. Subsequently it will be forwarded to the next handler of the same supertype by a call to `a ChannelHandlerContext` implementation.

ChannelHandlerContext

A `ChannelHandlerContext` enables a `ChannelHandler` to interact with its `ChannelPipeline` and other handlers. A handler can notify the next `ChannelHandler` in the `ChannelPipeline` and even modify dynamically the `ChannelPipeline` it belongs to.

`ChannelHandlerContext has a rich API for handling events and performing I/O operations. (Please see Section 6.3 for more information on ChannelHandlerContext.)`

Figure 6.2 illustrates a typical ChannelPipeline layout with both inbound and outbound ChannelHandlers.

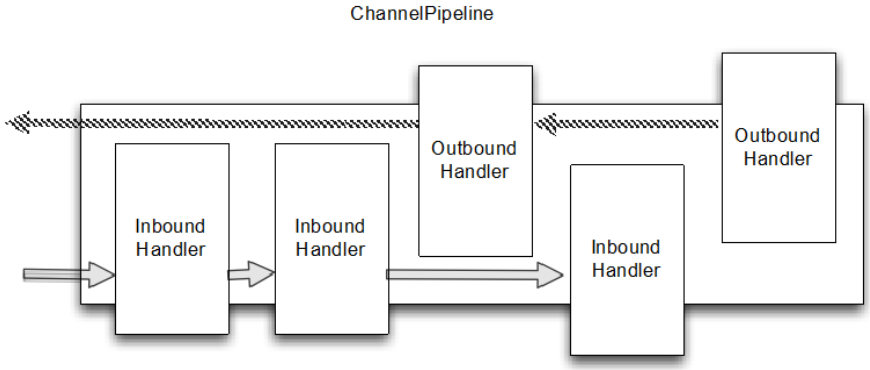


Figure 6.2 `ChannelPipeline` and `ChannelHandlers`

Figure 6.2 illustrates our earlier statement that a `ChannelPipeline` is primarily a series of `ChannelHandlers`. `ChannelPipeline` also provides methods to propagate events through the `ChannelPipeline` itself. If an inbound event is triggered it is passed from the beginning to the end of the `ChannelPipeline`. For example, in this diagram an outbound I/O event will start at the right end of the `ChannelPipeline` and proceed to the left.

ChannelPipeline relativity

You might say that from the point of view of an event traveling through the `ChannelPipeline`, the "beginning" of the `ChannelPipeline` depends on whether the event is inbound or outbound. However, Netty always refers to the inbound entry to the `ChannelPipeline` (the left side in Figure 6.2) as the "beginning" and the outbound entry (the right side) as the "end."

When we have finished adding our mix of inbound and outbound handlers to a `ChannelPipeline` using the `ChannelPipeline.add*()` methods, the "ordinal" of each `ChannelHandler` is its position from "beginning" to "end" as we have just defined them. Thus, if we number the handlers in Figure 6.1 in order from left to right, the first `ChannelHandler` seen by an inbound event will be #1, while the first handler seen by an outbound event will be #5.

As the pipeline propagates an event, it determines whether the next `ChannelHandler` is of a type that matches the direction of movement. If not, the `ChannelPipeline` skips that `ChannelHandler` and proceeds to the next one in the appropriate direction. Keep in mind, however, that a handler might implement both `ChannelInboundHandler` and `ChannelOutboundHandler` interfaces.

6.2.1 Modifying a ChannelPipeline

A `ChannelHandler` can modify the layout of a `ChannelPipeline` in real time by adding, removing or replacing other `ChannelHandlers`. (It can remove itself from the `ChannelPipeline` as well.) This is one of the most important capabilities of the `ChannelHandler` so let's take a look at how it is done.

Table 6.6 lists the relevant methods of the `ChannelHandler API`.

Table 6.6 `ChannelHandler` methods for modifying 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`.

Listing 6.5 shows these methods in operation.

Listing 6.5 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 a `FirstHandler` instance**
2. **Add this instance to the `ChannelPipeline` as "`handler1`"**
3. **Add an instance of a `SecondHandler` to the `ChannelPipeline` in the first slot, as "`handler2`". This means it will be placed before the already existing "`handler1`".**
4. **Add a `ThirdHandler` instance to the ChannelPipeline in the last slot as "`handler3`".**
5. **Remove "`handler3`" by name.**
6. **Remove the `FirstHandler` by reference (there is only one so it is not necessary to use the assigned name "`handler1`".**
7. **Replace the `SecondHandler` instance which was added as "`handler2`" with a `FourthHandler` named "`handler4`".**

We will see later on that this ability to add, remove and replace `ChannelHandler`s with ease lends itself to the implementation of extremely flexible logic.

`ChannelHandler` execution in the ChannelPipeline and blocking

Normally each `ChannelHandler` that is added to the `ChannelPipeline` will process the event that is passed through it its `EventLoop` (the I/O thread). It is critically important not to block this thread as it would have a negative effect on the overall handling of I/O.

Sometimes it may be necessary to interface with legacy code that uses blocking APIs. For this case the `ChannelPipeline` has `add()` methods that accept an `EventExecutorGroup`. 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 away from the `EventLoop` of the `Channel` itself. A default implementation which is called `DefaultEventExecutorGroup` comes as part of Netty.

In addition to these operations there are others for accessing `ChannelHandlers` present in the pipeline by type or by name, listed in Table 6.7.

Table 6.7 ChannelPipeline operations for retrieving ChannelHandlers

Name	Description
`get(…)`	Return a `ChannelHandler` by type or name
`context(…)`	Return the `ChannelHandlerContext bound to a ChannelHandler.`
`names()` `iterator()`	Return the names or of all the `ChannelHander` in the `ChannelPipeline`.

6.2.2 Firing events

The ChannelPipeline API exposes additional methods for invoking inbound and outbound operations. Table 6.8 lists the inbound operations, which notify ChannelInboundHandlers of events occurring in the ChannelPipeline.

Table 6.8 Inbound operations on ChannelPipeline

Name	Description
fireChannelRegistered	Calls channelRegistered(ChannelHandlerContext) on the next ChannelInboundHandler in the ChannelPipeline.
fireChannelUnregistered	Calls channelUnregistered(ChannelHandlerContext) on the next ChannelInboundHandler in the ChannelPipeline.
fireChannelActive	Calls channelActive(ChannelHandlerContext) on the next ChannelInboundHandler in the ChannelPipeline.
fireChannelInactive	Calls channelInactive(ChannelHandlerContext) on the next ChannelInboundHandler in the ChannelPipeline.
fireExceptionCaught	Calls exceptionCaught(ChannelHandlerContext, Throwable) on the next ChannelHandler in the ChannelPipeline.
fireUserEventTriggered	Calls userEventTriggered(ChannelHandlerContext, Object) on the next ChannelInboundHandler in the ChannelPipeline.
fireChannelRead	Calls channelRead(ChannelHandlerContext, Object msg) on the next ChannelInboundHandler in the ChannelPipeline.
fireChannelReadComplete	Calls channelReadComplete(ChannelHandlerContext) on the next ChannelStateHandler in the ChannelPipeline.

On the outbound side, handling an event will cause some action to be taken on the underlying socket. Table 6.9 lists the outbound operations of the ChannelPipeline API.

Table 6.9 Outbound operations on ChannelPipeline

Method name	Description
bind	Bind the Channel to a local address. This will call bind(ChannelHandlerContext, SocketAddress, ChannelPromise) on the next ChannelOutboundHandler in the ChannelPipeline.
connect	Connect the Channel to a remote address. This will call connect(ChannelHandlerContext, SocketAddress, ChannelPromise) on the next ChannelOutboundHandler in the ChannelPipeline.
disconnect	Disconnect the Channel. This will call disconnect(ChannelHandlerContext, ChannelPromise) on the next

	ChannelOutboundHandler in the ChannelPipeline.
close	Close the Channel. This will call close(ChannelHandlerContext, ChannelPromise) on the next ChannelOutboundHandler in the ChannelPipeline.
deregister	Deregister the Channel from the previously assigned EventExecutor (the EventLoop). This will call deregister(ChannelHandlerContext, ChannelPromise) on the next ChannelOutboundHandler in the ChannelPipeline.
flush	Flush all pending writes of the Channel. This will call flush(ChannelHandlerContext) on the next ChannelOutboundHandler in the ChannelPipeline.
write	Write a message to the Channel. This will call write(ChannelHandlerContext, Object msg, ChannelPromise) on the next ChannelOutboundHandler in the ChannelPipeline. Note: this does not write the message to the underlying Socket, but only queues it. To write it to the Socket call flush() or writeAndFlush().
writeAndFlush	Convenience method for calling write() then flush().
read	Requests to read more data from the Channel. This will call read(ChannelHandlerContext) on the next ChannelOutboundHandler in the ChannelPipeline.

Let's summarize what we've learned in this section.

- A ChannelPipeline holds the ChannelHandlers associated with a Channel.
- A ChannelPipeline can be modified on the fly by adding and removing ChannelHandlers as needed.
- ChannelPipeline has a rich API for invoking actions in response to inbound and outbound events.

6.3 ChannelHandlerContext

The interface ChannelHandlerContext represents an association between a ChannelHandler and a ChannelPipeline, and an instance is created whenever a ChannelHandler is added to a ChannelPipeline. The primary function of ChannelHandlerContext to manage the interaction of the ChannelHandler with which it is associated with other ChannelHandlers in the same ChannelPipeline.

ChannelHandlerContext has numerous methods, some of which are also present on Channel and ChannelPipeline itself. However, if you invoke these methods on a Channel or ChannelPipeline instance they propagate through the entire pipeline. By contrast, the same method called on a ChannelHandlerContext starts at the current associated ChannelHandler and propagates only to the next ChannelHandler in the pipeline capable of handling the event.

A summary of the `ChannelHandlerContext` API is given in Table 6.10.

Table 6.10 ChannelHandlerContext API

Method name	Description
`bind`	Request to bind to the given `SocketAddress` and return a `ChannelFuture`.
`channel`	Return the `Channel` which is bound to this instance.
`close`	Request to close the `Channel` and return a `ChannelFuture`.
`connect`	Request to connect to the given `SocketAddress` and return a `ChannelFuture`.
`deregister`	Request to deregister from the previously assigned `EventExecutor` and return a `ChannelFuture`.
`disconnect`	Request to disconnect from the remote peer and return a `ChannelFuture`.
`executor`	Return the `EventExecutor` that dispatches events.
`fireChannelActive`	A `Channel` is active (connected).
`fireChannelInactive`	A `Channel` is inactive (closed).
`fireChannelRead`	A `Channel` received a message.
`fireChannelReadComplete`	Triggers a `channelWritabilityChanged` event to the next `ChannelInboundHandler`.
`handler`	Returns the `ChannelHandler` bound to this instance.
`isRemoved`	Returns true if the associated `ChannelHandler` was removed from the `ChannelPipeline`.
`name`	Returns the unique name of this instance.
`pipeline`	Returns the associated `ChannelPipeline`.
`read`	Request to read data from the `Channel` into the first inbound buffer. Triggers a `channelRead` event if successful and notifies the handler of `channelReadComplete`.
`write`	Request to write a message via this instance through the pipeline.

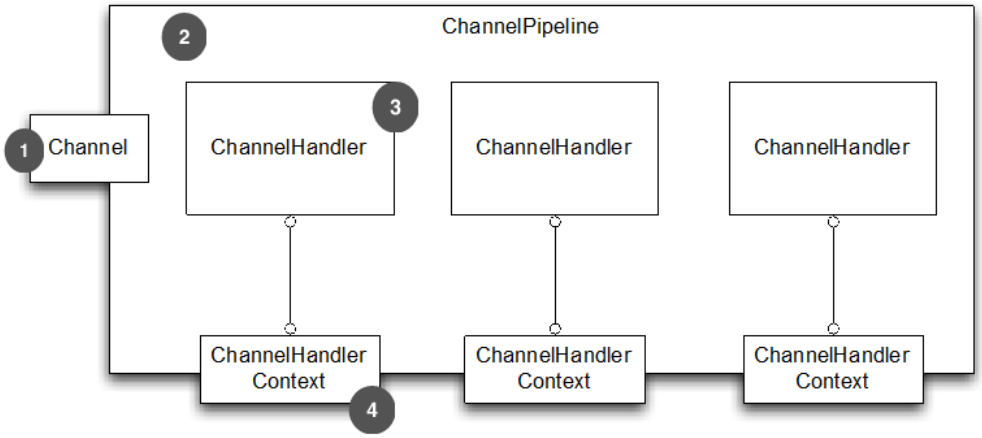
Additional notes:

- The `ChannelHandlerContext` associated with a `ChannelHandler` never changes so it is safe to cache a reference to it.

- As we indicated earlier, `ChannelHandlerContext` methods involve a shorter event flow than the same methods available on other classes. This should be exploited where possible to provide the best performance.

6.3.1 Using ChannelHandler

In this section we'll illustrate the use of `ChannelHandlerContext` and the differing behaviors of methods available on `ChannelHandlerContext`, `Channel` and `ChannelPipeline`. Please examine Figure 6.3, which shows the relationships among `ChannelPipeline`, `Channel`, `ChannelHandler` and `ChannelHandlerContext`.



1. **`Channel` bound to `ChannelPipeline`**
2. **`ChannelPipeline` bound to `Channel` containing `ChannelHandlers`**
3. **`ChannelHandler`**
4. **`ChannelHandlerContext` created when adding `ChannelHandler` to `ChannelPipeline`**

Figure 6.3 Channel, ChannelPipeline, ChannelHandler and ChannelHandlerContext

In Listing 6.6 we get a reference to the `Channel` from a `ChannelHandlerContext`. Calling `write()` on the `Channel` causes a write event to flow all the way through the pipeline.

Listing 6.6 Accessing the Channel from a ChannelHandlerContext

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

1. **Get a reference to the `Channel` associated with the `ChannelHandlerContext`**
2. **Write buffer via the `Channel`**

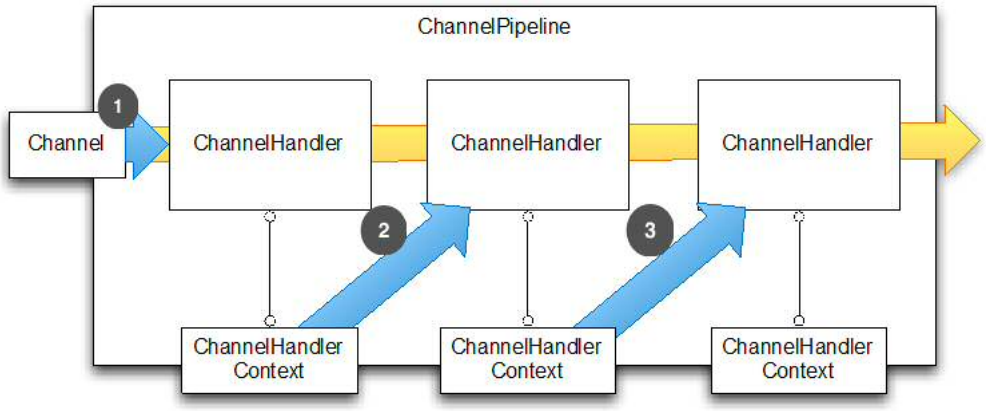
Listing 6.7 shows a similar example using a `ChannelPipeline`. Again, the reference is retrieved from the `ChannelHandlerContext`.

Listing 6.7 Accessing the ChannelPipeline from a ChannelHandlerContext

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

1. **Get a reference to the `ChannelPipeline` associated with the `ChannelHandlerContext`**
2. **Write the buffer via the `ChannelPipeline`**

The flow in both Listings 6.6 and 6.7 is identical, as shown in Figure 6.4. It is important to note that although the `write()` invoked on either the `Channel` or the `ChannelPipeline` operation propagates the event all the way through the pipeline, at the `ChannelHandler` level the movement from one handler to the next is invoked on the `ChannelHandlerContext`.



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 6.4 Event propagation via the Channel or the ChannelPipeline

Why might you want to propagate an event starting at a specific point in the `ChannelPipeline`?

- to reduce the overhead of passing the event through `ChannelHandlers` that are not interested in it
- to exclude processing by specific handlers that would be interested in it

To invoke processing starting with a specific `ChannelHandler` you must refer to the `ChannelHandlerContext` that is associated with the `ChannelHandler` *before* that one. This `ChannelHandlerContext` will invoke the `ChannelHandler` that *follows* the one with which it is associated.

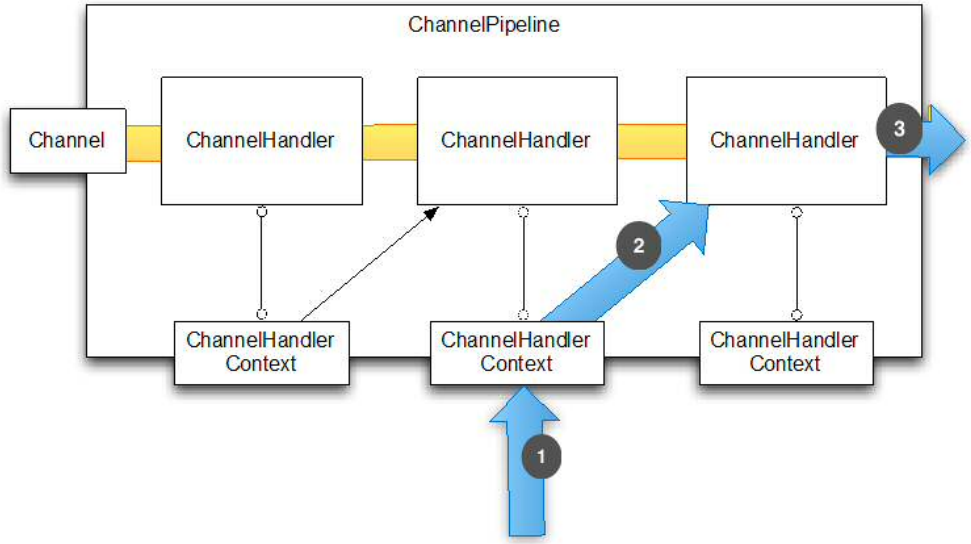
Listing 6.8 and Figure 6.5 illustrate this usage.

Listing 6.8 Events via ChannelPipeline

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

1. **Get a reference to a `ChannelHandlerContext`**
2. **`write()` will send the buffer to the next `ChannelHandler`.**

As shown below, the message flows through the `ChannelPipeline` starting at the *next* `ChannelHandler`, `bypassing` all the preceding ones.



1. **`ChannelHandlerContext` method invoked**
2. **Event is passed to next `ChannelHandler`**
3. **Event moves out of `ChannelPipeline` as this `ChannelHandler` is the last one.**

Figure 6.5 Event flow for operations triggered via the ChannelHandlerContext

The use case we have just described is a common one, especially useful for calling operations on a specific `ChannelHandler` implementation.

6.3.2 Advanced usages of ChannelHandler and ChannelHandlerContext

As we saw in Listing 6.6 you can get a reference to the enclosing `ChannelPipeline` by calling the `pipeline()` method of `ChannelHandlerContext`. This enables runtime manipulation of the pipeline's `ChannelHandlers and can be exploited to implement some sophisticated requirements, for example,` adding a `ChannelHandler` to a pipeline to support a dynamic protocol change.

Other advanced use cases can be implemented by keeping a reference to a `ChannelHandlerContext` for later use, which might take place outside any `ChannelHandler` methods and even from a different thread. Listing 6.9 shows this pattern being used to trigger an event.

Listing 6.9 `ChannelHandlerContext` usage

```
public class WriteHandler extends ChannelHandlerAdapter {
    private ChannelHandlerContext ctx;
    @Override
    public void handlerAdded(ChannelHandlerContext ctx) {
        this.ctx = ctx;                                          //1
    }
    public void send(String msg) {                               //2
        ctx.write(msg);
    }
}
```

1. **Store reference to `ChannelHandlerContext` for later use**
2. **Send message using previously stored `ChannelHandlerContext`**

Since a `ChannelHandler` can belong to more than one `ChannelPipeline`, it can be bound to multiple `ChannelHandlerContext` instances. However, a `ChannelHandler` intended for this usage must be annotated with `@Sharable`. Otherwise, attempting to add it to more then one `ChannelPipeline` will trigger an exception. Furthermore, it must be both thread-safe and safe to use with multiple simultanious channels (i.e., connections).

Listing 6.10 shows a correct implementation of this pattern.

Listing 6.10 A shareable ChannelHandler

```
@Sharable                                                        //1
public class SharableHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        System.out.println("Channel read message " + msg);
        ctx.fireChannelRead(msg);                                //2
    }
}
```

1. **Annotate with `@Sharable`**
2. **Log method call and forward to next ChannelHandler**

This `ChannelHandler` implementation above meets all the requirements for inclusion in multiple pipelines; it is annotated with `@Sharable` and doesn't hold any state. Listing 6.11, on the otherhand, will cause problems.

Listing 6.11 Invalid usage of `@Sharable`

```
@Sharable                                                        //1
public class UnsharableHandler 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`**

The problem with this code is that it has state: an instance variable that holds a count of method invocations. Adding an instance of this class to the `ChannelPipeline` will very likely produce errors when it is accessed by concurrent channels. (Of course, this simple case could be corrected by modifying `channelRead()` with `synchronized`.)

In summary, use `@Sharable` only if you are certain of your `ChannelHandler's` thread-safety.

Why share a ChannelHandler?

A common reason for installing a single `ChannelHandler` in multiple `ChannelPipelines` is to gather statistics across multiple `Channels`.

This concludes our discussion of `ChannelHandlerContext` and its relationship to other framework components Next we will examine the `Channel` state model, preparatory to looking more closely at `ChannelHandler` itself.

6.4 Summary

This chapter provided an in-depth look into Netty's data processing component: `ChannelHandler`. We discussed how `ChannelHandlers` are chained and how they interact with the `ChannelPipeline` in their incarnations as `ChannelInboundHandlers` and `ChannelOutboundHandlers`.

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

7

The Codec Framework

This chapter covers

- Decoders
- Encoders
- Codecs

In the previous chapter we discussed different ways of hooking into the processing chain to intercept operations or data, and we showed how `ChannelHandler` and its associated classes can be used to implement almost any kind of logic required by an application. But just as standard architectural patterns often have dedicated frameworks, common processing patterns are good candidates for targeted implementations, which can save us considerable development time and effort.

In this chapter we'll study encoding and decoding - the conversion of data from one protocol-specific format to another. This processing pattern is handled by components commonly called "codecs". Netty provides components that make it straightforward to write codecs for a broad range of protocols. For example, if you are building a Netty-based mail server you will find these tools invaluable for implementing `POP3`[13], `IMAP`[14] and `SMTP`[15].

7.1 What is a Codec?

Every network application has to define how raw bytes transferred between peers are to be parsed and converted to - and from - the target program's data format. This conversion logic will be handled by a "codec," which consists of a "decoder" and an "encoder".

Both decoders and encoders transform one sequence of bytes to another. How do we distinguish them?

Think of a "message" as a structured sequence of bytes having semantic meaning for a specific application - its "data". The *encoder* is the component that converts that message to a format suitable for transmission (most likely a byte stream), while the corresponding *decoder* transforms the transmitted data back to the program's message format. It is logical, then, to think of conversion "from" a message as operating on *outbound* data, while conversion "to" a message is handling *inbound* data.

Let’s have a look at the classes that Netty provides for implementing codecs.

7.2 Decoders

In this section we'll survey the classes supplied for decoder implementation and present some concrete examples of how and when you might use them.

---

[13] http://www.ietf.org/rfc/rfc1939.txt
[14] https://www.ietf.org/rfc/rfc2060.txt
[15] http://www.ietf.org/rfc/rfc2821.txt

Netty has a rich set of abstract base classes that simplify the writing of decoders. These cover two distinct use cases:

- decoding from bytes to message (`ByteToMessageDecoder` and `ReplayingDecoder`)
- decoding from message to message (`MessageToMessageDecoder`)

As we said, a decoder is responsible for transforming inbound data from one format to another. So it won't surprise you to learn that a Netty decoder is an abstract implementation of `ChannelInboundHandler`. When would you use a decoder? Simple: whenever you need to transform inbound data for the next `ChannelInboundHandler` in the `ChannelPipeline`.

Furthermore, thanks to the design of `ChannelPipeline`, you can chain together multiple decoders in the pipeline to implement the required processing logic. This is an important example of how the framework supports reuse.

7.2.1 ByteToMessageDecoder

Decoding from bytes to messages (or to another sequence of bytes) is such a common task that Netty provides an abstract base class just to handle it: `ByteToMessageDecoder`. You can't know whether the remote peer will send a complete "message" all at once, so this class buffers inbound data for you until it is ready for processing. Table 7.1 explains its two most important methods.

Table 7.1 ByteToMessageDecoder API

Method name	Description
`Decode`	This is the only abstract method you need to implement. It is called with a `ByteBuf` having the incoming bytes and a `List` into which decoded messages are added. `decode()` is called repeatedly until the `List` is empty on return. The contents of the `List` are then passed to the next handler in the pipeline.
`decodeLast`	The default implementation provided simply calls `decode()`. This method is called once, when the `Channel` goes inactive. Override to provide special handling,

Suppose we receive a byte stream containing simple integers, each to be handled separately. In this case we will read each integer from the inbound `ByteBuf` and pass it to the next `ChannelInboundHandler` in the pipeline. To "decode" the byte stream into integers we will extend `ByteToMessageDecoder`. The implementation class, "To`IntegerDecoder`", is shown in Figure 7.1.

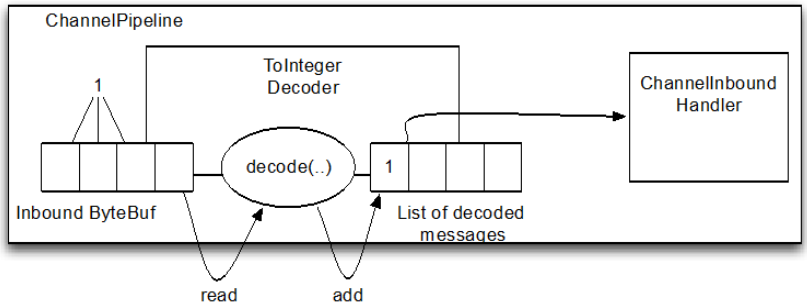


Figure 7.1 ToIntegerDecoder

Four bytes at a time are read from the inbound `ByteBuf`, decoded to an integer, and added to a `List` (in this case as an `Integer`). When no more items can be added to the list, its contents will be sent to the next `ChannelInboundHandler`.

Listing 7.1 shows the code for `ToIntegerDecoder`.

Listing 7.1 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 `int` from inbound `ByteBuf`, add to the `List` of decoded messages**

Although `ByteToMessageDecoder` simplifies this pattern, you might find it a bit annoying to have to verify that the input `ByteBuf` has enough data before you can do the actual read (here `readInt()`). In the next section we'll look into `ReplayingDecoder`, a special decoder which, at the cost of a little overhead, eliminates this step.

Reference Counting in Codecs

As we mentioned in Chapters 5 and 6, special care should be taken with regard to reference counting. For encoders and decoders the procedure is quite simple. Once a message has been encoded or decoded it wll automatically be released by a call to `ReferenceCountUtil.release(message)`. If you need to hold on to a reference for later use and don't want to release the message, you can call `ReferenceCountUtil.retain(message)`. This will increment the reference count, preventing the message from being released.

7.2.2 ReplayingDecoder

`ReplayingDecoder` is a special abstract base class for byte-to-message decoding that eliminates the need for the `readableBytes()` call in Listing 7.1. It accomplishes this by wrapping the input `ByteBuf` with a custom implementation that performs that check. If sufficient data are not available the `decode` loop is terminated (see comment in Table 1).

ByteToMessageDecoder and ReplayingDecoder

Note that `ReplayingDecoder` extends `ByteToMessageDecoder`, so its API is the same as that shown in Table 7.1.

You should be aware of these aspects of the `ByteBuf` wrapper implementation used by `ReplayingDecoder`:

- Not all standard `ByteBuf` operations are supported. If an unsupported method is called an `UnreplayableOperationException` will be thrown.
- `ReplayingDecoder` is slightly slower then `ByteToMessageDecoder`.

If these restrictions are acceptable you may prefer using `ReplayingDecoder` to `ByteToMessageDecoder`. Here is a simple guideline:

Use `ByteToMessageDecoder` if it doesn't introduce excessive complexity. Otherwise, use `ReplayingDecoder`.

Listing 7.2 shows the alternative implementation of `ToIntegerDecoder` using `ReplayingDecoder`.

Listing 7.2 `ReplayingDecoder`

```
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.**

If insufficient bytes are available the `ByteBuf` throws an `Error` which is caught by the `ReplayDecoder`. The `decode` method will be called later, once more data are ready. Otherwise, the decoded data are added to the `List`.

If we compare Listings 7.1 and 7.2 it is evident that the implementation using `ReplayingDecoder` is a bit simpler. While in itself this is a simple example, the implications of using one or the other base classes in a more complex case might be significant.

More Decoders

The following classes handle more complex use cases.

- `io.netty.handler.codec.LineBasedFrameDecoder`. This class, used internally by Netty, parses the incoming data by end-of-line control characters (`"\n"` or `"\r\n"`).
- `io.netty.handler.codec.http.HttpObjectDecoder` is a decoder for HTTP data.

There are numerous other useful decoder implementations as well.

7.2.3 MessageToMessageDecoder

In this section we'll explain how to decode one message format to another (for example, POJO to POJO) using the abstract base class `MessageToMessageDecoder`. Table 7.2 lists the methods.

Table 7.2 MessageToMessageDecoder API

Method name	Description
`decode`	`decode` is the only abstract method you need to implement. It is called for each inbound message to be decoded to another format . The decoded messages are then passed to the next `ChannelInboundHandler` in the pipeline.
`decodeLast`	The default implementation provided simply calls `decode()`. This method is called once, when the `Channel` goes inactive. Override to provide special handling,

As an example let's take a case where you need to convert `Integers` encoded in the incoming data to `String`s. By now we have enough experience with Netty's reuse-oriented design to know that this processing will take place in the `ChannelPipeline`, executed by a dedicated decoder instance. (Recall that Netty's codec classes extend `ChannelHandler`.)

To decode from one message format (`Integer`) to another (`String`), we'll provide an `IntegerToStringDecoder` that extends `MessageToMessageDecoder`.

As this is a parameterized class, the signature of our implementation will be:

```
public class IntegerToStringDecoder extends
      MessageToMessageDecoder<Integer>
```

So the signature of our `decode()` method will be:

```
protected void decode( ChannelHandlerContext ctx,
      Integer msg, List<Object> out ) throws Exception
```

That is, the inbound message is passed in not as a `ByteBuf` but as the parameter type declared in the class definition (here `Integer`). As in previous examples, the decoded messages (here `Strings`) will be added to the `List<Object>` and forwarded to the next `ChannelInboundHandler`.

This is illustrated in Figure 7.2.

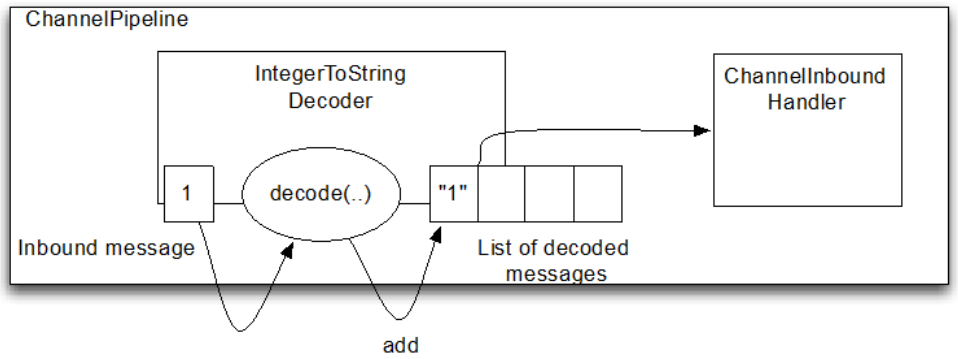


Figure 7.2 IntegerToStringDecoder

The implementation is shown in Listing 7.3.

Listing 7.3 `MessageToMessageDecoder` **- 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()`**

As we pointed out above, the parameter type applied to the class (here `Integer`) specifies the type of the message argument to `decode()`.

HttpObjectAggregator

For a more complex example, please examine the class `io.netty.handler.codec.http.HttpObjectAggregator`, which extends the same base class we have been studying as `MessageToMessageDecoder<HttpObject>`.

7.2.4 Handling too big frames while decoding

As Netty is an asynchronous framework you will need to buffer bytes in memory until you are able to decode them in some way. Consequently, you must not allow your decoder to buffer enough data to exhaust available memory. To address this common concern Netty provides a `TooLongFrameException`, usually thrown by decoders if a frame becomes too long.

To avoid this problem you can set a threshold maximum number of bytes in your decoder which, if exceeded, will cause a `TooLongFrameException` to be thrown (and caught in `ChannelHandler.exceptionCaught()`). It is then up to the user of the decoder to decide how to handle it. While some protocols, such as HTTP, allow such cases to be handled by the return of a special response, others may not, in which event the only option may be to close the connection.

Listing 7.4 shows how a `ByteToMessageDecoder` can make use of the `TooLongFrameException` to notify other `ChannelHandlers` in the `ChannelPipeline` about the occurence.

Listing 7.4 `ShortToByteEncoder` encodes shorts into a `ByteBuf`

```
public class SafeByteToMessageDecoder extends ByteToMessageDecoder {      //1
    private static final MAX_FRAME_SIZE = 1024;

    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in,
            List<Object> out) throws Exception {
        int readable = in.readableBytes();
        if (readable > MAX_FRAME_SIZE) {                       //2
            in.skipBytes(readable);                            //3
            throw new TooLongFrameException("Frame too big!");
        }
        // do something
        ...
    }
}
```

1. **Implementation extends `ByteToMessageDecoder` to decode bytes to messages**
2. **Check if the buffer has more then MAX_FRAME_SIZE bytes buffered.**
3. **Skip all readable bytes and throw a `TooLongFrameException` to notify the `ChannelHandlers` in the `ChannelPipeline` about the frame.**

This kind of protection is especially important if you decode a protocol that has a variable frame size.

Up to here we have examined common use cases for decoders and the abstract base classes Netty provides for building them. But decoders are only one side of the coin. On the other, completing the Codec API, we have encoders, which transform messages to outbound data. This will be our next topic.

7.3 Encoders

To review our earlier definition, an encoder trasforms outbound data from one format to another, thus it implements `ChanneOutboundHandler`. As you might expect based on the

preceding discussion, Netty offers a set of classes to help you to write encoders. Not surprisingly, they address the reverse of the decoder functions:

- encode from message to bytes
- encode from message to message

We'll start our examination of these classes with the abstract base class `MessageToByteEncoder`.

7.3.1 MessageToByteEncoder

Earlier we learned how to convert bytes to messages using `ByteToMessageDecoder`. We'll do the reverse with `MessageToByteEncoder`. Table 7.3 shows the API.

Table 7.3 MessageToByteEncoder API

Method name	Description
`encode`	The `encode` method is the only abstract method you need to implement. It is called with the outbound message, which this class will encodes to a `ByteBuf`. The `ByteBuf` is then forwarded to the next `ChannelOutboundHandler` in the `ChannelPipeline`.

The reason this class has only one method, while decoders have two, is that decoders often need to to produce a "last message" after the `Channel` has closed. For this reason the `decodeLast()` method is provided. Clearly this is not the case for an encoder; there is no sense in producing a message after the connection has been closed!

Let's see this class in action to better understand its usage. In the following example we produce `Short` values and want to encode them into a `ByteBuf` to send over the wire. We provide `ShortToByteEncoder` for this purpose. Figure 7.3 shows the logic.

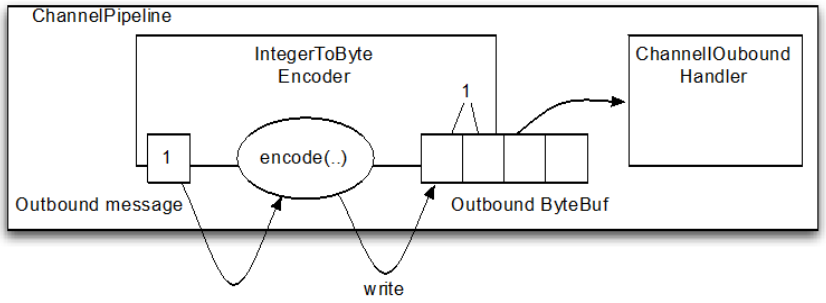


Figure 7.3 ShortToByteEncoder

Figure 7.3 shows that the encoder receives `Short` messages, encodes them, and writes them to a `ByteBuf`. This `ByteBuf` is then forwarded to the next `ChannelOutboundHandler` in the pipeline. Every `Short` will take up two bytes in the `ByteBuf`.

The implementation is shown in Listing 7.5.

Listing 7.5 `ShortToByteEncoder` encodes shorts into a `ByteBuf`

```
public class ShortToByteEncoder 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 provides several `MessageToByteEncoder` classes to serve as the basis for your own implementations. The class `WebSocket08FrameEncoder` provides a good practical example. You will find it in the package `io.netty.handler.codec.http.websocketx`.

7.3.2 MessageToMessageEncoder

We have already seen how to *decode inbound* data from one message format to another. To complete the picture we need a way to *encode* from one message to another for *outbound* data. `MessageToMessageEncoder` provides this capability, as illustrated in. Table 7.4. Again there is only one method, there being no need to produce "last messages".

Table 7.4 MessageToMessageEncoder API

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

For our example, we will encode Integer messages to String messages. You can do this easily with `MessageToMessageEncoder`, as shown in Figure 7.4.

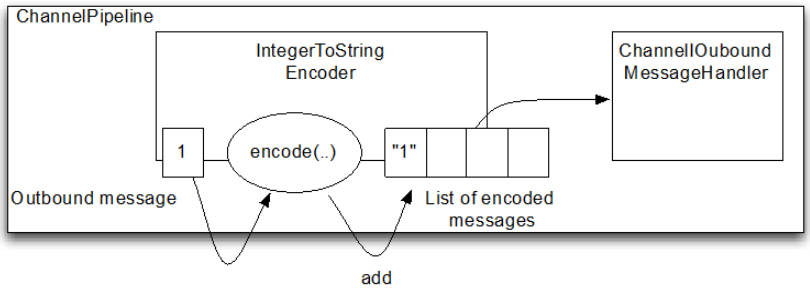


Figure 7.4 IntegerToStringEncoder

The encoder extracts `Integer`s from the outbound byte stream and forwards their `String` representation to the next `ChannelOutboundHandler` in the `ChannelPipeline`. Listing 7.6 shows the details.

Listing 7.6 `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`**

For a more complex example employing the `MessageToMessageEncoder` please examine `ProtobufEncoder`, which can be found in the package `io.netty.handler.codec.protobuf`.

7.4 Abstract Codec classes

Although we have been discussing decoders and encoders as distinct entities you may sometimes find it useful to manage transformations of both inbound and outbound data and messages in one class. Netty's abstract Codec classes are useful for this purpose, as they bundle together pairs of decoders and encoders to provide the same operations on bytes and messages we have just studied. (And as you might suspect, these classes implement both `ChannelInboundHandler` and `ChannelOutboundHandler`.)

You might wonder if there as a reason not to use these composite classes all the time rather than using separate decoders and encoders? The simplest answer is that tightly coupling the two functions reduces their reusability, while keeping them separate allows for easier extension of each. As we look at the abstract codec classes we'll also compare and contrast them with the corresponding, individual decoders and encoders.

7.4.1 ByteToMessageCodec

Lets examine the case where we need to decodes bytes to some kind of message, perhaps a POJO, and then back again. A `ByteToMessageCodec` will handle this for us, as it combines both a `ByteToMessageDecoder` and the reverse, a `MessageToByteDecoder`. The important methods are listed in table 7.5.

Table 7.5 ByteToMessageCodec API

Method name	Description
`Decode`	This method is called as long as bytes are available to be consumed. It converts the inbound `ByteBuf` to the specified message format and forwards them to the next

	`ChannelInboundHandler` in the pipeline.
`decodeLast`	The default implementation of this method delegates to `decode()`. It is called only be called once, when the `Channel` goes inactive. For special handling it can be oerridden.
`encode`	This method is called for each message to be written through the `ChannelPipeline`. The encoded messages are contained in a `ByteBuf` which is forwarded to the next `ChannelOutboundHandler` in the pipeline.

What would be a good use case for the `ByteToMessageCodec`? Any kind of a request / response protocol could be a candidate, for example SMTP. The codec would read incoming bytes and decode them to a custom message type, say `SmtpRequest`. When this received, an `SmtpResponse` will be produced, which will be encoded back to bytes for transmission.

7.4.2 MessageToMessageCodec

In Section 7.3.2 we saw an example of using `MessageToMessageEncoder` to convert from one message format to another. Let's look at `MessageToMessageCodec` to see how to do the round trip with a single class.

Before going into the details, let's look at the important methods in table 7.6.

Table 7.6 Methods of MessageToMessageCodec

Method name	Description
`decode`	This method is called with the inbound messages of the codec and decodes them to messages. Those messages are forwarded to the next ChannelInboundHandler in the `ChannelPipeline`
`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 outbound message to be moved through the `ChannelPipeline`. The encoded messages are forwarded to the next `ChannelOutboundHandler` in the pipeline.

`MessageToMessageCodec` is a parameterized class, defined as follows:

```
public abstract class MessageToMessageCodec<INBOUND,OUTBOUND>
```

The full signatures of the methods shown above are thus

```
protected abstract void encode(ChannelHandlerContext ctx,
       OUTBOUND msg, List<Object> out)

protected abstract void decode(ChannelHandlerContext ctx,
       INBOUND msg, List<Object> out)
```

That is, `encode()` handles the transformation of an outbound message of type `OUTBOUND` to `INBOUND` and `decode() does the reverse`. Where might we make use of such a codec?

In reality, this is a fairly common use case, often involving the conversion of data back and forth between two disctinct messaging APIs. This is often the case when we have to interoperate with an API that uses a legacy or proprietary message format.

Listing 7.7 shows how such a conversation might take place. In this case we have parameterized `MessageToMessageCode with an INBOUND` type of `WebSocketFrame` and an `OUTBOUND` type of `MyWebSocketFrame`, the latter being a static nested class of `WebSocketConvertHandler`.

Listing 7.7 `MessageToMessageCodec`

```
public class WebSocketConvertHandler extends
    MessageToMessageCodec<WebSocketFrame,
        WebSocketConvertHandler.MyWebSocketFrame> {
    @Override
    protected void encode(ChannelHandlerContext ctx,                          //1
        WebSocketConvertHandler.MyWebSocketFrame msg,
            List<Object> out) throws Exception {
      ByteBuf payload = msg.getData().duplicate().retain();
        switch (msg.getType()) {                                              //2
            case BINARY:
                out.add(new BinaryWebSocketFrame(payload));
                break;
            case TEXT:
                out.add(new TextWebSocketFrame(payload));
                break;
            case CLOSE:
                out.add(new CloseWebSocketFrame(true, 0, payload));
                break;
            case CONTINUATION:
                out.add(new ContinuationWebSocketFrame(payload));
                break;
            case PONG:
                out.add(new PongWebSocketFrame(payload));
                break;
            case PING:
                out.add(new PingWebSocketFrame(payload));
                break;
            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));
        }
        else if (msg instanceof CloseWebSocketFrame) {
            out.add(new MyWebSocketFrame (
                    MyWebSocketFrame.FrameType.CLOSE, payload));
```

```
        }
        else if (msg instanceof PingWebSocketFrame) {
            out.add(new MyWebSocketFrame (
                    MyWebSocketFrame.FrameType.PING, payload));
        }
        else if (msg instanceof PongWebSocketFrame) {
            out.add(new MyWebSocketFrame (
                    MyWebSocketFrame.FrameType.PONG, payload));
        }
        else if (msg instanceof TextWebSocketFrame) {
            out.add(new MyWebSocketFrame (
                    MyWebSocketFrame.FrameType.TEXT, payload));
        }
        else if (msg instanceof ContinuationWebSocketFrame) {
            out.add(new MyWebSocketFrame (
                    MyWebSocketFrame.FrameType.CONTINUATION, payload));
        }
        else {
        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` of the corresponding `FrameType`.**
3. **Decodes `WebSocketFrame` mssages to `MyWebSocketFrame` messages using the `instanceof` check to find the right `FrameType`.**
4. **The custom message type from which we want to encode to `WebSocketFrame` messages and to which we want to decode from `WebSocketFrame` messages.**
5. **An `enum` to indicate the type of the `MyWebSocketFrame`**

7.4.3 CombinedChannelDuplexHandler

As we mentioned earlier, combining a decoder and an encoder can impose a cost in terms of reusability. However, there is way to avoid this penalty without sacrificing the convenience of deploying a decoder and encoder as a logical unit to the `ChannelPipeline`.

The key is the following class:

```
public class CombinedChannelDuplexHandler
    <I extends ChannelInboundHandler,
    O extends ChannelOutboundHandler>
This class is parameterized by types that extend ChannelInboundHandler and
    ChannelOutboundHandler. This provides a container in which separate decoder
    and encoder classes can cooperate without obliging us to extend the
    abstract codec classes directly. We'll illustrate this in the following
    example. First examine the ByteToCharDecoder in Listing 7.8.
```

Listing 7.8 `ByteToCharDecoder`

```
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`**

The `decode()` method extracts two bytes at a time from the incoming data and writes them to the `List` as a `char`. (Notice that the implementation extends `ByteToMessageDecoder` because it reads chars from a `ByteBuf`.)

Now have a look at Listing 7.9, the encoder that converts chars back to bytes.

Listing 7.9 `CharToByteEncoder`

```
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 `char`s directly into the `ByteBuf`.

Now that we have a decoder and encoder, we combine them to build up a codec. Listing 7.10 shows how this is done using `CombinedChannelDuplexHandler`.

Listing 7.10 `CombinedByteCharCodec`

```
public class CombinedByteCharCodec extends
    CombinedChannelDuplexHandler<ByteToCharDecoder, CharToByteEncoder> {   //1
    public CombinedByteCharCodec() {
        super(new ByteToCharDecoder(), new CharToByteEncoder());        //2
    }
}
```

1. **`CombinedByteCharCodec` is parameterized by our decoder and encoder implementations to handle 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**

As you can see, it may be simpler and more flexible in some cases to combine implementations in this way than rather than using one of the codec classes. It may also come down to you personal preference or style.

7.5 Summary

In this chapter we studied the use of the Netty codec API to write decoders and encoders. We also learned why it is preferable to use this purpose rather than the plain `ChannelHandler` API.

We saw how the different abstract codec classes provide support for handling decoding and encoding in one implementation. On the other hand, if we need greater flexibility or wish to combine existing implementations we also have the option of combining them without needing to extend any of the abstract codec classes.

In the next chapter, we'll talk about the `ChannelHandler` implementations and codecs that are part of Netty itself that you can use out-of-the box to handle specific protocols and tasks.

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 delimited and length-based protocols
- Writing big data
- Serialization

Netty provides codecs and handlers for numerous common protocols that you can use practically "out of the box," reducing time otherwise spent on fairly tedious matters of infrastructure. In this chapter we'll explore these tools and their benefits. These include support for SSL/TLS, WebSockets and Google's SPDY, squeezing better performance out of HTTP with data compression.

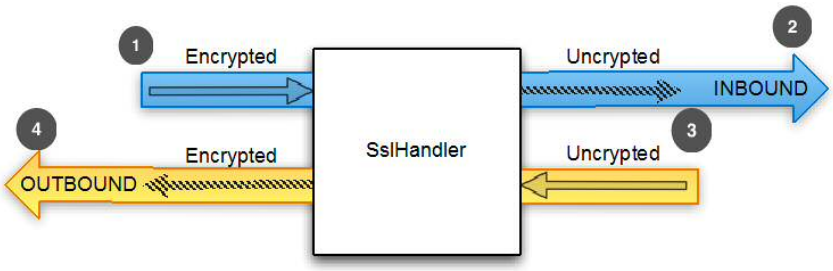
8.1 Securing Netty applications with SSL/TLS

Data privacy is a matter of great concern today and as developers we need to be prepared to address it. At a minimum we need to be familiar with encryption protocols such as SSL and TLS[16], which are layered on top of other protocols to implement data security. It's a pretty safe bet that you have encountered them as a user of an HTTPS website. Of course, these protocols are widely used in applications that are not HTTP-based, for example Secure SMTP (SMTPS) mail services and even relational database systems.

To support SSL/TLS, Java provides the `javax.net.ssl API`, whose classes `SslContext` and `SslEngine` make it relatively straightforward to implement decryption and encryption. Netty leverages this API by way of a `ChannelHandler` implementation named `SslHandler`, `which` employs an `SslEngine` internally to do the actual work.

Figure 8.1 shows a data flow using `SslHandler`.

[16] http://tools.ietf.org/html/rfc5246



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 an `SslHandler` is added to the `ChannelPipeline` using a `ChannelInitializer`. (Recall that `ChannelInitializer` is used 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. (If it is a client `startTls` should be used.)**
2. **Obtain a new `SslEngine` from the `SslContext`. Use a new `SslEngine` for each `SslHandler` instance**
3. **Set the client or server mode of the `SslEngine`**
4. **Add the `SslHandler` in the pipeline as first handler**

In most cases the `SslHandler` will be the first `ChannelHandler` in the `ChannelPipeline`. This ensures that encryption takes place only after all other `ChannelHandlers` have applied their logic to the data, thus securing their changes.

The `SslHandler` has some useful methods, as shown in Table 8.1. For example, during the handshake phase the two peers validate each other and agree upon an encryption method. You can configure `SslHandler` to modify its behavior or provide notification once the SSL/TLS handshake is complete, after which all data will be encrypted. The SSL/TLS handshake will be executed automatically.

Table 8.1 SslHandler methods

Name	Description

`setHandshakeTimeout(...)`
`setHandshakeTimeoutMillis(...)`
`getHandshakeTimeoutMillis()`
	Set and get the timeout, after which the handshake `ChannelFuture` is notified of failure.

`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 be 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`.

8.2 Building Netty HTTP/HTTPS applications

HTTP/HTTPS[17] is one of the most common protocol suites and with the success of smartphones it is more widely used with each passing day. While every company has a homepage that you can access via HTTP or HTTPS, this isn't its only use. Many organizations expose WebService APIS via HTTP(S), intended to be used with ease in a platform-independent manner.

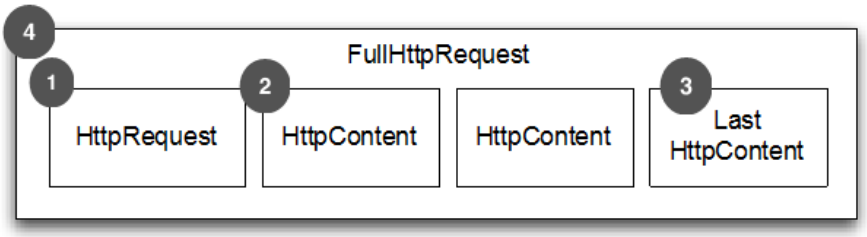
Let's have a look at the `ChannelHandlers` provided by Netty that allow you to use HTTP and HTTPS without having to write your own codecs.

8.2.1 HTTP Decoder, Encoder, and Codec

HTTP is based on a request–response pattern: the client sends an HTTP request to the server and the server sends back an HTTP response. Netty provides a variety of encoders and

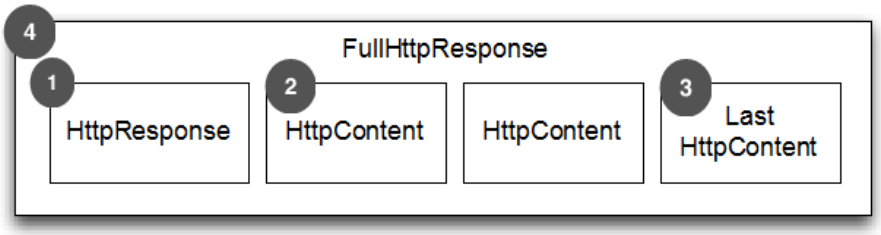
[17] http://www.w3.org/Protocols/rfc2616/rfc2616.html

decoders to simplify working with this protocol. Figures 8.2 and 8.3 show the methods for producing and consuming HTTP requests and responses, respectively.



1. **The first part of the HTTP Request contains headers**
2. **HttpContent contains data and may be followed by one or more HttpContent parts.**
3. **LastHttpContent subtype marks the end of the HTTP request and may also contain trailing headers**
4. **The full HTTP request**

Figure 8.2 HTTP request component parts



1. **The first part of the HTTP response contains headers**
2. **HttpContent part contains data and may be followed by one or more HttpContent parts**
3. **LastHttpContent subtype that marks the end of the HTTP response and may also contain trailing headers**
4. **The full HTTP response**

Figure 8.3 HTTP response component parts

As shown in figures 8.2 and 8.3 an HTTP request/response may consist of more than one data part, and it always terminates with a `LastHttpContent` part. The `FullHttpRequest` and `FullHttpResponse` message are special subtypes that represent a complete request and response, respectively. All types of HTTP messages (`FullHttpRequest`, `LastHttpContent`, and those shown in listing 8.2) implement the `HttpObject` interface.

Table 8.2 gives an overview of the HTTP decoders and encoders that handle and produce these 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 simple it is to add support for HTTP to your application. Merely add the correct `ChannelHandlers` to the `ChannelPipeline`.

Listing 8.2 Add support for HTTP

```
public class HttpPipelineInitializer
        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. **client: add an `HttpResponseDecoder` to handle responses from the server**
2. **client: add `HttpRequestEncoder` to send requests to the server**
3. **server: add `HttpRequestDecoder` to receive request from the client**
4. **server: add `HttpResponseEncoder` to send responses to the client**

8.2.2 HTTP message aggregation

After you have installed the initializer in the `ChannelPipeline`, you'll be able to operate on the different `HttpObject` messages. But since HTTP requests and responses can be comprised of many parts you'll need to aggregate them to form complete messages. To eliminate this cumbersome task Netty provides an aggregator, which merges message parts into

`FullHttpRequest` and `FullHttpResponse` messages. This way you always see the full message contents.

There is a slight cost to this operation since the message segments need to be buffered until complete messages can be forwarded to the next `ChannelInboundHandler` in the pipeline. But the benefit is that you don't have to be concerned about message fragmentation.

Introduction this automatic aggregation is just a matter of adding another `ChannelHandler` to `ChannelPipeline`. Listing 8.3 shows how this is done.

Listing 8.3 Automatically aggregate HTTP message fragments

```
public class HttpAggregatorInitializer extends ChannelInitializer<Channel> {
    private final boolean isClient;

    public HttpAggregatorInitializer(boolean isClient) {
        this.isClient = isClient;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        if (isClient) {
            pipeline.addLast("codec", new HttpClientCodec());           //1
        } else {
            pipeline.addLast("codec", new HttpServerCodec());           //2
        }
        pipeline.addLast("aggegator",
            new HttpObjectAggregator(512 * 1024));                      //3
    }
}
```

1. **client: add `HttpClientCodec`**
2. **server: add `HttpServerCodec` as we are in server mode**
3. **Add `HttpObjectAggregator` to the `ChannelPipeline`, using a max message size of 512kb.**

8.2.3 HTTP compression

When using HTTP it is often advisable to use compression to reduce as much as possible the size of transmitted data. While compression does have some cost in CPU cycles, it is a good idea most of the time, especially for text data.

Netty provides `ChannelHandler` implementations for compression and decompression that support both "gzip" and "deflate" encodings.

HTTP Request Header

The client can indicate supported encryption modes by supplying the header below. The server is not, however, obliged to compress the data it sends.

```
GET /encrypted-area HTTP/1.1
Host: www.example.com
Accept-Encoding: gzip, deflate
```

An example is shown in Listing 8.4.

Listing 8.4 Automatically compress HTTP messages

```
public class HttpAggregatorInitializer extends ChannelInitializer<Channel> {
    private final boolean isClient;

    public HttpAggregatorInitializer(boolean isClient) {
        this.isClient = isClient;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        if (isClient) {
            pipeline.addLast("codec", new HttpClientCodec());           //1
                pipeline.addLast("decompressor",
                        new HttpContentDecompressor());                 //2
        } else {
            pipeline.addLast("codec", new HttpServerCodec());           //3
                pipeline.addLast("compressor",
                        new HttpContentCompressor());                  //4
        }
    }
}
```

1. **client: add `HttpClientCodec`**
2. **client: add `HttpContentDecompressor` to handle compressed content from the server**
3. **server: `HttpServerCodec`**
4. **server: `HttpContentCompressor` to compress the data if the client supports it**

Compression and dependencies

Please note that if you are using Java 6 or earlier, to support compression you will need to add jzlib[18] to your classpath.

The Maven dependency:

```
<dependency>
    <groupId>com.jcraft</groupId>
    <artifactId>jzlib</artifactId>
    <version>1.1.3</version>
</dependency>
```

[18] http://www.jcraft.com/jzlib/

8.2.4 Using HTTPS

Listing 8.5 shows that enabling secure HTTP (HTTPS) is just a matter of adding an `SslHandler` to the mix.

Listing 8.5 Using HTTPS

```
public class HttpsCodecInitializer extends ChannelInitializer<Channel> {
    private final SSLContext context;
    private final boolean isClient;

    public HttpsCodecInitializer(SSLContext context, boolean isClient) {
        this.context = context;
        this.isClient = isClient;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        SSLEngine engine = context.createSSLEngine();
        engine.setUseClientMode(isClient);
        pipeline.addFirst("ssl", new SslHandler(engine));               //1

        if (isClient) {
            pipeline.addLast("codec", new HttpClientCodec());           //2
        } else {
            pipeline.addLast("codec", new HttpServerCodec());           //3
        }
    }
}
```

1. **Add SslHandler to the pipeline to use HTTPS**
2. **client: add HttpClientCodec**
3. **server: add HttpServerCodec as we are in server mode**

The above code is a good example of how Netty's architectural approach turns "reuse" into "leverage." We can add a new capability, even one as significant as encryption support, with almost no effort by simply adding a `ChannelHandler` to the `ChannelPipeline`.

8.2.5 WebSocket

Netty's extensive toolset for HTTP-based applications includes support for some of its most advanced features. In this section we'll explore WebSocket, a protocol standardized by the Internet Engineering Task Force in 2011.

WebSocket addresses a long-standing problem: how to publish information in real time when the underlying protocol, HTTP, is a sequence of request - response interactions. AJAX provides some improvement, but the flow of data is still driven by requests from the client side. There have been other more or less clever approaches[19], but in the end they remain workarounds with limited scalability.

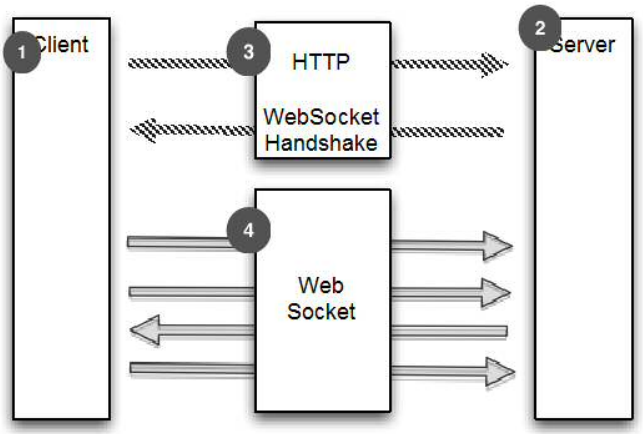
[19] http://en.wikipedia.org/wiki/Comet_%28programming%29

The WebSocket specification and its implementations represent an attempt at a more effective solution. Simply stated, a WebSocket provides

> *a single TCP connection for traffic in both directions. [...] Combined with the WebSocket API [...], it provides an alternative to HTTP polling for two-way communication from a web page to a remote server*[20].

That is, WebSockets provide true *bidirectional* exchange of data between client and server. We won't go into too much detail about the internals, but we should mention that while the earliest implementations were limited to text data this is no longer the case; a WebSocket can be used for any arbitrary data, much like a normal socket.

Figure 8.4 gives a general idea of the WebSocket protocol. In this scenario the communication starts as plain HTTP and "upgrades" to bidirectional WebSocket.



1. **Client (HTTP) communication with Server**
2. **Server (HTTP) communication with Client**
3. **Client issues WebSocket handshake via HTTP(s) and waits for acknowledgement**
4. **Connection protocol upgraded to WebSocket**

Figure 8.4 WebSocket protocol

Adding support for WebSocket to your application requires only adding the appropriate client-side or server-side WebSocket `ChannelHandler` to the pipeline. This class will handle the special message types defined by WebSocket, known as "frames." As shown in Table 8.3, these can be classed as "data" and "control" frames.

---

[20] "The WebSocket Protocol", http://tools.ietf.org/html/rfc6455, p.1

Table 8.3 WebSocketFrame types

Name	Description
`BinaryWebSocketFrame`	Data frame: binary data
`TextWebSocketFrame`	Data frame: text data
`ContinuationWebSocketFrame`	Data frame: text or binary data that belongs to a previous `BinaryWebSocketFrame` or `TextWebSocketFrame`
`CloseWebSocketFrame`	Control frame: a CLOSE request, close status code and a phrase
`PingWebSocketFrame`	Control frame: requests the send of a `PongWebSocketFrame`
`PongWebSocketFrame`	Control frame: sent as response to a `PingWebSocketFrame`

Since Netty is principally a server-side technology we'll focus here on creating a WebSocket server[21]. Listing 8.6 presents a simple example using `WebSocketServerProtocolHandler`. This class handles the protocol upgrade handshake as well as the three "control" frames - `Close`, `Ping` and `Pong`. `Text` and `Binary` data frames will be passed along to the next handlers (implemented by you) for processing.

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
    }

    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
```

[21] For client-side examples plase refer to the examples included in the Netty source code: https://github.com/netty/netty/tree/4.0/example/src/main/java/io/netty/example/http/websocketx/client

```
        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` to provide aggregated `HttpRequests` for the handshake**
2. **Add `WebSocketServerProtocolHandler` to handle the upgrade handshake if a request is sent to the endpoint "`/websocket.`" This will also handle the `Ping`, `Pong` and `Close` frames after the upgrade is complete.**
3. **`TextFrameHandler` will handle `TextWebSocketFrames`**
4. **`BinaryFrameHandler` will handle `BinaryWebSocketFrames`**
5. **`ContinuationFrameHandler` will handle `ContinuationWebSocketFrames`**

Secure WebSocket

To provide "secure" WebSockets, simply insert the `SslHandler` as the first `ChannelHandler` in the pipline.

For a more extensive example please see Chapter 11, which explores in depth the design of a real-time WebSocket application.

8.2.6 SPDY

SPDY[22] (SPeeDY) is one of the latest developments in Web technologies. Created by Google, it forms the basis of the HTTP 2.0 protocol currently under development by the IETF.

The primary goals of SPDY are to reduce the load time of web pages and improve web security. These are achieved by:

- Compressing headers
- Encrypting everything
- Multiplexing connections
- Providing support for different transfer priorities

SPDY is a versioned protocol. To date there have been five versions:

- 1 – Initial version, not in use

[22] http://www.chromium.org/spdy

- 2 – New features including server-push
- 3 – New features including flow control and updated compression
- 3.1 – Session-layer flow control
- 4.0 – Stream flow control and more integration with HTTP 2.0

At the time of writing SPDY is supported by numerous browsers including Google Chrome, Firefox, and Opera.

Netty provides support for Versions 2 and 3 (including 3.1). These are are currently the most widely used and should enable you to support most users. For an example of how to use SPDY with Netty please see Chapter 12, which is entirely devoted to this topic.

8.3 Idle connections and Timeouts

So far our discussion of HTTP has focused on Netty's support for some of its variants including HTTPS, WebSocket and SPDY, via specialized codecs and handlers. These technologies can make our web applications more effective, usable and secure. However, the sophistication of these tools will not serve us well if we don't manage our actual network resources efficently. So let's talk about connection management.

Detecting idle connections and timeouts is essential to freeing up resources in a timely manner. A common method for testing an inactive connection is to send a message, usually referred to as a "heartbeat," to the remote peer to determine whether it is still alive. (A more radical approach is simply to disconnect the remote peer after a specified interval of inactivity.)

Dealing with idle connections is such a common task that Netty provides several `ChannelHandler` implementations just for this purpose. Table 8.4 gives an overview.

Table 8.4 ChannelHandlers for idle connections and timeouts

Name	Description
`IdleStateHandler`	fires an `IdleStateEvent` if the connection idles too long. You can then handle the `IdleStateEvent` by overriding the `userEventTriggered(…)` method in your `ChannelInboundHandler`.
`ReadTimeoutHandler`	throws a `ReadTimeoutException` and closes the `Channel` when no inbound data is received for a specified interval. The `ReadTimeoutException` can be detected by overriding the `exceptionCaught(…)` method of your `ChannelHandler`.
`WriteTimeoutHandler`	throws a `WriteTimeoutException` and closes the `Channel` when no inbound data is received for a specified interval. The `WriteTimeoutException` can be detected by overriding the `exceptionCaught(…)` method of your `ChannelHandler`.

Let's take a closer look at `IdleStateHandler`, the one most used in practice. Listing 8.7 shows how you can use the `IdleStateHandler` to get notification if no data have been received or

sent data for 60 seconds. In such a case, a heartbeat is written to the remote peer. If there is no response the connection is closed.

Listing 8.7 Sending heartbeats

```
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. **`IdleStateHandler` will call `userEventTriggered` with an `IdleStateEvent` if the connection has not received or sent data for 60 seconds**
2. **The heartbeat to send to the remote peer**
3. **Send the heartbeat and add a listener that will close the connection if the send operation fails**
4. **The event is not an `IdleStateEvent` so pass it to the next handler**

To summarize, this example illustrates how to employ `IdleStateHandler` to test whether the remote peer is still alive and to free up resources by closing the connection if it is not.

8.4 Decoding delimited and length-based protocols

As you work with Netty you will encounter delimited and length-based protocols that require decoders. This section explains the implementations that Netty provides to handle these cases.

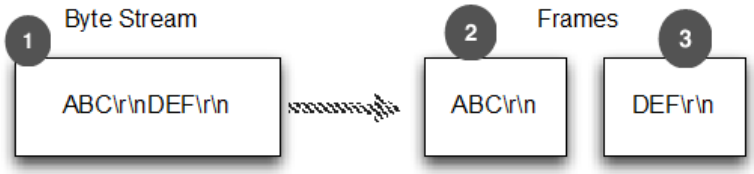
8.4.1 Delimited protocols

Protocols specified by RFC's are often either delimited or extensions of other delimited protocols. SMTP[23], POP3[24], IMAP[25], and Telnet[26] are among the best-known of these. The decoders listed in Table 8.5 make it easy to define custom decoders that can extract frames delimited by any arbitrary sequence of tokens.

Table 8.5 Decoders for handling delimited and length-based protocols

Name	Description
`DelimiterBasedFrameDecoder`	a generic decoder that extracts frames using any user- provided delimiter
`LineBasedFrameDecoder`	a decoder that extracts frames delimited by the line-endiings "`\n`" or "`\r\n`" Note: faster than `DelimiterBasedFrameDecoder`.

Figure 8.5 shows how frames are handled when delimited by the end-of-line sequence "`\r\n`" (carriage return + line feed).



1. **byte stream**
2. **first frame**
3. **second frame**

Figure 8.5 Handling delimited frames

Listing 8.8 shows how to use the `LineBasedFrameDecoder` to handle the case shown above.

Listing 8.8 Handling line-delimited frames

```
public class LineBasedHandlerInitializer extends ChannelInitializer<Channel> {
    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
```

[23] http://www.ietf.org/rfc/rfc2821.txt
[24] http://www.ietf.org/rfc/rfc1939.txt
[25] http://tools.ietf.org/html/rfc3501
[26] http://tools.ietf.org/search/rfc854

```
        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,              //3
                ByteBuf msg) throws Exception {
            // Do something with the data extracted from the frame
        }
    }
}
```

1. **Add a `LineBasedFrameDecoder` to extract the frames and forward them to the next handler in the pipeline, in this case a `FrameHandler`**
2. **Add the `FrameHandler` to receive the frames**
3. **Each call passes the contents of a single frame**

If you are working with frames delimited by something other than line endings, you can use the `DelimiterBasedFrameDecoder` in a similar fashion - you only need to specify the specific delimiter sequence to the constructor.

These decoders are your tools for implementing your own delimited protocols. Consider the following case:

- The incoming data stream is a series of frames, each delimited by a line feed (`"\n"`).
- Each frame consists of a series of items, each delimited by a single space character.
- The contents of a frame represent a "command": a name followed by a variable number of arguments.

The implementation shown in Listing 8.9. defines the following classes:

- `class Cmd` - stores the contents of the frame in one `ByteBuf` for the name and another for the arguments
- `class CmdDecoder` - retrieves a line from the overridden `decode()` method and constructs a `Cmd` instance from its contents
- `class CmdHandler` - receives the decoded `Cmd` object from the `CmdDecoder` and performs some processing on it.

As you can see, the key to this custom protocol decoder is the extension of `LineBasedFrameDecoder`.

Listing 8.9 Decoder for the command and the handler

```
public class CmdHandlerInitializer extends ChannelInitializer<Channel> {
    final byte SPACE = (byte)' ';
    @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(), SPACE);                                 //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 a `CmdDecoder` to the pipeline; it will extract a `Cmd` object and forward it to the next handler in the pipeline.**
2. **Add the `CmdHandler` that will receive and process the `Cmd` objects.**
3. **the POJO that represents a command**
4. **`super.decode()` extracts a frame from the `ByteBuf` delimited by an end-of-line sequence.**
5. **No frame in the input so return `null`**
6. **Find the index of the first space character. What precedes it is the command name; what follows it is the sequence of arguments.**
7. **Instantiate a new `Cmd` object from the slice of the frame that precedes the index and the one that follows it.**
8. **Handle the `Cmd` object passed through the pipeline.**

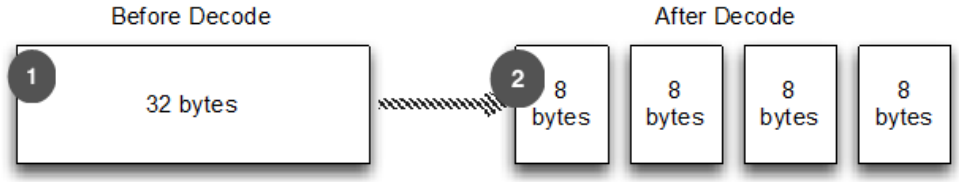
8.4.2 Length-based protocols

A length-based protocol defines a frame by encoding its length in a header segment of the frame itself rather than by mark its end with a special delimiter. Table 8.6 lists the two decoders Netty provides for handling this type of protocol.

Table 8.6 Decoders for length-based protocols

Name	Description
`FixedLengthFrameDecoder`	extracts frames of a fixed size, specified when the constructor is called.
`LengthFieldBasedFrameDecoder`	extracts frames based on a length value encoded in a field in the frame header; the offset and length of the field are specified in the constructor.

Figure 8.6 shows the operation of a `FixedLengthFrameDecoder` that has been constructed with a frame length of "8".

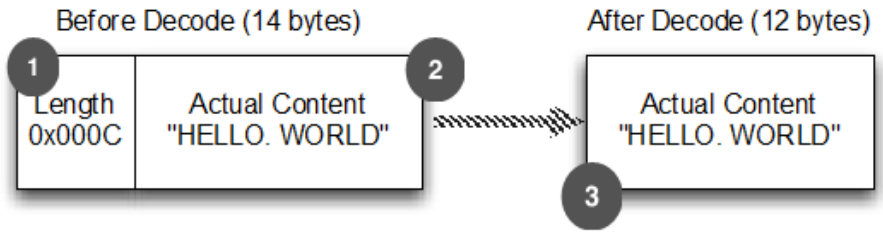


1. **byte stream**
2. **4 extracted frames of 8 bytes each**

Figure 8.6 Decoding a frame length of 8 bytes

It is more common to encounter cases where the size of the frame is encoded in the header. For this purpose you can use the `LengthFieldBasedFrameDecoder`, which determines the frame length from the specified header field and extract the specified number of bytes from the data stream.

Figure 8.7 shows an example where the length field in the header is at offset 0 and has a length of 2 bytes.



1. **Length "0x000C" (12) is encoded in the first two bytes of the frame**
2. **The last 12 bytes has the contents**
3. **The extracted frame with the contents and without the header**

Figure 8.7 Message that has frame size encoded in the header

The `LengthFieldBasedFrameDecoder` provides several constructors to cover a variety of header length field configuration cases. Listing 8.10 shows the use of a constructor whose three arguments are `maxFrameLength`, `lengthFieldOffset` and `lengthFieldLength`. In this case, the length of the frame is encoded in the frame's first 8 bytes.

Listing 8.10 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 a `LengthFieldBasedFrameDecoder` to extract frames based on the encoded length in the first 8 bytes of the frame.**
2. **Add a `FrameHandler` to handle each frame.**
3. **Do something with the frame data.**

To summarize, this section explored the codecs Netty provides to support protocols that define the structure of the byte stream by specifiying delimiters or the length of a protocol frame. These codecs will be useful to you as a great many common protocols fit into one or the other of these categories.

8.5 Writing big data

Writing big chunks of data efficiently is a special problem in asynchronous frameworks because of the possibility of network saturation. Since the write operations are non-blocking they return even if the data can not be written out and just notify the `ChannelFuture` once it is done. When this occurs you have to stop writing or risk running out of memory. So when writing large masses of data we need to be prepared to handle cases where a slow connection to the remote peer can cause a delays in freeing memory. As an example let's consider writing the contents of a file to the network.

In our discussion of transports (see Section 4.2) we mentioned the "zero-copy" feature of NIO that eliminates copying steps in moving the contents of a file from the file system to the network stack. All of this happens in Netty's core, so all that is required at the level of application code is to use an implementation of `interface FileRegion`, defined in the Netty API documentation as

> a region of a file that is sent via a `Channel` that supports zero-copy file transfer.

Listing 8.11 shows how to transmit a file's contents using zero-copy by creating a `DefaultFileRegion` from a `FileInputStream` and writing it to a `Channel`.

Listing 8.11 Transferring file contents 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`**
2. **Create a new `DefaultFileRegion` for the full length of the file**
3. **Send the `DefaultFileRegion` and register a `ChannelFutureListener`**
4. **Handle send failure**

The example just seen applies only to the direct transmission of a file's contents, with no processing of the data performed by the application. In the contrary case, where copying of the data from the file system into user memory is required, you can use `ChunkedWriteHandler`. This class provides support for writing a large data stream asynchronously without incurring high memory consumption.

The key is `interface ChunkedInput`. Each of the implementations provided, listed in Table 8.7, represent a data stream of indefinite length, which is then consumed by `ChunkedWriteHandler`.

Table 8.7 ChunkedInput implementations

Name	Description
`ChunkedFile`	fetches data from a file chunk by chunk, for use when your platform doesn't support zero-copy or you need to transform the data
`ChunkedNioFile`	similar to `ChunkedFile` except that it uses `NIOFileChannel`
`ChunkedStream`	transfers content chunk by chunk from an `InputStream`.
`ChunkedNioStream`	transfers content chunk by chunk from a `ReadableByteChannel`

Listing 8.12 illustrates the use of `ChunkedStream`, the implementation most used in practice. The class shown is instantiated with a `File` and an `SslContext`. When `initChannel()` is called it initializes the channel with the chain of handlers shown.

When the channel becomes active the `WriteStreamHandler` will write data from the file chunk by chunk as a `ChunkedStream`. Finally the data will be encrypted by the `SslHandler` before being transmitted.

Listing 8.12 Transfer file content with `FileRegion`

```
public class ChunkedWriteHandlerInitializer
    extends ChannelInitializer<Channel> {
    private final File file;
    private final SslContext sslCtx;

    public ChunkedWriteHandlerInitializer(File file, SslContext sslCtx) {
        this.file = file;
       this.sslCtx = sslCtx;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
      ChannelPipeline pipeline = ch.pipeline();
      pipeline.addLast(new SslHandler(sslCtx.createEngine());          //1
      pipeline.addLast(new ChunkedWriteHandler());                             //2
      pipeline.addLast(new WriteStreamHandler());                              //3
    }

    public final class WriteStreamHandler
      extends ChannelInboundHandlerAdapter {

      @Override
      public void channelActive(ChannelHandlerContext ctx)             //4
          throws Exception {
          super.channelActive(ctx);
          ctx.writeAndFlush(
        new ChunkedStream(new FileInputStream(file)));
       }
```

```
    }
}
```

1. **Add an `SslHandler` to the `ChannelPipeline`.**
2. **Add a `ChunkedWriteHandler` to handle data passed in as `ChunkedInput`.**
3. **The `WriteStreamHandler` starts to write the contents of the file once the connection is established.**
4. **`channelActive()` triggers writing the contents of the file using `ChunkedInput` when the connection is established. (The `FileInputStream` is shown for illustration; any `InputStream` can be used).**

ChunkedInput

All that is required to use your own `ChunkedInput` implementations is to install a `ChunkedWriteHandler` in the pipeline.

In this section we discussed

- how to transfer files efficiently by using the zero-copy feature
- how to write large data without risking `OutOfMemoryErrors` by using `ChunkedWriteHandler`.

In the next section we will examine several different approaches to serializing POJOs.

8.6 Serializing data

The JDK provides `ObjectOutputStream` and `ObjectInputStream` for serializing and deserializing primitive data types and graphs of POJOs over the network. The API is not complex and can be applied to any object that supports the `java.io.Serializable` interface. It is also not terribly performant. In this section we'll see what Netty has to offer.

8.6.1 JDK Serialization

If your application has to interact with peers that use `ObjectOutputStream` and `ObjectInputStream` and compatibility is your primary concern then JDK serialization[27] is the right choice.

Table 8.8 lists the serialization classes that Netty provides for interoperating with the JDK.

[27] http://docs.oracle.com/javase/7/docs/technotes/guides/serialization/

Table 8.8 JDK Serialization codecs

Name	Description
`CompatibleObjectDecoder`	Decoder for interoperating with non-Netty peers that use JDK serialization.
`CompatibleObjectEncoder`	Encoder for interoperating with non-Netty peers that use JDK serialization.
`ObjectDecoder`	Uses custom serialization for decoding on top of JDK Serialization. Provides a speed improvement when external dependencies are excluded. Otherwise the other serialization implementations are preferable.
`ObjectEncoder`	Uses custom serialization for encoding on top of JDK Serialization. Provides a speed improvement when external dependencies are excluded. Otherwise the other serialization implementations are preferable.

8.6.2 Serialization with JBoss Marshalling

If you are not restricted in the use of external dependencies, JBoss Marshalling is ideal. It is up to three times faster than JDK Serialization and more compact.

> JBoss Marshalling is an alternative serialization API that fixes many of the problems found in the JDK serialization API while remaining fully compatible with `java.io.Serializable` and its relatives, and adds several new tunable parameters and additional features, all of which are pluggable via factory configuration (externalizers, class/instance lookup tables, class resolution, and object replacement, to name a few).[28]

Netty supports JBoss Marshalling with the two decoder / encoder pairs shown in Table 8.9. The first set is compatible with peers that use only JDK Serialization while the second, providing maximum performance, is for use only with peers that use JBoss Marshalling.

Table 8.9 JBoss Marshalling codecs

Name	Description
`CompatibleMarshallingDecoder`	For compatibility with peers that use JDK serialization.
`CompatibleMarshallingEncoder`	For compatiblity with peers that use JDK serialization.
`MarshallingDecoder`	Uses custom serialization for decoding, must be used with `MarshallingEncoder`.
`MarshallingEncoder`	Uses custom serialization for encoding, must be used with `MarshallingDecoder`

[28] https://www.jboss.org/jbossmarshalling

Listing 8.13 shows how to use `MarshallingDecoder` and `MarshallingEncoder`. Again, it is mostly a matter of configuring the `ChannelPipeline` appropriately.

Listing 8.13 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 of Netty's solutions for serialization is a codec that utilizes ProtoBuf,[29] a data interchange format developed by Google and now open-sourced.

ProtoBuf encodes and decodes structured data in a way that is both compact and efficient. It has bindings for all sorts of programming languages, making it a good fit for cross-language projects.

Table 8.10 shows the `ChannelHandler` implementations Netty supplies for ProtoBuf support.

[29] https://code.google.com/p/protobuf/

Table 8.10 ProtoBuf codec

Name	Description
`ProtobufDecoder`	decodes a message using ProtoBuf
`ProtobufEncoder`	encodes a message using ProtoBuf
`ProtobufVarint32FrameDecoder`	splits received `ByteBufs` dynamically by the value of the Google Protocol "Base 128 Varints"[30] integer length field in the message.

Here again, using ProtoBuf is just a matter of adding the right `ChannelHandler` to the `ChannelPipeline`, as shown in Listing 8.14.

Listing 8.14 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
        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 we explored the different serialization options supported by Netty's specialized decoders and encoders. These are the standard JDK serialization API, JBoss Marshalling and Google ProtoBuf.

[30] https://developers.google.com/protocol-buffers/docs/encoding

8.7 Summary

The codecs and handlers provided by Netty can be combined and extended to implement a very broad range of processing scenarios. Furthermore, they are proven and robust components that have been employed in many large systems.

Please note we have covered only the most common examples. The API documents provide complete coverage.

9

Bootstrapping

This chapter covers

- Bootstrapping clients and servers
- Bootstrapping clients from within a Channel
- Adding ChannelHandlers
- Using ChannelOptions and attributes

As we have seen, `ChannelPipelines`, `ChannelHandlers` and codecs provide tools with which we can handle a broad range of data processing requirements. But you may well ask, "Once I have created my components, how do I assemble them to form a working application?"

The answer is "bootstrapping." Up to now we have used the term somewhat vaguely and the time has come to define it. In the simplest terms, bootstrapping is the process of configuring an application. But as we shall see, there is more to it than just that; Netty's classes for bootstrapping clients and servers insulate your application code from the network infrastructure, connecting and enabling all of the components in the background.

In short, bootstrapping is the missing piece of the puzzle - when you put it in place your Netty application will be complete.

9.1 Bootstrap types

Netty includes two different types of bootstraps. Rather than just thinking of them as "server" and "client" bootsraps, it is more useful to consider the application functions they are intended to support. In this sense, "servers" are applications that devote a "parent" channel to accepting connections and create "child" channels for them, while a "client" will most likely require only a single, non-"parent" channel for all network interactions[31].

As shown in Figure 9.1, the two bootstrap implementations extend from one superclass named `AbstractBootstrap`.

[31] This is also true of connectionless transports such as UDP, which don't require a channel-per-connection or "child channels."

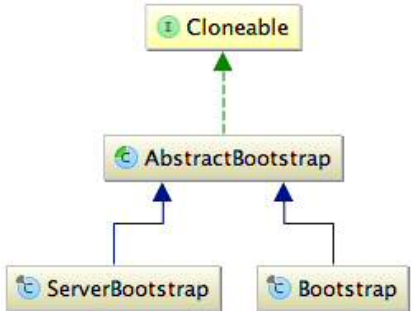


Figure 9.1 Bootstrap hierarchy

Many of the topics we've covered in previous chapters are common concerns that apply equally to clients and servers. These are handled by `AbstractBootstrap`, thus preventing duplication of functionality and code. The specialized bootstrap classes can then focus entirely on their distinct areas of concern.

Cloning Bootstraps

We frequently need to create multiple channels with similar or identical settings. To support this pattern without requiring the creation and configuration of a new bootstrap instance for each channel, `AbstractBootstrap` has been marked `Cloneable`[32]. Calling `clone()` on an already configured bootstrap will return another bootstrap instance which is immediately usable.

Note that since this creates only a shallow copy of the bootstrap's `EventLoopGroup`, the latter will be shared among all of the cloned channels. This is acceptable, as the cloned channels are often short-lived[33].

The rest of this chapter will focus on `Bootstrap` and `ServerBootstrap`, starting with the former as it is the less complex of the two.

9.2 Bootstrapping clients and connectionless protocols

Bootstrapping a client or an application that uses a connectionless protocol will employ the `Bootstrap` class. In this section we'll review the various methods available for bootstrapping clients, the bootstrapping process, and the available channel implementations.

[32] http://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html
[33] A typical case is a channel created to make an HTTP request.

9.2.1 Client bootstrapping methods

Table 9.1 gives an overview of the methods of the `Bootstrap` class. Note that many of these are inherited from `AbstractBootstrap`.

Table 9.1 `Bootstrap` methods

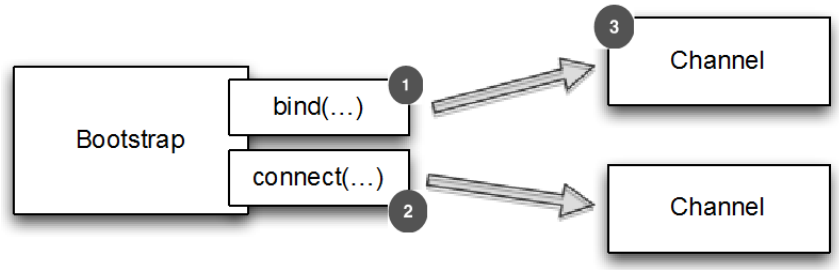
Name	Description
`group`	sets the `EventLoopGroup` that will handle all events for the `Channel`.
`channel` `channelFactory`	`channel()` specifies the `Channel`.implementation class. If the class doesn't provide a default constructor, you can call `channelFactory()` to specify a factory class to be called by `bind()`.
`localAddress`	specifies the local address the `Channel` should be bound to. If not provided, a random one will be created by the operating system. Alternatively, you can specify the `localAddress` with `bind()` or `connect()`
`option`	sets a `ChannelOption` to apply to the `ChannelConfig`.of a newly created `Channel`. Those options will be set on the channel by `bind` or `connect`, depending on which is called first. This method has no effect after channel creation. The `ChannelOption`s supported depend on the channel type used . Please refer to Section 9.6 and to the API docs of the `ChannelConfig` for the `Channel` type used.
`attr`	Specifies an attribute of a newly created `Channel`. These are set on the channel by `bind` or `connect`, depending on which is called first. This method has no effect after channel creation. Please refer to Section 9.6.
`handler`	sets the `ChannelHandler` that is added to the `ChannelPipeline` to receive event notification.
`clone`	creates a clone of the current `Bootstrap` with the same settings as the original.
`remoteAddress`	sets the remote address. Alternatively, you can specify it with `connect()`.
`connect`	Connect to the remote peer and return a `ChannelFuture`, which is notified once the connection operation is complete.
`bind`	Bind the channel and return a `ChannelFuture`, which is notified once the bind operation is complete, after which `Channel.connect()` must be called to establish the connection.

The next section will provide a step-by-step explanation of client bootstrapping.

9.2.2 How to bootstrap a client

The `Bootstrap` class is responsible for creating channels for clients or applications that utlilize connectionless protocols and does so after `bind()` or `connect()` is called.

Figure 9.2 shows how this works.



1. **`Bootstrap` will create a new channel when `bind()` is called, after which `connect()` is called on the `Channel` to establish the connection.**
2. **`Bootstrap` will create a new channel when `connect()` is called.**
3. **The new `Channel`.**

Figure 9.2 Bootstrap process

Listing 9.1 illustrates bootstrapping a client that uses the NIO TCP transport.

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("Received 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 and connec tnew client channels.**
2. **Specify the `EventLoopGroup`.**
3. **Specify the `Channel` implementation to be used.**
4. **Set the handler for `Channel` events and data.**
5. **Connect to the remote host**

Note that `Bootstrap` provides a "fluent" syntax - the methods used in the example (except for `connect()`) are chained by the reference they return to the `Bootstrap` instance itself.

9.2.3 Compatibility

The `Channel` implementation and the `EventLoop` that are processed by the `EventLoopGroup` must be compatible (please see the the API docs for details). Note that the compatible pairs of `EventLoop`s and `EventLoopGroup`s in defined in the same package as the `Channel` implementation itself.

For example, `NioEventLoop`, `NioEventLoopGroup`, and `NioServerSocketChannel` are to be used together - all are in fact prefixed with "Nio". You can't mix components having different prefixes, for example `OioEventLoopGroup` and `NioServerSocketChannel`.

EventLoop and EventLoopGroup

Remember that the `EventLoop` assigned to the `Channel` is responsible for handling all the operations for the `Channel`. 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 assigns an `EventLoop` to the `Channel` when it is registered.

We will cover this topic in greater detail in Chapter 15.

Listing 9.2 shows the result of trying to use a `Channel` type with an compatible `EventLoopGroup`.

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**
2. **Register the `EventLoopGroup` that will be used to get `EventLoops`**
3. **Specify the `Channel` class to be used. Notice that we are using the NIO version for the `EventLoopGroup` and OIO for the `Channel`**
4. **Set a handler for channel I/O events and data**

5. **Try to connect to the remote peer. This will throw an `IllegalStateException` as `NioEventLoopGroup` isn't compatible with `OioSocketChannel`**

The `IllegalStateException` is 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
      at
      io.netty.channel.AbstractChannel$AbstractUnsafe.register(AbstractChannel.java:5
      71)
...
```

Other situations where an `IllegalStateException` can be thrown include failing to call methods that set parameters required before `bind()` or `connect()` can be called, namely:

- `group()`
- `channel()` or `channnelFactory()`
- `handler()`

The `handler()` method is particularly important as the `ChannelPipeline` needs to be configured appropriately.

Once these parameters have been provided, your application is set to make full use of the Netty's capabilities.

9.3 Bootstrapping servers

Server bootstrapping shares some logic with client bootstrapping. As before, we'll start by outlining the API and follow this with a description of the steps involved.

9.3.1 Methods for bootstrapping servers

Table 9.2 lists the methods of `ServerBootstrap`.

Table 9.2 Methods of ServerBootstrap

Name	Description
`group`	sets the `EventLoopGroup` to be used by the `ServerBootstrap`. This `EventLoopGroup` serves the I/O of the `ServerChannel` and accepted `Channels`.
`channel` `channelFactory`	sets the class of the `ServerChannel` to instantiate. If the channel can't be created via a default constructor, you can provide a `ChannelFactory`.
`localAddress`	specifies 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` with `bind()` or `connect()`
`option`	specifies a `ChannelOption` to apply to the `ChannelConfig of a newly created ServerChannel`. Those options will be set on the channel by `bind()` or `connect()`, depending on which is called first. Setting or changing a `ChannelOption` after those

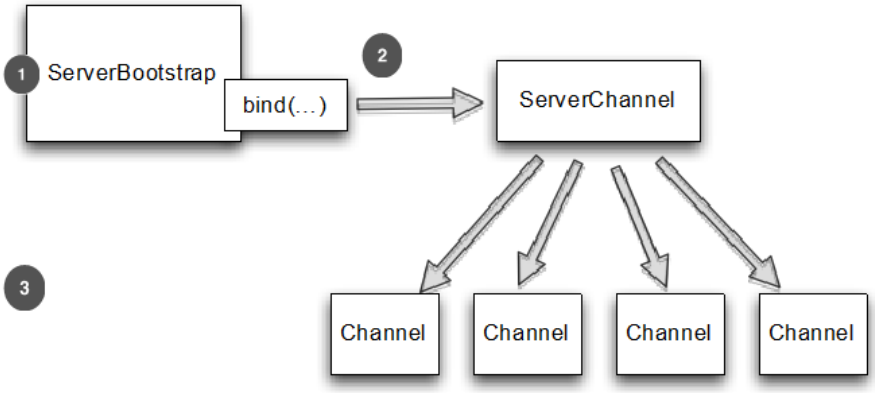
	methods have been called has no effect. Which `ChannelOptions` are supported depends on the actual channel type used. Please refer to the API docs of the `ChannelConfig` used.
`childOption`	`Specifies a ChannelOption` to apply to a `Channel's ChannelConfig`. when the channel has been accepted. Which `ChannelOptions` are supported depends on the actual channel type used. Please refer to the API docs of the `ChannelConfig` used.
`attr`	specifies an attribute on the `ServerChannel`. Attributes will be set on the channel by `bind()`. Changing them after calling `bind()` has no effect.
`childAttr`	applies an attribute to accepted `Channels`. Subsequent calls have no effect.
`handler`	sets the `ChannelHandler` that is added to the `ChannelPipeline` of the `ServerChannel`. However, `childHandler()` is used more frequently - please see the description of `childHandler()`.
`childHandler`	sets the `ChannelHandler` that is added to the `ChannelPipeline` of accepted `Channels`. The difference between `handler()` and `childHandler()` is that the former adds a handler which is processed by the *accepting* `ServerChannel`, while `childHandler()` adds a handler which is processed by the *accepted* `Channel`. The latter represents a socket bound to a remote peer.
`clone`	Clone the `ServerBootstrap` for connecting to a different remote peer with settings identical to those of the original `ServerBoostrap`.
`bind`	Bind the `ServerChannel` and return a `ChannelFuture`, which is notified once the connection operation is complete (with success or error result).

The next section explains the steps involved in server bootstrapping.

9.3.2 How to bootstrap a server

You may have noticed that Table 9.2 lists several methods not present in Table 9.1. These methods, `childHandler()`, `childAttr()` and `childOption()`, have been added to `ServerBootstrap` to support operations that are typical of server applications. Specifically, `ServerChannel` implementations are responsible for creating child `Channels`, which represent accepted connections. Therefore `ServerBootstrap`, which bootstraps `ServerChannel`s, provides these methods to simplify the task of applying settings to the `ChannelConfig` member of an accepted `Channel`.

Figure 9.3 shows `ServerBootstrap` creating the `ServerChannel` on `bind()` and the latter managing a number of child `Channels`.



1. **`ServerBootstrap` 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 a child channel for each one.**
3. **`Channels` for accepted connections**

Figure 9.3 ServerBootstrap

Remember that the `child*` methods will operate on the child `Channels`, which are managed by the `ServerChannel`.

In Listing 9.4 `ServerBootstrap` will create a `NioServerSocketChannel` instance when `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.**
4. **Set a child handler to handle I/O and data for the accepted channels.**
5. **Bind the channel via with the configured bootstrap**

9.4 Bootstrapping clients from a Channel

On occasion you may need to bootstrap a client `Channel` from another `Channel`. This can happen if you are writing a proxy or need to retrieve data from other systems. The latter case is common since many Netty applications integrate with an organization's existing systems, such as web services or databases.

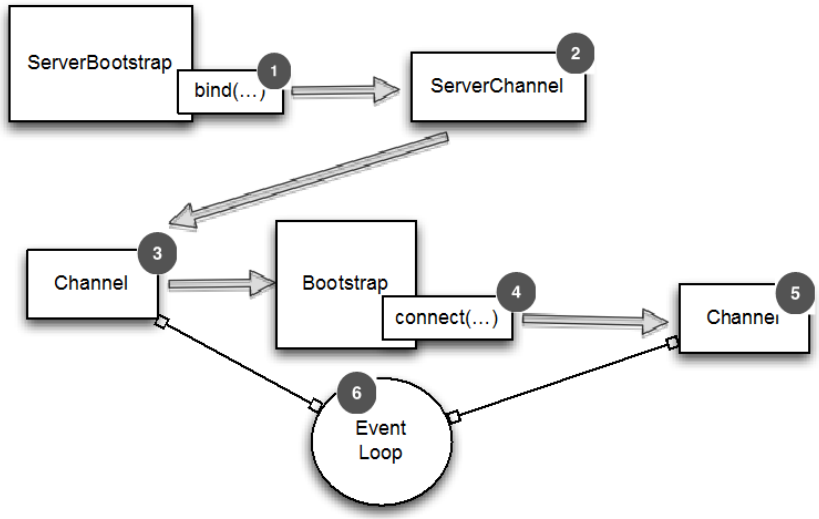
While you could of course create a new `Bootstrap` and use it as described in section 9.2.1, this solution is not as efficient as it could be. At a minimum, you will have to create another `EventLoop` for the new client `Channel`, and the exchange of data between the accepted `Channel` and the client `Channel` will require context-switching between `Threads`.

Fortunately, because `EventLoop` extends `EventLoopGroup`, you can pass the `EventLoop` of the accepted `Channel` to the `group()` method of the `Bootstrap`. This allows the client `Channel` to operate on the same `EventLoop` and eliminates all the extra thread creation and related context switching.

Why share the EventLoop ?

When you share an `EventLoop` you are guaranteed that all `Channel`s assigned to the `EventLoop` will use the same thread, eliminating context-switching and the related overhead. (Remember that an `EventLoop` is assigned to a `Thread` that executes its operations.)

Sharing of an EventLoop is depicted in figure 9.4.



1. **`ServerBootstrap` creates a new `ServerChannel` when `bind()` is called. After successful `bind()` this channel will accept child channels.**
2. **`ServerChannel` accepts new connections and creates child channels to serve them.**
3. **`Channel` for an accepted connection**
4. **`Bootstrap` created by the channel itself to create a new channel when `connect()` is called.**
5. **New channel connected to the remote peer**
6. **`EventLoop` shared between the channel that was created after the accept and the new one that was created by connect()**

Figure 9.4 EventLoop shared between channels with ServerBootstrap and Bootstrap

Implementing `EventLoop` sharing involves setting the `EventLoop` on the bootstrap via the `Bootstrap.eventLoop()` method. This is shown in Listing 9.5.

Listing 9.5 Bootstrapping a server

```
ServerBootstrap bootstrap = new ServerBootstrap();                    //1
bootstrap.group(
    new NioEventLoopGroup(),                                          //2
    new NioEventLoopGroup())
    .channel(NioServerSocketChannel.class)                            //3
    .childHandler(                                                    //4
        new SimpleChannelInboundHandler<ByteBuf>() {
            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("Received 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("Bind 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 to be used**
4. **Set a handler to handle I/O and data for 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 assigned to the accepted channel**
9. **Connect to remote peer**
10. **Do something with the data if connection is complete (for example, proxy)**
11. **Bind the channel via configured `Bootstrap`**

Note that a new `EventLoop` will use a new `Thread`. For this reason, `EventLoop` instances should be reused wherever possible. If this is not possbile, try to limit the number of instances created in order to avoid exhausting system resources.

9.5 Adding multiple ChannelHandlers during a bootstrap

In all of the code examples shown we added only one `ChannelHandler` during the bootstrap process with `handler()` or `childHandler()`. While this may be sufficient for simple applications it will not meet the needs of more complex ones. For example, in an application that must support multiple protocols such as HTTP or WebSockets, you will usually end up with many different `ChannelHandler`s in your `ChannelPipeline`. Attempting to manage all these protocols in one handler would result in a large and complex class. As we have seen repeatedly, you can deploy as many `ChannelHandlers` as you require by installing them in a chain in a `ChannelPipeline`. But how are we to do this if you can set only one `ChannelHandler` during the bootstrapping process? The answer is simple. Indeed, we'll use only one `ChannelHandler`, but it's a particular one . . .

For exactly this use case Netty provides a special abstract base class called `ChannelInitializer`, a subclass of `ChannelInboundHandlerAdapter`, which provides a way to add `ChannelHandlers` to the `ChannelPipeline` of a `Channel`. Your implementation will be called to do its work once the channel is registered to its `EventLoop`, following which it will remove itself from the `ChannelPipeline`.

Listing 9.6 shows that this apparently complex operation is actually quite straightforward.

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 and bind new `Channels`**
2. **Specify the `EventLoopGroup`s which will be used to provide `EventLoop`s from and register with the `ServerChannel` and the accepted channels**
3. **Specify the `Channel` class**
4. **Set a handler for I/O and data for the accepted channels**
5. **Bind the channel via the configured bootstrap**
6. **`ChannelInitializer` which is responsible for setting up the `ChannelPipeline`**
7. **Implement `initChannel()` to add the required handlers to the `ChannelPipeline`. Once this 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`s into the `ChannelPipeline` as your application requires.

9.6 Using Netty ChannelOptions and Attributes

It would be tedious to have to manually configure every channel when it's created. In fact, you don't have to; instead, you apply `ChannelOptions` to a bootstrap with the `option()` method. These values will be applied automatically to all `Channels` created in the bootstrap. The options available include low-level details about connections such as the channel "keep-alive" or "timeout" properties, buffer settings, and others.

Netty applications are often integrated with an organization's proprietary software. In some cases, components such as `Channel` are passed around and used outside the normal Netty lifecycle. In the event that some of the usual properties and data may not be available, Netty offers the abstraction `AttributeMap`, `a collection which is provided by Netty's channel and bootstrap classes, and AttributeKey<T>, a generic class for inserting and retrieving attribute values`. Attributes allow you to safely associate any kind of data item with both client and server `Channels`.

For example, consider a server application that tracks the relationship between users and `Channels`. This can be accomplished by storing the user's ID as an attribute of a `Channel`. A similar technique could be used to route messages to users based their ID or to shutdown a channel based on user activity.

Listing 9.7 shows how to use `ChannelOption`s to configure a `Channel` and an attribute to store an integer value.

Listing 9.7 `Using Attributes`

```
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("Received 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` with 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**
5. **Set a handler to handle I/O and data for the channel**
6. **Retrieve the attribute with the `AttributeKey` and its value**
7. **Set the `ChannelOption`s that will be set on the created channels on connect or bind**
8. **Store the `id` attribute**
9. **Connect to the remote host with the configured `Bootstrap`**

9.7 Bootstrapping DatagramChannels

In the previous bootstrap code examples we used a `SocketChannel`, which is TCP-based. But a `Bootstrap` can also be used for connectionless protocols such as UDP. For this usage 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();                                  //1
bootstrap.group(new OioEventLoopGroup()).channel(                       //2
    OioDatagramChannel.class)                                           //3
    .handler(
        new SimpleChannelInboundHandler<DatagramPacket>(){              //4
            @Override
            public void channelRead0(ChannelHandlerContext ctx,
                DatagramPacket msg) throws Exception {
                // Do something with the packet
            }
        }
    );
ChannelFuture future = bootstrap.bind(new InetSocketAddress(0));        //5
future.addListener(new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture channelFuture)
        throws Exception {
        if (channelFuture.isSuccess()) {
            System.out.println("Channel bound");
        } else {
            System.err.println("Bind attempt failed");
            channelFuture.cause().printStackTrace();
        }
    }
});
```

1. **Create a new `Bootstrap` to create and bind new datagram channels**
2. **Specify the `EventLoopGroup` to get `EventLoops` from and register with the channels**
3. **Specify the `Channel` class**
4. **Set a handler to handle I/O and data for the channel**
5. **Call `bind()` as the protocol is connectionless**

9.8 Shutting down a previously bootstrapped Client or Server

Bootstrapping gets your application up and running, but sooner or later you will also need to shut it down gracefully. You could of course just let the JVM handle everything on exit but this would not meet the definition of "graceful," which refers to releasing resources cleanly. There is not much magic needed to shutdown a Netty application, but there are a few things to keep in mind.

The main thing to remember is to shutdown the `EventLoopGroup`, which will handle any pending events and tasks and subsequently release all active threads. This is just a matter of calling `EventLoopGroup.shutdownGracefully()`. This call will return a `Future` which is notified when the shutdown completes. Note that `shutdownGracefully()` is also an asynchronous operation, so you will need to either block until it completes or register a listener with the returned `Future` to be notified of completion.

Listing 9.9 meets the definition of a "graceful shutdown."

Listing 9.9 Graceful shutdown

```
EventLoopGroup group = new NioEventLoopGroup()                          //1
Bootstrap bootstrap = new Bootstrap();                                  //2
bootstrap.group(group)
    .channel(NioSocketChannel.class);
...
...
Future<?> future = group.shutdownGracefully();                          //3
// block until the group has shutdown
future.sync();
```

1. **Create the `EventLoopGroup` that is used to handle the I/O**
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 `Channel`s which are currently in use.**

Alternatively, you can call `Channel.close()` explicitly on all active channels before calling `EventLoopGroup.shutdownGracefully()`. But in all cases, remember to shutdown the `EventLoopGroup` itself.

9.9 Summary

In this chapter you learned how to bootstrap your Netty-based server and client applications (including those that use connectionless protocols), how to specify configuration options on channels, and how to attach information to a channel using attributes.

In the next chapter we will study how you can test your `ChannelHandler` implementations to ensure their correctness.

10 Unit Testing

In this Chapter

- Unit testing
- EmbeddedChannel

As we have seen, ChannelHandlers are the critical elements of a Netty application. Logically, then, testing them thoroughly should be a vital part of our development process. Best practices dictate that we test not only to prove the correctness of an implementation, but to make it easy to isolate problems that crop up as code is modified.

While there is no universal definition of "unit testing," most practitioners would agree that it includes the goals just stated. The basic idea is to test your code in the smallest possible chunks and isolated as much as possible from runtime dependencies: databases, networks and other code modules. If your tests verify that each unit works correctly by itself, it becomes much easier to find the culprit when something goes awry. In this chapter we'll study a special Channel implementation that provides valuable support for unit testing ChannelHandlers.

Since the code module or "unit" under test needs to be executed outside its normal runtime environment, we need a framework or "harness" within which to run it. In our examples we will use JUnit 4 as the testing framework, so you will need a basic understanding of its usage. If it is new to you, have no fear; it is both powerful and simple, and you will find all the information you need on the JUnit website, *junit.org*.

You might also want to review the previous chapters on ChannelHandler and codecs, as these will provide the material for our examples.

10.1 Overview

We already know that ChannelHandler implementations can be chained together to build up the processing logic of a ChannelPipeline. We explained previously that this design approach supports the decomposition of potentially complex processing into small and reusable components, each of which handles a well-defined processing task or step. In this chapter we will show how it simplifies testing as well.

Netty facilitates the testing of ChannelHandlers by way of what it calls an "embedded" transport. This is provided by a special Channel implementation, EmbeddedChannel, which provides a simple way to pass events trough the pipeline.

The idea is straightforward: you write inbound or outbound data into an EmbeddedChannel and then check whether anything reached the end of the ChannelPipeline. In this way you can determine whether messages were encoded or decoded and whether any ChannelHandler actions were triggered.

The relevant methods are listed in Table 10.1.

Table 10.1 Special EmbeddedChannel methods

Name	Responsibility
writeInbound	Write an inbound message to the EmbeddedChannel. Returns true if data can be read from the EmbeddedChannel via readInbound().
readInbound	Read an inbound message from the EmbeddedChannel. Anything returned traversed the entire ChannelPipeline. This method returns null if nothing is ready to read.
writeOutbound	Write an outbound message to the EmbeddedChannel. Returns true if something can now be read from the EmbeddedChannel via readOutbound().
readOutbound	Read an outbound message from the EmbeddedChannel. Anything returned traversed the entire ChannelPipeline. This method returns null if nothing is ready to read.
Finish	Mark the EmbeddedChannel as complete and return true if data can be read from either the inbound or outbound. This will also call close on the EmbeddedChannel.

Testing Inbound and Outbound data

Inbound data are processed by ChannelInboundHandlers and represent data read from the remote peer. Outbound data are processed by ChannelOutboundHandlers and represent data to be written to the remote peer.

Depending on the ChannelHandler you are testing you would choose writeInbound(), writeOutbound(), or perhaps both.

Figure 10.1 shows how data flow through the ChannelPipeline using the methods of EmbeddedChannel.

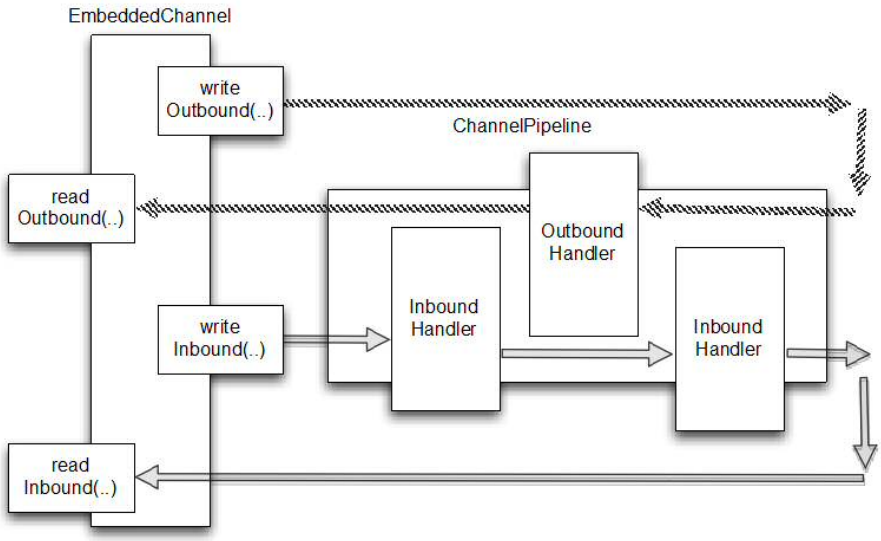


Figure 10.1 EmbeddedChannel data flow

Figure 10.1 shows how you can use `writeOutbound`() to write a message to the `Channel` and pass it through the `ChannelPipeline` in the outbound direction. Subsequently you can read the processed message with `readOutbound`() to determine whether the result is as expected. Similarly, for inbound data you can use `writeInbound`() and `readInbound`().

In each case, messages are passed through the ChannelPipeline and processd by the relevant `ChannelInboundHandlers` or `ChannelOutboundHandlers`. If the message is not consumed you can use `readInbound`() or `readOutbound`() as appropriate to read the messages out of the `Channel` after processing them.

Let's take a closer look at both scenarios and see how they apply to testing your application logic.

10.2 Testing ChannelHandler

In this section we will illustrate the use of `EmbeddedChannel` to test a `ChannelHandler`.

10.2.1 Testing inbound messages

Figure 10.2 depicts a simple `ByteToMessageDecoder` implementation. This will produce frames of a fixed size, given sufficient data. 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.

In this case, the fixed frame size is three bytes, which you can see from the frames produced. Note that it may require more then one "event" to provide enough bytes to produce a frame.

Finally, each frame will be passed to the next `ChannelHandler` in the `ChannelPipeline`.

Figure 10.2 Decoding via FixedLengthFrameDecoder

The implementation of this decoder is 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` to handle inbound bytes and decode them to messages
#2 Specify the length of the frames to be produced
#3 Check if enough bytes are ready to read to produce the next frame
#4 Read a new frame out of the `ByteBuf`
#5 Add the frame to the `List` of decoded messages.

Now let's create a unit test to make sure this code works as expected. As we pointed out earlier, even if the code is simple, unit tests will help to avoid problems that might occur if the code is refactored in the future.

Listing 10.2 shows a test of the above code 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
        Assert.assertTrue(channel.writeInbound(input));                             #4
        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` so that JUnit will recognize the method as a test**
2. **Create a new `ByteBuf` and fill it with bytes**
3. **Create a new `EmbeddedChannel` and add the `FixedLengthFrameDecoder` to be tested**
4. **Write data to the `EmbeddedChannel`**
5. **Mark the channel finished**
6. **Read the produced messages and verify**

The method `testFramesDecoded()` verifies that a `ByteBuf` containing 9 readable bytes is decoded into 3 `ByteBuf`s, each containing 3 bytes. Notice how the `ByteBuf` is populated with 9 readable bytes in one call of `writeInbound()`. After this, `finish()` is executed to mark the `EmbeddedChannel` complete. Finally, `readInbound()` is called to read exactly 3 frames and a `null` from the `EmbeddedChannel`.

The method `testFramesDecoded2()` is similar, with one difference. Here the inbound `ByteBuf`s are written in two steps. When `writeInbound(input.readBytes(2))` is called, false is returned. Why? As stated in Table 10.1 above, `writeInbound()` returns true if a subsequent call to `readInbound()` would return data. But the `FixedLengthFrameDecoder` will produce output only when 3 or more bytes are readable. The rest of the test is identical to `testFramesDecoded()`.

10.2.2 Testing outbound messages

Testing the processing of outbound messages is similar to what we have just seen. This example will use an implementation of `MessageToMessageEncoder`: `AbsIntegerEncoder`.

- When a `flush()` is received it will read 4-byte integers from the `ByteBuf` and call `Math.abs()` on each one.
- Each integer is then written to the `ChannelHandlerPipeline`.

Figure 10.3 shows the logic.

Figure 10.3 Encoding via AbsIntegerEncoder

Listing 10.3 implements the diagram. The `encode()` method writes the produced values to a `List`.

Listing 10.3 AbsIntegerEncoder

```
public class AbsIntegerEncoder extends
      MessageToMessageEncoder<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 `MessageToMessageEncoder` to encode a message to another format
#2 Check if there are enough bytes to encode
#3 Read the next `int` out of the input `ByteBuf` and calculate the absolute value
#4 Write the `int` to the `List` of encoded messages

As in the previous example we will test the code using `EmbeddedChannel`. Listing 10.4

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 write negative `int`s
#3 Create a new `EmbeddedChannel` and install the `AbsIntegerEncoder` to be tested
#4 Write the `ByteBuf` and assert that `readOutbound()` will produce data.
#5 Mark the channel finished
#6 Read the produced messages and check that the negative values have been encoded to absolute values.

Here are the steps executed:

- Write 9 ints to a new `ByteBuf`.
- Create a new `EmbeddedChannel`.
- `MessageToMessageEncoder` is a `ChannelOutboundHandler` which will manipulate data to be written to the remote peer, so we call `writeOutbound()`.
- Mark the channel finished.
- Read all the `int`s from the outbound side of the `EmbeddedChannel` and verify that it contains only absolute values.

10.3 Testing exception handling

We usually have additional tasks to handle beyond just transforming data. For example, we may need to respond to maformed input or an excessive volume of data. In the next example we will throw a `TooLongFrameException` if the number of bytes read exceeds a specified limit. This is an approach often used to guard against resource exhaustion.

In Figure 10.4 the maximum frame size has been set to 3 bytes.

Figure 10.4 Decoding via FrameChunkDecoder

If the input bytes exceed that limit they are discarded and a `TooLongFrameException` is thrown. The other `ChannelHandlers` in the pipeline can handle the exception in `exceptionCaught()` or just ignore it.

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` to decode inbound bytes to messages
#2 Specify the maximum allowable size of the frames to be produced
#3 Discard the frame if it is too large and throw a `TooLongFrameException`
#4 Otherwise read the new frame out of the `ByteBuf`
#5 Add the frame to the `List` of decoded messages.

Again we will test the code using `EmbeddedChannel`, as 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();                                                #6
        } catch (TooLongFrameException e) {
            // expected
        }
        Assert.assertTrue(channel.writeInbound(input.readBytes(3)));      #7
        Assert.assertTrue(channel.finish());                              #8

        // Read frames                                                    #9
        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 write 9 bytes to it
#3 Create a new `EmbeddedChannel` and install a `FixedLengthFrameDecoder` to be tested
#4 Write 2 bytes to it and assert that they produced a new frame (message)
#5 Write a frame which is larger then the maximum frame size (3) and check for a `TooLongFrameException`
#6 If the exception is not caught we will reach this assertion and the test will fail. Note that if the class implements `exceptionCaught()` and handles the exception then the exception will not be caught here.
#7 Write the remaining 2 bytes and assert a valid frame
#8 Mark the channel finished
#9 Read the produced messages and verify the values. Note that `assertEquals(Object,Object)` tests for equality using `equals()`, not equality of the object references.

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`.

At first glance this looks quite similar to the test we wrote in Listing 10.2, but it has an interesting twist; namely, the handling of the `TooLongFrameException`. The `try/catch` block used here is a special feature of `EmbeddedChannel`. If one of the "`write*`" methods produces a checked `Exception` it will be thrown wrapped in a `RuntimeException`. This makes it easy to test if an `Exception` was processed as part of the processing.

10.4 Summary

Unit testing employing a test harness such as JUnit is an extremely effective way to guarantee the correctness of your code and enhance its maintainability. In this chapter you learned how to test your custom `ChannelHandler`s to verify that they work as intended.

In the next chapters we will focus on writing "real-world" applications with Netty. Even if we don't present any further examples of test code we hope you will keep in mind the importance of the testing approach we have explored here.

11

WebSockets

This chapter covers

- WebSockets
- ChannelHandler, Decoder and Encoder
- Bootstrapping your Application

> *The **real-time web** is a set of technologies and practices that enable users to receive information as soon as it is published by its authors, rather than requiring that they or their software check a source periodically for updates*[34]*.*

While a full-blown "real-time web" may not be just around the corner, the idea behind it is fueling a growing expectation of instantaneous access to information. Let's be clear that we are not talking about so-called "hard" real-time computing, where computation results are guaranteed within a specified interval - the request/response design of HTTP alone makes that impractical. This has been shown by the failure of numerous workarounds devised over the years to provide a truly satisfactory solution.

The WebSockets protocol was designed from the ground up to provide bidirectional data transmission, allowing client and server to transmit messages at any time and requiring them to handle message receipt asynchronously. Most recent browsers support WebSockets as the client-side API of HTML5.

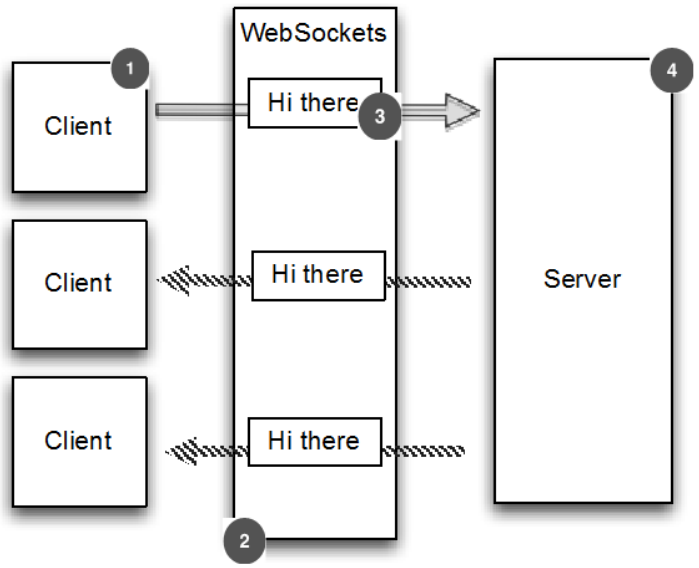
Netty's support for WebSockets[35] includes all of the principal implementations in use, so adopting it in your next application is straightforward. As usual with Netty, you can make complete use of the protocol without having to worry about its internal implementation details. We'll demonstrate this by developing a real-time chat application built on WebSockets.

11.1 The example WebSockets application

To illustrate real-time functionality our example application will use the WebSockets protocol to implement a browser-based chat application such as you may have encountered in the text-messaging feature of Facebook. But we will take it further by allowing multiple users to communicate with each other simultaneously.

Figure 11.1 shows the application logic.

[34] http://en.wikipedia.org/wiki/Real-time_web
[35] http://tools.ietf.org/html/rfc6455



#1 Client / User connects to the Server and is part of the chat
#2 Chat messages are exchanged via WebSockets
#3 Messages are sent bidirectionally
#4 The Server handles all the Clients / Users

Figure 11.1 Application logic

The logic is straightforward:

1. A client sends a message.
2. The message is broadcast to all other connected clients.

This is just how you expect a chat-room to work: everyone can talk to everyone else. This example will provide only the server side, the client being a browser that accesses the chat room via a web page. As you will see in the next few pages, WebSockets makes this simple.

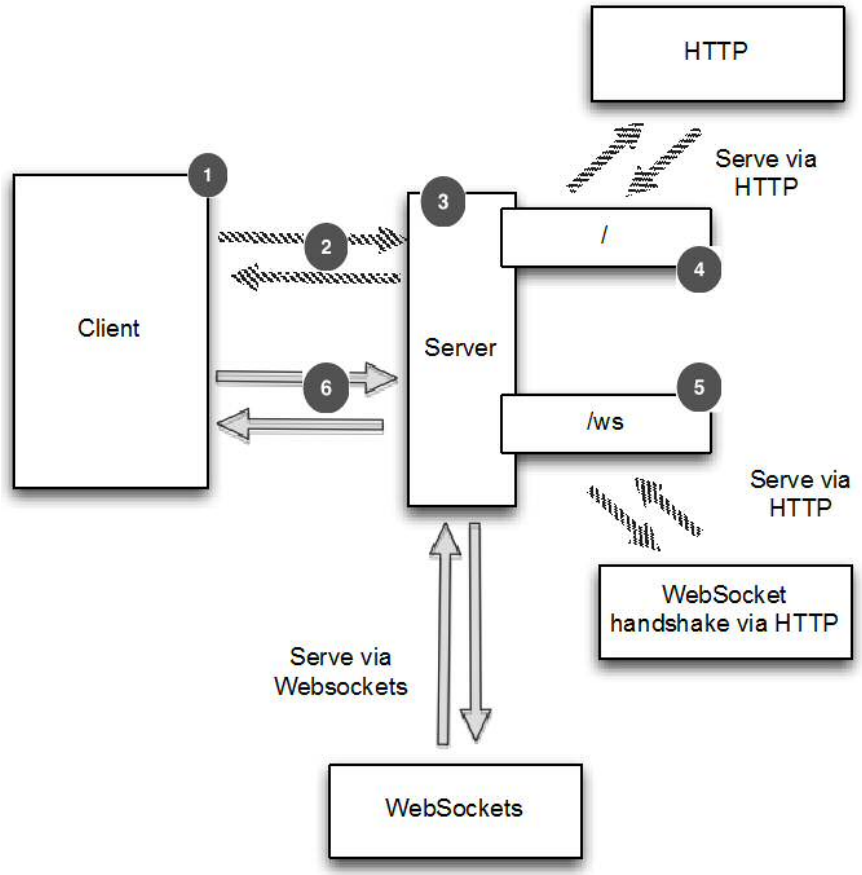
11.2 Adding WebSockets support

WebSockets uses a mechanism known as the "Upgrade handshake"[36] to switch from standard HTTP or HTTPS protocol to WebSockets. Thus, an application that uses WebSockets will always start with HTTP/S and then perform the Upgrade. At what point this happens is specific to the application; it may be at startup or when a specific URL has been requested.

[36] https://developer.mozilla.org/en-US/docs/HTTP/Protocol_upgrade_mechanism

In our application we will upgrade the protocol to WebSockets only if the URL requested ends with "`/ws`." Otherwise the server will use basic HTTP/S. Once upgraded the connection will transmit all data using WebSockets.

Figure 11.2 shows the server logic.



#1 Client / User connects to the server and joins the chat
#2 HTTP request for page or WebSocket Upgrade handshake
#3 Server handles all Clients / Users
#4 Respond to request for URI "/", which will transmit index.html
#5 Handle the WebSockets Upgrade if the URI "/ws" is accessed
#6 Send chat messages via WebSockets after the Upgrade completes

Figure 11.2 Server logic

As always in Netty, the logic will be implemented by a set of `ChannelHandler`s. In the next sections we will describe them as well as the techniques used to handle the HTTP and WebSockets protocols.

11.2.1 Handling HTTP requests

In this section we will implement the component that handles HTTP requests, which will serve the page that provides access to the "chat room" and display the messages sent by connected clients. Listing 11.1 has the code for this `HttpRequestHandler`, a `ChannelInboundHandler` implementation for `FullHttpRequest` messages. Notice how it ignores requests for the `"/ws"` URI.

Listing 11.1 HTTPRequestHandler

```
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 If the request is a WebSocket Upgrade request increment the reference count (`retain`) and pass it to the next `ChannelInboundHandler` in the `ChannelPipeline`.
#3 Handle "100 Continue" requests in conformity with HTTP 1.1
#4 Read index.html
#5 If keepalive is requested add the required headers
#6 Write the `HttpResponse` to the client.
#7 Write index.html to the client. Depending on whether an `SslHandler` is in the `ChannelPipeline` use `DefaultFileRegion` or `ChunkedNioFile`
#8 Write and flush the `LastHttpContent` to the client which marks the response as complete.
#9 If keepalive is not requested close the `Channel` after the write completes.

The `HttpRequestHandler` shown in Listing 11.1 does the following:

- If the HTTP request was sent to the URI "`/ws`" call `retain()` on the `FullHttpRequest` and forward it to the next `ChannelInboundHandler` by calling `fireChannelRead(msg)`. The call to `retain()` is needed because after `channelRead()` completes it will call `release()` on the `FullHttpRequest` to release its resources. (Please refer to our earlier discussion of `SimpleChannelInboundHandler` in Chapter 6.)
- `If the client sends the HTTP 1.1 header "Expect: 100-continue" send a "100 Continue"` response.

- Write an `HttpResponse` back to the client after the headers are set. Note that this is not a `FullHttpResponse` as it is only the first part of the response. Also, we do not use `writeAndFlush() here - this is done at the end.`
- If neither encryption nor compression are required the greatest efficiency can be achieved by storing the contents of index.html in a `DefaultFileRegion`. This will utilize zero-copy to perform the transmission. For this reason we check to see if there is an `SslHandler` in the `ChannelPipeline`. Alternatively, we use `ChunkedNioFile`.
- Write a `LastHttpContent` to mark the end of the response and terminate it
- If keepalive is not requested add a `ChannelFutureListener` to the `ChannelFuture` of the last write and close the connection. Note that here we call `writeAndFlush()` to flush all previously written messages.

This represents the first part of the application, which handles pure HTTP requests and responses. Next we will handle the WebSocket frames, which transmit the chat messages.

WebSocket frames

WebSockets transmits data in "frames," each of which represents a part of a message. A complete message may utilize many frames.

11.2.2 Handling WebSocket frames

The WebSockets[37] "Request for Comments" (RFC) defines six different frames; Netty provides a POJO implementation for each of them. Table 11.1 lists the frame types and describes their use.

Table 11.1 WebSocketFrame types

Frame type	Description
`BinaryWebSocketFrame`	contains binary data
`TextWebSocketFrame`	contains text data
`ContinuationWebSocketFrame`	contains text or binary data that belongs to a previous `BinaryWebSocketFrame` or `TextWebSocketFrame`
`CloseWebSocketFrame`	represents a `CLOSE` request and contains close status code and a phrase
`PingWebSocketFrame`	requests the transmission of a `PongWebSocketFrame`
`PongWebSocketFrame`	sent as a response to a `PingWebSocketFrame`

[37] https://tools.ietf.org/html/rfc6455

Our chat application will use the following frame types:

- `CloseWebSocketFrame`
- `PingWebSocketFrame`
- `PongWebSocketFrame`
- `TextWebSocketFrame`

However, `TextWebSocketFrame` is the only one we actually need to handle. In conformity with the WebSockets RFC, Netty provides a `WebSocketServerProtocolHandler` to manage the others.

Listing 11.2 shows our `ChannelInboundHandler` for `TextWebSocketFrames`, which will also track 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 `userEventTriggered()` method to handle custom events
#3 If the event received indicates that the handshake was successful remove the `HttpRequestHandler` from the `ChannelPipeline` as no further HTTP messages will be received.
#4 Write a message to all connected WebSocket clients about the new `Channel` that has been connected
#5 Add the new WebSocket `Channel` to the `ChannelGroup` so it will receive all messages
#6 Retain the received message and `writeAndFlush()` it to all connected clients.

The `TextWebSocketFrameHandler` shown in Listing 11.2 has a very limited set of responsibilities:

- When the WebSocket handshake with the new client has completed successfully notify all connected clients by writing to all the `Channels` in the `ChannelGroup`, then add the new `Channels` to the `ChannelGroup`.
- If a `TextWebSocketFrame` is received, call `retain()` on it and write and flush it to the `ChannelGroup` so that all connected WebSockets `Channels` will receive it. As before, calling `retain()` is required because the reference count of `TextWebSocketFrame` will be decremented when `channelRead0()` returns. Since all operations are asynchronous, `writeAndFlush()` might complete later and we don't want it to access a reference that has become invalid.

Since Netty handles most of the remaining functionality internally, the only thing left for us to do now is to initialize the `ChannelPipeline` for each new `Channel` that is created. For this we'll need a `ChannelInitializer`.

11.2.3 Initializing the ChannelPipeline

Next we need to install our two `ChannelHandler`s in the `ChannelPipeline`. For this we will extend `ChannelInitializer` and implement `initChannel()`. Listing 11.3 shows the code for the resulting `ChatServerInitializer`.

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`
#2 Add all needed `ChannelHandler`s to the `ChannelPipeline`

The `initChannel()` method sets up the `ChannelPipeline` of the newly registered `Channel`, installing all the required `ChannelHandler`s. These are summarized in Table 11.2 along with their individual responsibilities.

Table 11.2 ChannelHandlers for the WebSockets Chat server

ChannelHandler	Responsibility
`HttpServerCodec`	Decode bytes to `HttpRequest`, `HttpContent`, `LastHttpContent`. Encode `HttpRequest`, `HttpContent`, `LastHttpContent` to bytes.
`ChunkedWriteHandler`	Write the contents of a file.
`HttpObjectAggregator`	This `ChannelHandler` aggregates an `HttpMessage` and its following `HttpContent`s into a single `FullHttpRequest` or `FullHttpResponse` (depending on whether it is being used to handle requests or responses). With this installed the next `ChannelHandler` in the pipeline will receive only full HTTP requests.
`HttpRequestHandler`	Handle `FullHttpRequest`s (those not sent to "`/ws`" URI).
`WebSocketServerProtocolHandler`	As required by the WebSockets specification, handle the WebSocket Upgrade handshake, `PingWebSocketFrames`, `PongWebSocketFrames` and `CloseWebSocketFrames`.
`TextWebSocketFrameHandler`	Handles `TextWebSocketFrames` and handshake completion events

The `WebSocketServerProtocolHandler` handles all mandated WebSockets frame types and the Upgrade handshake itself. If the handshake is successful the required `ChannelHandler`s are added to the pipeline and those that are no longer needed are removed. The state of the pipeline before the Upgrade is illustrated in Figure 11.3. This represents the `ChannelPipeline` just after it has been initialized by the `ChatServerInitializer`.

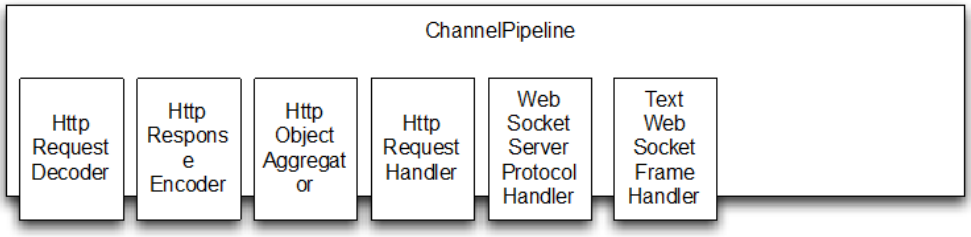


Figure 11.3 ChannelPipeline before WebSockets Upgrade

When the Upgrade is completed the `WebSocketServerProtocolHandler` replaces the `HttpRequestDecoder` with a `WebSocketFrameDecoder` and the `HttpResponseEncoder` with a `WebSocketFrameEncoder`. To maximize performance it will then remove any `ChannelHandlers` that are not required for WebSockets connections. These would include the `HttpObjectAggregator` and `HttpRequestHandler` shown in Figure 11.3.

Figure 11.4 shows the `ChannelPipeline` after these operations have completed. Note that Netty currently supports four versions of the WebSockets protocol, each by way of its own implementation classes. Selection of the correct version of `WebSocketFrameDecoder` and `WebSocketFrameEncoder` is performed automatically, depending on what the client (here the browser) supports.[38]

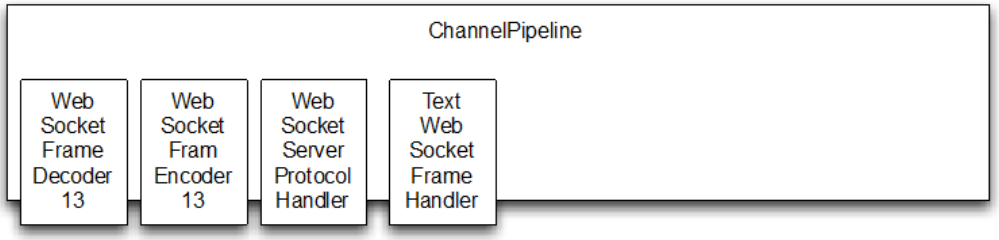


Figure 11.4 ChannelPipeline after WebSockets Upgrade

11.2.4 Bootstrapping

The final piece of the picture is the code that bootstraps the server and sets the `ChannelInitializer`. This will be done by the `ChatServer` class as shown in Listing 11.4.

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
```

[38] In this example we assume that Version 13 of the WebSockets protocol is used, thus `WebSocketFrameDecoder13` and `WebSocketFrameEncoder13` are shown in the figure.

```
        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 `ChannelInitializer`
#4 Handle server shutdown including release of all resources

That completes the application itself - now let's test it.

11.3 Testing the Application

The example code in the directory *chapter11* has everything you need to build and run the server. (If you have not yet set up your development environment including Apache Maven, please refer to the instructions in Chapter 2.)

We will use the following Maven command to build and start the server

```
mvn -PChatServer clean package exec:exec
```

The project file *pom.xml* is configured to start the server on port 9999. To use a different port you can either edit the value in the file or override it with a system property:

```
mvn -PChatServer -Dport=1111 clean package exec:exec
```

Listing 11.5 shows the main output of the command (some lines have been deleted).

Listing 11.5 Compile and start the ChatServer

```
$ mvn -PChatServer clean package exec:exec

[INFO] Scanning for projects...
```

```
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building ChatServer 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
...
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action ---
[INFO] Building jar: D:/netty-in-action/chapter11/target/chat-server-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ chat-server ---
Starting ChatServer on port 9999
```

You can access the application by pointing your browser to **http://localhost:9999**. Figure 11.5 shows the user interface in the Chrome browser.

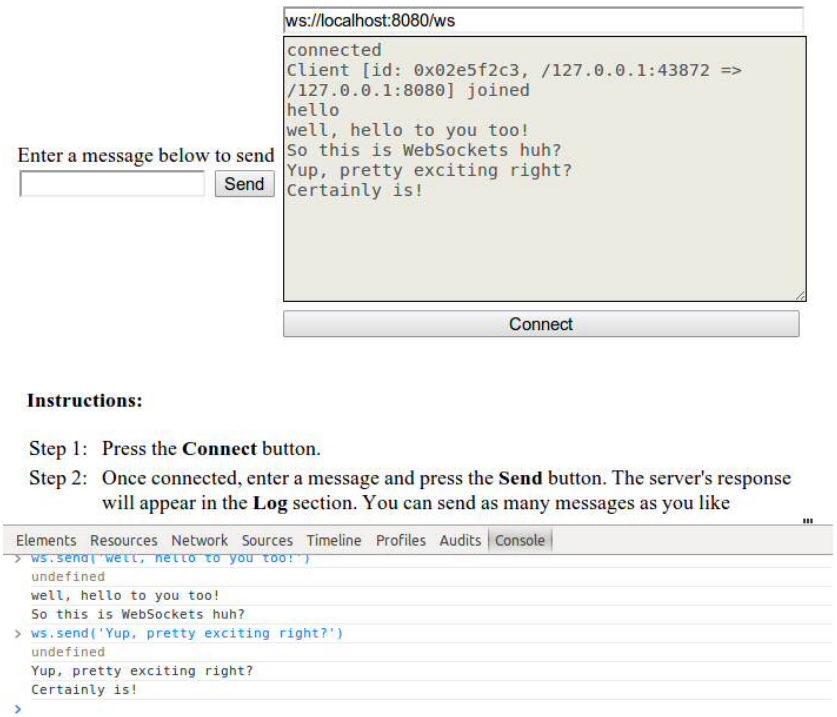


Figure 11.5 WebSockets ChatServer demonstration

The figure shows two connected clients. The first is connected using the interface at the top. The second client is connected via the Chrome 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 demonstration of how WebSockets enables real-time communication in a browser.

11.3.1 What about Encryption?

In a real-life scenario, you would soon be asked to add encryption to this server. With Netty there is not much more to this than adding an `SslHandler` to the `ChannelPipeline` and configuring it. Listing 11.6 shows how this is done by extending the `ChatServerInitializer`.

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` to add encryption
#2 Add the `SslHandler` to the `ChannelPipeline`

Finally we step adapt the `ChatServer` to use the `SecureChatServerInitializer` and pass in the `SSLContext`. This is shown in Listing 11.7.

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
#2 Return the previously created `SecureChatServerInitializer` to enable encryption

That's all that is needed to enable SSL/TLS[39] encryption of all communications. As before you can use Apache Maven to start the application and have it pull in all needed dependencies.

Listing 11.8 Start the SecureChatServer

```
$ mvn -PSecureChatServer clean package exec:exec
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building ChatServer 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
...
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action ---
[INFO] Building jar: D:/netty-in-action/chapter11/target/chat-server-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ chat-server ---
Starting SecureChatServer on port 9999
```

Now you can acess the SecureChatServer from its HTTPS URL: **https://localhost:9999**.

11.4 Summary

In this chapter we learned how to use Netty's WebSockets implementation to manage real-time data in a web application. We covered the data types supported and discussed the limitations you may encounter. While it may not be possible to use WebSockets in all cases, it should be clear that it represents an important advance in technologies for the web.

Next we'll talk about another another development of "Web 2.0." Perhaps you haven't yet heard about "SPDY," but once you have read the next chapter you might well use it in your next application.

[39] http://tools.ietf.org/html/rfc5246

12 SPDY

This chapter covers

- An overview of SPDY
- ChannelHandler, Decoder, and Encoder
- Bootstrapping a Netty-based application
- Testing SPDY/HTTPS

> ***SPDY*** [40] *(pronounced* speedy*) is an open networking protocol developed primarily at Google for transporting web content. SPDY manipulates HTTP traffic, with particular goals of reducing web page load latency and improving web security. SPDY achieves reduced latency through compression, multiplexing, and prioritization although this depends on a combination of network and website deployment conditions. The name "SPDY" is a trademark of Google and is not an acronym.*[41]

Netty bundles support for SPDY, just as it does for so many protocols. And as we have already seen in other cases, this support will enable you to use SPDY without having to worry about all of its internal details. In this chapter we'll provide all the information you need to SPDY-enable your application and support both SPDY and HTTP at the same time.

12.1 SPDY background

Google developed SPDY to address problems of scalability. One of its main tasks is to make the loading of content as fast as possible, and to this end SPDY does the following:

- Every header is compressed. Compression of the message body is optional because it can be problematic for proxy servers.
- All encryption uses TLS[42].
- Multiple transfers per connection are possible.
- Data sets can be individually prioritized, allowing critical content to be transferred first.

Table 12.1 shows how this approach compares with HTTP.

Table 12.1 Comparison of SPDY and HTTP

Browser	HTTP 1.1	SPDY
Encrypted	Not by default	Yes
Header compression	No	Yes

[40] http://www.chromium.org/spdy/spdy-whitepaper
[41] http://en.wikipedia.org/wiki/SPDY
[42] http://en.wikipedia.org/wiki/Transport_Layer_Security

Full duplexing	No	Yes
Server push	No	Yes
Priorities	No	Yes

Some benchmarks have shown that SPDY can provide a 50 percent speed increase over HTTP. However, actual results will vary depending on the usage pattern.

Although initially only Google Chrome supported SPDY, it is now supported by most web browsers. SPDY is currently available in three different versions, corresponding to SPDY Protocol Draft specifications 1, 2 and 3. Currently Netty supports drafts 2 and 3, these being the ones supported by the most widely-used browsers at the time of writing. These browsers are listed in Table 12.2.

Table 12.2 Browsers that support SPDY

Browser	Version
Chrome	19+
Chromium	19+
Mozilla Firefox	11+ (enabled by default since 13)
Opera	12.10+

12.2 The sample application

We'll write a simple server application to show you how you can integrate SPDY into your next application. All it will do is serve some static content back to the client. This content will vary depending on whether the protocol used is HTTPS or SPDY. The switch to SPDY will happen automatically if the client browser supports the SPDY version provided by our server. Figure 12.1 shows the flow of the application.

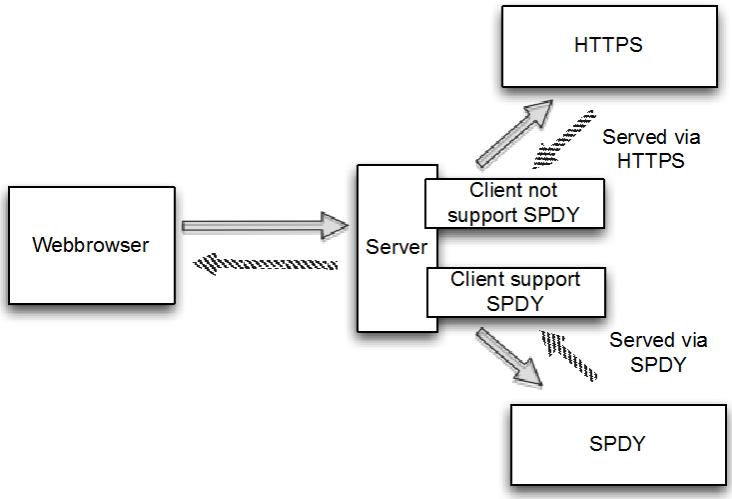


Figure 12.1 Application logic

For this application we'll write only a server component that handles both HTTPS and SPDY. To demonstrate its capability we'll use two different web browsers, one that supports SPDY and one that does not.

12.3 Implementation

SPDY uses a TLS extension called Next Protocol Negotiation (NPN) to choose the protocol. In Java we have two different ways to act on the selected protocols based on NPN:

- Use ssl_npn, an opensource SSL provider for NPN.
- Use NPN via the Jetty extension library.

In this example we'll use the Jetty library. If you want to use ssl_npn, please refer to the project documentation at https://github.com/benmmurphy/ssl_npn.

Jetty NPN Library

The Jetty NPN library is an external library, not part of Netty itself. It handles Next Protocol Negotiation, which is used to detect whether or not the client supports SPDY.

12.3.1 Integration with Next Protocol Negotiation

The Jetty library provides an interface called `ServerProvider`, which determines the protocol to use and hooks into the one selected. The implementation may differ according to which versions of HTTP and SPDY versions are supported. The following listing shows the implementation that will be used in our example application.

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 protocols supported by this ServerProvider implementation
#2 Set the protocol to http/1.1 if SPDY detection fails
#3 Return the list of supported protocols
#4 Set the selected protocol
#5 Return the selected protocol

In this `ServerProvider` implementation we will support selection among three different protocols:

- SPDY draft 2
- SPDY draft 3
- HTTP 1.1

If we fail to detect the protocol we will default to HTTP 1.1. This will allow all clients to be served even if SPDY isn't supported.

This is the only integration code that we require to make use of NPN. Next we will write the code that will utilize it.

12.3.2 Implementation of the various ChannelHandlers

Our first `ChannelInboundHandler` is a simple one that will handle HTTP requests for clients that don't support SPDY. Since this protocol is still very new and not universally supported on the client side, it is important to be able to fall back to HTTP. Otherwise, many users will have an unpleasant experience trying to use our service.

Listing 12.2 shows the handler for HTTP traffic.

Listing 12.2 Implementation that 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 channelRead0() , which will be called for each FullHttpRequest received
#2 Check if a Continue response is expected, and if so write one
#3 Create a new FullHttpResponse, which will answer the request
#4 Generate the content of the response and write it to its payload
#5 Set the header so the client knows how to interpret the payload of the response
#6 Check if the request was set with keepalive enabled; if so, set the headers to comply with the HTTP RFC

#7 Write the response to the client and get a reference to the Future which will get notified once the write completes
#8 If the response isn't using keepalive, close the connection after the write completes
#9 Return the content that will be used as the response payload
#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's review the implementation in Listing 12.2:

1. The `channelRead0()` method is called once a `FullHttpRequest` is 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 whether keepalive 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 is closed if keepalive is not set.

This is essentially standard HTTP handling with Netty. You might decide to handle specific URIs individually and respond with different status codes, depending on whether a resource is present or not, but the basic concept would be the same.

Our next task will be to provide a component to support SPDY as the preferred protocol. Thanks to the SPDY handlers Netty provides, this is easy. These will enable you to reuse the `FullHttpRequest` and `FullHttpResponse` messages and transparently receive and send them via SPDY.

While we can reuse the code in `HttpRequestHandler`, we will change the content we write back to the client just to highlight the protocol change; ordinarily you would just return the same content. The following listing shows the implementation, which extends the previously written `HttpRequestHandler`.

Listing 12.3 Implementation that 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 shares the same logic
#2 Generates the content that is written to the payload. This overrides the implementation of `getContent()` in HttpRequestHandler.

`SpdyRequestHandler` extends `HttpRequestHandler`, but with the difference we mentioned: the content that's written to the payload states that the response was written over SPDY.

With the two handlers in place we can implement logic that will choose the one that matches the protocol detected. However adding, the previously written handler to the `ChannelPipeline` isn't enough; the correct codec also need to be added. Its responsibility is to detect the transmitted bytes and then work with the `FullHttpResponse` and `FullHttpRequest` abstractions.

Netty ships with a base class that does exactly this. All you need to do is to implement the logic to select the protocol and select the appropriate handler.

Listing 12.4 shows the implementation, which uses the abstract base class provided by Netty.

Listing 12.4 Implementation that 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 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 was detected
#7 Will be called to add the handler for FullHttpRequest messages. This method is called only if no SPDY is supported by the client and so HTTPS should be used.
#8 Will be called to add the handler for FullHttpRequest messages. This method is called if SPDY is supported.

This implementation will take care of detecting the correct protocol and setting up the `ChannelPipeline` for you. It handles SPDY versions 2 and 3 and HTTP 1.1 but could be easily modified to support additional versions of SPDY.

12.3.3 Setting up the ChannelPipeline

With all the pieces in place it remains only to wire the handlers together. This is done by implementing `ChannelInitializer` as usual. As you've learned, this will set up the `ChannelPipeline` and add all the needed `ChannelHandlers`.

SPDY requires two `ChannelHandlers`:

- the `SslHandler`, because detection of SPDY is done via a TLS extension
- our `DefaultSpdyOrHttpChooser`, which will add the correct `ChannelHandlers` to the `ChannelPipeline` once the protocol has been detected

Besides adding the `ChannelHandlers` to the `ChannelPipeline`, `ChannelInitializer` has another responsibility; namely, to assign the previously created `DefaultServerProvider` to the `SslEngine` used by the `SslHandler`. This will be done by the `NextProtoNego` helper class, which is provided by the Jetty NPN library.

Listing 12.5 shows the details.

Listing 12.5 Implementation that 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 ChannelInitializer for an easy starting point
#2 Pass the SSLContext, which will be used to create the SSLEngines
#3 Create a new SSLEngine, which will be used for the new channel/connection
#4 Configure the SSLEngine for non-client use
#5 Bind the DefaultServerProvider to the SSLEngine via the NextProtoNego helper class
#6 Add the SslHandler to the ChannelPipeline; this will remain in the ChannelPipeline even after the protocol is detected
#7 Add the DefaultSpyOrHttpChooser to the ChannelPipeline. This implementation will detect the protocol, add the correct ChannelHandlers into the ChannelPipeline, and remove itself.

The actual `ChannelPipeline` setup is done later by the `DefaultSpdyOrHttpChooser` implementation, because at this point it's possible to know only whether the client supports SPDY or not.

To illustrate this, let's recap and have a look at the different `ChannelPipeline` states during the lifetime of the connection with the client. Figure 12.2 shows the `ChannelPipeline` after the `Channel` has been initialized.

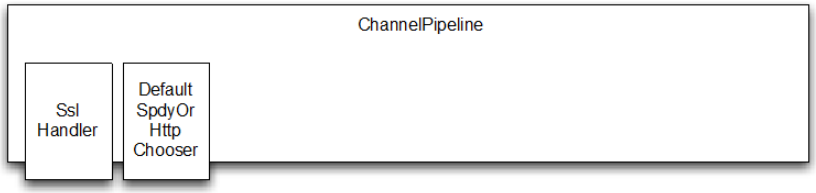


Figure 12.2 ChannelPipeline after connection

Now, depending on whether the client supports SPDY or not, the pipeline will be modified by the `DefaultSpdyOrHttpChooser` to handle the protocol. `After` adding the required `ChannelHandler`s to the `ChannelPipeline` it is no longer needed, so it removes itself. This logic is encapsulated by the abstract `SpdyOrHttpChooser` class, parent of `DefaultSpdyOrHttpChooser`.

Figure 12.3 shows the `ChannelPipeline` configured for a connected client that supports SPDY.

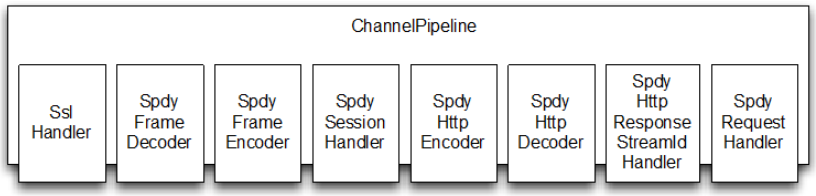


Figure 12.3 ChannelPipeline if SPDY is supported

Each of the `ChannelHandler`s is responsible for a small part of the work, a perfect illustration of a correctly constructed Netty-based application. The roles of the individual `ChannelHandler`s are shown in table 12.3.

Table 12.3 Responsibilities of the `ChannelHandler`s when SPDY is used

Name	Responsibility
`SslHandler`	Encrypt/decrypt data exchanged between two peers
`SpdyFrameDecoder`	Decode the bytes received by SPDY frames
`SpdyFrameEncoder`	Encode SPDY frames back into bytes
`SpdySessionHandler`	Handle the SPDY session
`SpdyHttpEncoder`	Encode HTTP messages into SPDY frames
`SpdyHttpDecoder`	Decode SDPY frames into HTTP messages
`SpdyHttpResponseStreamIdHandler`	Handle the mapping between requests and responses based on the SPDY ID
`SpdyRequestHandler`	Handle the `FullHttpRequest`s, which were decoded from the SPDY frame and so allow transparent usage of SPDY

The `ChannelPipeline` will look quite a bit different when the protocol is HTTP(s), as seen in Figure 13.4.

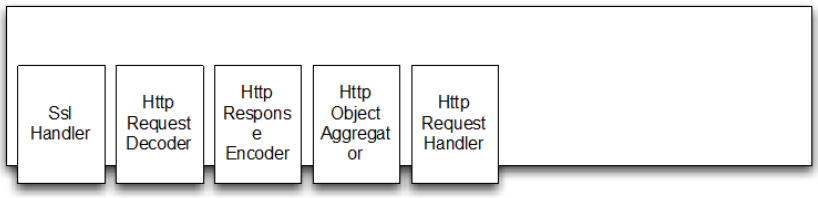


Figure 12.3 `ChannelPipeline` if SPDY is not supported

As before, each `ChannelHandler` has a defined responsibility, as highlighted in table 12.4.

Table 12.4 Responsibilities of the `ChannelHandler`s when HTTP is used

Name	Responsibility
`SslHandler`	Encrypt/decrypt data exchanged between the two peers
`HttpRequestDecoder`	Decode the bytes received by HTTP requests
`HttpResponseEncoder`	Encode HTTP responses into bytes
`HttpObjectAggregator`	Handle the SPDY session
`HttpRequestHandler`	Handle the `FullHttpRequest`s, which were decoded

12.3.4 Wiring things together

After all the `ChannelHandler` implementations are prepared, we need to wire them together and bootstrap the server. These tasks are performed by the `SpdyServer` in the following listing.

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 to handle the I/O

#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 to accept connections at the specified address.
#6 Destroy the server, which closes the channel and shuts down the NioEventLoopGroup
#7 Obtain an SSLContext from BogusSslContextFactory. This is a dummy implementation for testing purposes. A real implementation would configure the SslContext with a proper KeyStore.

The flow of the `SpdyServer` is as follows:

1. The `ServerBootstrap` is created and configured with the previously created `SpdyChannelInitializer`. The `SpdyChannelInitializer` sets up the `ChannelPipeline` once the channel is accepted.
2. Bind to the given `InetSocketAddress` and accept connections on it.

It also registers a shutdown hook to release all resources once the JVM exits.

Now let us test the `SpdyServer` and see if it works as expected.

12.4 Start the SpdyServer and test it

Note that when you use the Jetty NPN library you need to provide its location to the JVM via a `bootclasspath` argument.. This step is needed because of way in which the library hooks into the `SslEngine`. (The `-Xbootclasspath` option allows you to override standard implementations of classes that are shipped with the JDK.)

The following listing 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 the application is to use the Maven project for this chapter, as shown in Listing 12.8.

Listing 12.8 Compile and start SpdyServer with Maven

```
$ 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 has been started successfully we'll use two different browsers to test it: one that supports SPDY and another that does not. At the time of writing these are Google Chrome (SPDY) and Safari. It would also be possible to use Google Chrome and a command-line too tool like `wget` to test for fallback to HTTPS. You may get a warning if you use a self-signed certificate, but that's all right. For a production system you would probably want to purchase an official certificate.

Open Google Chrome and navigate to the https://127.0.0.1:9999 (or the address/port that you specified). You should see the image in Figure 12.4, which displays the message sent by our `SpdyRequestHandler`.

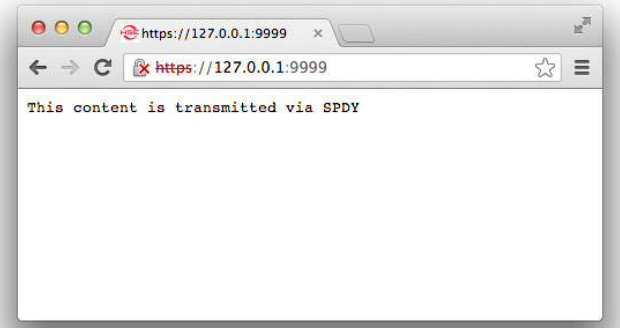


Figure 12.4 SPDY supported by Google Chrome

A nice feature in Google Chrome is that you can check some statistics about SPDY and get a deeper understanding of what is transmitted and how many connections are currently handled. For this purpose open chrome://net-internals/#spdy. Figure 12.5 shows the statistics that are exposed there.

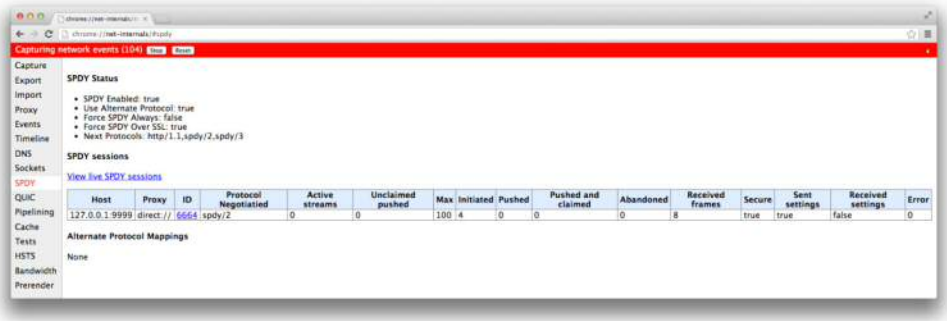


Figure 12.5 SPDY statistics

This gives you an view of all the SPDY connections that are established. You also see details about the number of bytes transmitted and so forth. To get an even more detailed overview of the transmitted data, click the ID link and refresh the window in which you've opened https://127.0.0.1:9999. This will give you details about the different SPDY frames that are transmitted to fulfill the request, as shown in Figure 12.6.

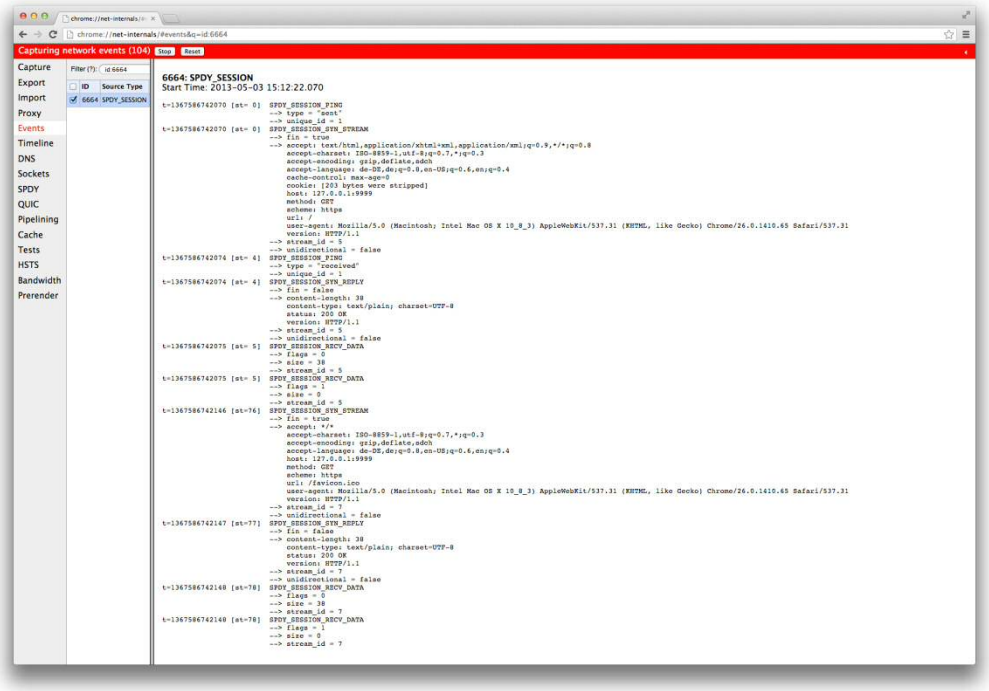


Figure 12.6 SPDY statistics

Explaining all the details displayed in figure 12.6 outside our scope, but you should be able to spot details about which HTTP headers are transmitted and which URL was accessed. These kinds of statistics are quite useful if you need to debug SPDY.

What about clients that don't support SPDY? Those will be served via HTTPS and shouldn't have a problem accessing the resources provided by the `SpdyServer`.

To test this, open a browser that doesn't support SPDY, which is Safari in our case. As before, access https://127.0.0.1:9999; you should see the response written back to the client by our `HttpRequestHandler` (Figure 12.7).

Figure 12.7 SPDY not supported by Safari

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. This provides a base from which you can benefit from the performance enhancements in SPDY while allowing existing clients to access your application.

You learned how to use the helper classes for SPDY provided by Netty and how to get more runtime information about the protocol using Google Chrome.

Along the way we saw again how modifying the `ChannelPipeline` helps you to build powerful multiplexers for switching protocols during the lifetime of a single connection.

In the next chapter you'll learn how to take advantage of the high-performance, connectionless nature of UDP.

13

Broadcasting Events with UDP

This chapter covers

- Overview of UDP
- ChannelHandler, Decoder, and Encoder
- Bootstrapping a Netty-based application

Most of the examples we have seen up to now have used connection-based protocols such as TCP. In this chapter we'll focus on a connectionless protocol, User Datagram Protocol (UDP), often used when performance is critical and some packet loss can be tolerated[43]. This chapter will give you a good grasp of connectionless protocols so you'll be able to make informed decisions about when to use UDP in your applications.

We'll start with an overview of UDP, its characteristics and limitations. Following that we will describe the example application to be developed in this chapter.

13.1 UDP Basics

Connection-oriented transports (like TCP) manage the establishment of a call (or "connection") between two network endpoints, the ordering and reliable transmission of messages sent during the lifetime of the call and finally, orderly termination of the call. By contrast, in a connectionless protocol like UDP there is no concept of a durable connection and each message (a UDP "datagram") is an independent transmission.

Furthermore, UDP does not have TCP's error-correcting mechanism, where each peer acknowledges the packets it receives and unacknowledged packets are retransmitted by the sender.

By analogy, a TCP connection is like a telephone conversation, where a series of ordered messages flow in both directions. UDP, on the other hand, resembles dropping a bunch of postcards in a mailbox. We can't know the order in which they will arrive at their destination, or even if they all will arrive.

While these aspects of UDP may strike us as serious limitations, they also explain why it is so much faster than TCP: all of the overhead of handshaking and message management has been eliminated. Clearly, UDP is a good fit only for applications that can handle or tolerate lost messages unlike, for example, those that handle money transactions.

13.2 UDP Broadcast

All of our examples to this point utilize a transmission mode called "unicast": "the sending of messages to a single network destination identified by a unique address[44]." This mode is supported by both connected and connectionless protocols.

---

[43] One of the best-known UDP-based protocols is the Domain Name Service (DNS), which maps fully-qualified names to numeric IP addresses (e.g., "netty.io" to 104.28.8.44).

[44] http://en.wikipedia.org/wiki/Unicast

UDP, however, provides additional transmission modes for sending a message to multiple recipients:

- multicast: transmits to a defined group of hosts
- broadcast: transmits to all of the hosts on a network (or a subnet)

The example application in this chapter will illustrate the use of UDP broadcast by sending messages that can be received by all the hosts on the same network. For this purpose we will use the special "limited broadcast" or "zero network" address 255.255.255.255. Messages sent to this address are destined for all the hosts on the local network (0.0.0.0) and are never forwarded to other networks by routers.

The next section will discuss the design of the sample application.

13.3 The UDP Sample Application

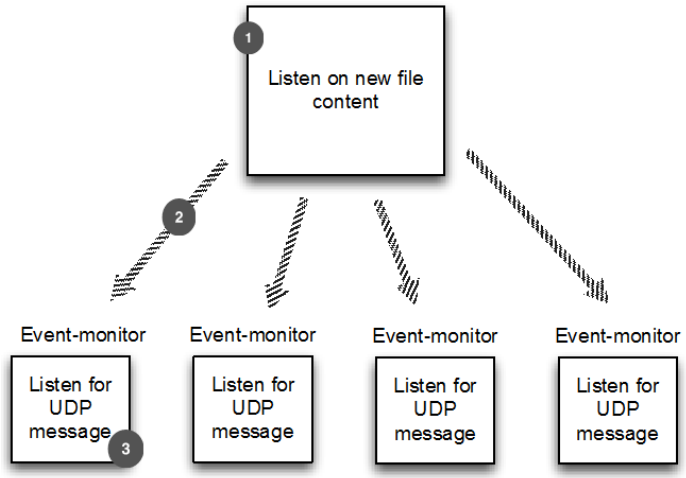
Our example application will open a file and broadcast each line as a message to a specified port via UDP. If you're familiar with UNIX-like operating systems, think of this as a very simplified version of the standard utility "syslog." UDP is a perfect fit for such an application because the occasional loss of a line of a log file can be tolerated, given that the file itself is stored in the file system. Furthermore, the application provides the very valuable capability of effectively handling a large volume of data.

UDP broadcast makes adding new event "monitors" to receive the log messages as simple as starting a listener program on a specified port. However, this ease of access also raises a potential security issue and indicates why UDP broadcast tends to be used in secure environments. Note also that broadcast messages may work only within a network because routers often block them.

Publish / Subscribe

Applications like *syslog* are typically classified as "publish/subscribe"; a producer or service publishes the events and multiple subscribers can receive them.

Let's take a high-level look at the application we're going to build. Figure 13.1 illustrates the design.



#1 Application listens for new file content
#2 Events are broadcast via UDP.
#3 The event monitor listens for and displays the content.

Figure 13.1 Application overview

The application has two components: the broadcaster and the monitor or (of which there may be multiple instances). To keep things simple we won't add authentication, verification, or encryption. (On the other hand, you may think of many potential uses for this application, and enhancing it should be fairly simple given its straightforward organization.)

In the next section we will start to explore the implementation, and we'll also discuss the differences between UDP and TCP from the point of view of application development.

13.4 EventLog POJOs

In messaging applications data is often represented by a POJO, which may hold configuration or processing information in addition to the actual message data. In this application the message unit is an "event." Since the data comes from a log file we'll call it `LogEvent`. Listing 13.1 shows the details of this simple POJO.

Listing 13.1 LogEvent message

```
public final class LogEvent {
        public static final byte SEPARATOR = (byte) ':';

    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);
    }
```

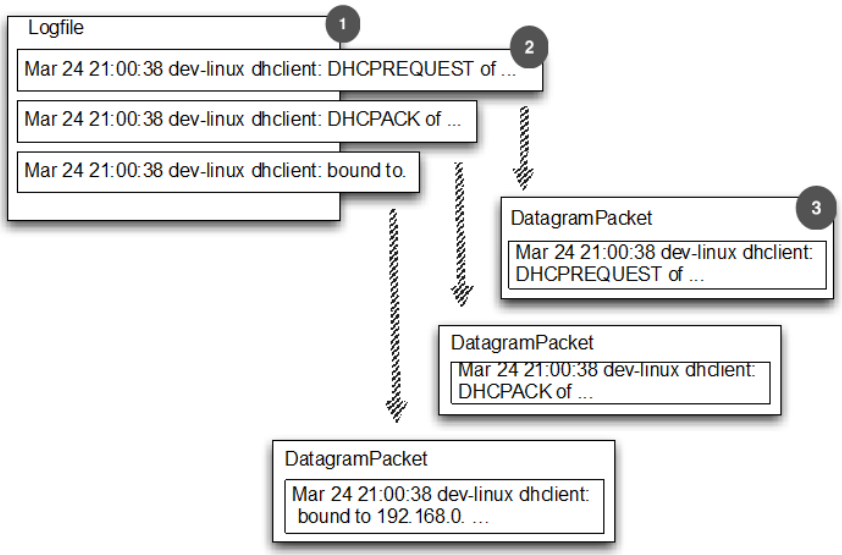
```
    public LogEvent(InetSocketAddress source, long received,
        String logfile, String msg) {                                    #2
        this.source = source;
        this.logfile = logfile;
        this.msg = msg;
        this.received = received;
    }

    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 Constructor for an outgoing message
#2 Constructor for an incoming message
#3 Return the InetSocketAddress of the source that sent the LogEvent
#4 Return the name of the log file for which the LogEvent was sent
#5 Return the message contents
#6 Return the time at which the LogEvent was received

With the message component defined we can implement the actual logic. In the next sections we'll focus on this, starting with the broadcaster.

13.5 Writing the broadcaster

In this section we'll go through all the steps needed to write the broadcaster component shown in Figure 13.1. As you can see in Figure 13.2 its function is to broadcast one `DatagramPacket` per log entry.



#1 The log file
#2 A log entry in the log file
#3 A DatagramPacket holding a single log entry

Figure 13.2 Log entries sent with `DatagramPackets`

Figure 13.3 represents a high-level view of the `ChannelPipeline` of the `LogEventBroadcaster`, showing how `LogEvents` flow through it.

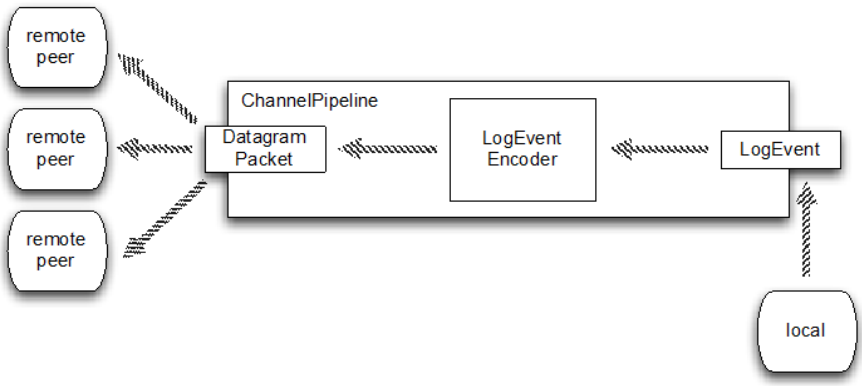


Figure 13.3 `LogEventBroadcaster: ChannelPipeline` and `LogEvent` flow

As we have seen, all data to be transmitted is encapsulated in `LogEvent` messages. The `LogEventBroadcaster` writes these via the channel on the local side, sending them through

the `ChannelPipeline` where they are converted (encoded) into `DatagramPacket` messages by a custom `ChannelHandler`. Finally, they are broadcast via UDP and picked up by remote peers.

Encoders and Decoders

Encoders and decoders convert messages from one format to another and are discussed in depth in Chapter 7. There we explore the base classes Netty provides to simplify the implementation of custom `ChannelHandler`s such as the `LogEventEncoder` we'll provide in this application.

The next listing shows how the encoder is 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 The LogEventEncoder creates DatagramPacket messages to be sent to the specified InetSocketAddress
#2 Write the filename to the ByteBuf
#3 Add a SEPARATOR
#4 Write the log message to the ByteBuf
#5 Add a new DatagramPacket to the list of outbound messages

Why use MessageToMessageEncoder?

We could of course write our own custom `ChannelOutboundHandler` to convert `LogEvent` objects to `DatagramPacket`s. But simply subclassing `MessageToMessageEncoder` does most of the work for us.

With the `LogEventEncoder` implemented we just need to define the runtime configuration of the server, which we call "bootstrapping". This includes setting various `ChannelOptions` and

installing needed `ChannelHandlers` in the `ChannelPipeline`. This will be done by the `LogEventBroadcaster` class, shown in Listing 13.3.

Listing 13.3 LogEventBroadcaster

```
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 Excption {
        Channel ch = bootstrap.bind(0).sync().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. To use broadcast we set the SO_BROADCAST socket option.
#2 Bind the channel. Note that there is no connection when using Datagram Channel.
#3 If necessary set the file pointer to the last byte of the file.
#4 Set the current file pointer so nothing old is sent
#5 Write a LogEvent to the channel that holds the filename and the log entry (we expect every log entry is only one line long)
#6 Store the current position within the file so we can later continue here
#7 Sleep for 1 second. If interrupted exit the loop else restart it.
#8 Construct a new instance of LogEventBroadcaster and start it

This completes the first part of our application. You can see the results at this point by using the "*netcat*" program. On UNIX/Linux systems you should find it installed as "*nc.*" You can find a version for Windows at *http://nmap.org/ncat*.

Netcat is perfect for a first test of our application; it just listens on a specified port and prints all data received to standard output. Set it to listen for UDP data on port 9999 as follows:

```
$ nc -l -u 9999
```

Now we need to start up our `LogEventBroadcaster`. Listing 13.4 shows how to compile and run the broadcaster using *mvn*. The configuration in *pom.xml* points to a file that is frequently updated, */var/log/syslog* (assuming a UNIX/Linux environment) and sets the port to 9999. The entries in the file will be broadcast to that port via UDP and printed to the console on which you started *netcat*.

Listing 13.4 Compile and start the LogEventBroadcaster

```
$ 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
```

To change the file and port values, specify the ones you want as system properties when invoking *mvn*. Listing 13.5 sets the logfile to */var/log/mail.log* and the port to 8888.

Listing 13.5 Compile and start the LogEventBroadcaster

```
$ mvn clean package exec:exec -Pchapter13-LogEventBroadcaster /
-Dlogfile=/var/log/mail.log -Dport=8888 -....
....
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ netty-in-action -
 LogEventBroadcaster running
```

When you see "`LogEventBroadcaster` running," you'll know it started up successfully. If there are errors an exception message will be printed. Once the process is running, it will broadcast any new log messages that are added to the logfile.

Using *netcat* is adequate for testing purposes but not suitable for a production system. This brings us to the second part of our application, the monitor we'll implement in the next section.

13.6 Writing the monitor

Our goal is to replace *netcat* with a more complete event "consumer," `EventLogMonitor`. This program will do the following:

- receive UDP `DatagramPacket`s broadcast by the `LogEventBroadcaster`
- decode them to `LogEvent` messages
- write the `LogEvent` messages to `System.out`

As before, the logic will be implemented by custom `ChannelHandler`s. Figure 13.4 depicts the `ChannelPipeline` of the `LogEventMonitor` and shows how `LogEvent`s will flow through it.

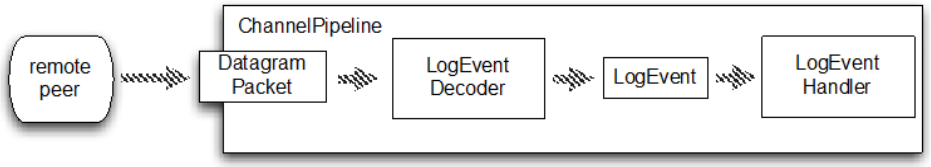


Figure 13.4 `LogEventMonitor`

The figure shows our two custom `ChannelHandlers`, `LogEventDecoder` and `LogEventHandler`. The first is responsible for decoding `DatagramPackets` received over the network into `LogEvent` messages (a typical setup for any Netty application that transforms incoming data). Listing 13.6 shows the implementation.

Listing 13.6 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 idx = data.indexOf(0, data.readableBytes(),
                LogEvent.SEPARATOR);                                   #2
        String filename = data.slice(0, idx)
            .toString(CharsetUtil.UTF_8);                              #3
        String logMsg =  data.slice(idx + 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 add it to the list

The second `ChannelHandler` will perform some processing on the `LogEvent` messages created by the first. In this case we will simply write them to `System.out`. In a real-world application you might aggregate them with other events in a separate logfile or post them to a database. The following listing, which shows the `LogEventHandler`, just exposes the basic machinery.

Listing 13.7 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 On exception print the stack trace and close the channel
#3 Create a StringBuilder and build up the output
#4 Print out the LogEvent data

The `LogEventHandler` prints out the `LogEvents` in an easy-to-read format that consists of the following:

- received timestamp in milliseconds
- `InetSocketAddress` of the sender, which consist of the IP address and port
- the absolute name of the file the `LogEvent` was generated from
- the actual log message, which represents one line in the log file

Now we need to install our handlers in the `ChannelPipeline` as shown in Figure 13.4. The next listing shows how this is done as part of the `LogEventMonitor` class.

Listing 13.8 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)                                          #1
            .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());        #2
                        pipeline.addLast(new LogEventHandler());
                    }
                }
            )
            .localAddress(address);                                                }

    public Channel bind() {
        return bootstrap.bind().sync().channel();                       #3
    }

    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(                  #4
```

```
                new InetSocketAddress(args[0]));
            try {
                Channel channel = monitor.bind();
                System.out.println("LogEventMonitor running");
                channel.closeFuture().sync();
            } finally {
                monitor.stop();
            }
        }
}
```

#1 Bootstrap the NioDatagramChannel. Set the SO_BROADCAST socket option.
#2 Add the ChannelHandlers to the ChannelPipeline.
#3 Bind the channel. Note that there is no connection when using DatagramChannel because those are connectionless.
#4 Construct a new LogEventMonitor.

13.7 Running the LogEventBroadcaster and LogEventMonitor

As above we'll use Maven to run the application. This time you will need to open two console windows, one for each of the programs. Each will keep running until you stop it with Ctrl-C.

First we'll start the `LogEventBroadcaster` as shown in Listing 13.4, except that having already build the project the following command will suffice (using the default values):

```
$ mvn exec:exec -Pchapter13-LogEventBroadcaster
```

As before, this will broadcast the log messages via UDP.

Now, in a new window, build and start the `LogEventMonitor` to receive and display the broadcast messages.

Listing 13.9 Compile and start the LogEventBroadcaster

```
$ 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'll know it started up successfully. On error an exception message will be printed.

The console will display any events as they are added to the logfile, as shown below. The format of the messages is that created by the `LogEventHandler`.

Listing 13.10 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 don't have access to a UNIX *syslog*, you can create a custom file and supply content manually to see the application in action. The steps show below use UNIX commands, starting with *touch* to create an empty file.

```
$ touch ~/mylog.log
```

Now start up the `LogEventBroadcaster` again and point it to the file by setting the System property:

```
$ mvn exec:exec -Pchapter13-LogEventBroadcaster -Dlogfile=~/mylog.log
```

Once the `LogEventBroadcaster` is running, you can manually add messages to the file to see the broadcast output in the `LogEventMonitor` console. Just use *echo* and direct the output to the file as shown below:

```
$ echo 'Test log entry' >> ~/mylog.log
```

Note that you can start as many instances of the monitor as you like; each will receive and display the same messages.

13.8 Summary

This chapter provided an introduction to connectionless protocols such as UDP. We saw that with Netty you can switch from TCP to UDP while still using the same APIs. You also saw how to organize processing logic via specialized `ChannelHandler`s. We did this by separating the decoder logic from the logic that handles the message object.

In the next chapter we'll explore implementing reusable codecs with Netty.

14

Implement a custom codec

This chapter covers

- 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. According to Memcached.org, Memcached is a "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 straightforward; this makes it ideal for inclusion in a single chapter, without letting it become too big.

14.1 Scope of the codec

We'll implement only 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 focusing on just three of them allows us to better explain what's needed while keeping things straight. Everything you learn here could be used to later implement the missing commands.

Memcached has a binary protocol and a plain-text protocol. Both of them can be used to communicate with a Memcached server, depending on whether the server supports either one of them or both. This chapter focuses on implementing the binary protocol because users are often faced with implementing protocols that are binary.

14.2 Implementing the Memcached codec

Whenever you're about to implement a codec for a given protocol, you should spend some time to understand its workings. Often the protocol is documented in great detail. How much detail you will find varies. Fortunately, the binary protocol of Memcached is written down in great extent. The specification for this is written up in an RFC-style document and available at the project's homepage[45].

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

45 https://code.google.com/p/Memcached/wiki/MemcacheBinaryProtocol

things simple and not extend the scope of the chapter. You can easily adapt the classes and so add support for other operations yourself. The code that shown serves as an example and should not be taken as a full implementation of the protocol itself.

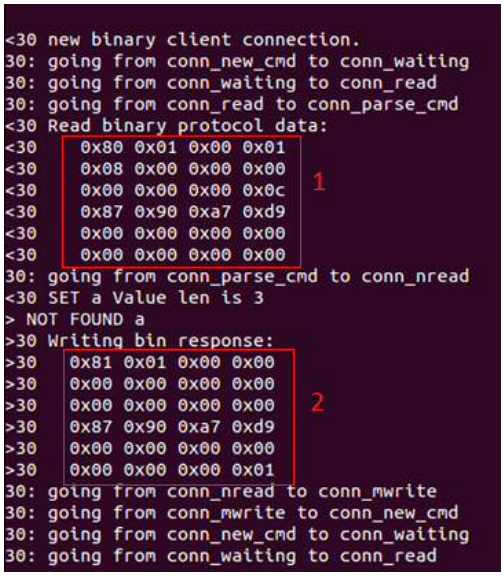
14.3 Getting to know the Memcached binary protocol

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-byte 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 of 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

Notice how many bytes are used in each section. This tells you what data type you should use later. For example, if a byte offset uses only byte 0, then you use a Java byte to represent it; if it uses 6 and 7 (2 bytes), you use a Java short; if it uses 12–15 (4 bytes), you use a Java int, and so on.



#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 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." Section 2 is the response.

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 you 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 that 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; for example, `MessageToByteEncoder<MemcachedRequest>` says this encoder encodes objects of the type `MemcachedRequest`.

MessageToByteEncoder and generics

As we said, it's possible to use generics to tell the `MessageToByteEncoder` to handle a 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 providing an encode method provides a decode method. In the next two sections you'll see how you can implement a Memcached decoder and encoder. Before you 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 doesn't have built-in support for Memcached. If the protocol was HTTP or another one of the many standard protocols Netty supports, then there would already be an encoder and decoder provided by Netty.

Encoder vs. decoder

Remember, an encoder handles outbound and a decoder handles inbound. This basically means the encoder will encode data that's about to get written to the remote peer . The decoder will handle data that was read from the remote peer.

It's important to remember that there are two different directions related to outbound and inbound.

Be aware that our encoder and decoder don't verify any values for max size to keep the implementation simple. In a real-world implementation you'd most likely want to put in some verification checks and raise an `EncoderException` or `DecoderException` (or a subclass of them) if a protocol violation is detected.

14.4.1 Implementing the Memcached encoder

This section puts our brief introduction to encoders into action. As we mentioned, the encoder is responsible for encoding 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'll first create the `MemcachedRequest` class, which we'll later encode into a series of bytes with the actual encoder implementation. The following listing 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 that 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 Indicates if 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's 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, as shown in the next listing.

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 you construct a new `MemcachedRequest` to specify which action should be triggered by it.

But let's concentrate on the encoder for now, which will encode the previously created `MemcachedRequest` class into a series of bytes. For this we extend the `MessageToByteEncoder`, which is a perfect fit for this use case. The next listing 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's responsible for encoding 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 because it currently isn't 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 is the steps are as follows:

- Write magic byte.
- Write opcode byte.
- Write key length (2 bytes).
- 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

Because we need to convert a `MemcachedRequest` object into a series of bytes, Memcached will return only 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, as shown in the following listing.

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 that was sent 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 it's binary or text based
#5 The status of the response, which indicates if the request was successful
#6 The unique id
#7 The compare –and-set value
#8 The extra flags used
#9 Indicates if the value stored for this response will expire at some point
#10 The key for which the response was created
#11 The actual data

The previously created `MemcachedResponse` class will now be used in our decoder to represent the response sent back from the Memcached server. Because we want to decode a series of bytes into a `MemcachedResponse`, we'll use the `ByteToMessageDecoder` base class. The next listing 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 responsible for creating the MemcachedResponse out of the read bytes
#2 Represents current parsing state, which means we need to parse either 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 previously 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, this code checks to see if the number of readable bytes available is at least 24. Once we have the first 24 bytes, we can determine how big the entire message is because this information is contained within the 24-byte 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 of it.

14.5 Testing the codec

With the encoder and decoder completed, our codec is in place. But there is still something missing: tests.

Without tests you'll only see if the codec works when running it against some real server, which is not what you should depend on. As shown in chapter 10, writing tests for a custom `ChannelHandler` is usually done via `EmbeddedChannel`.

So this is exactly what we'll do now to test our custom codec, which includes an encoder and decoder. Let's start with the encoder again. The listing that follows 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 if they were written
#9 Check if the data type was written
#10 Check if the reserved data was inserted
#11 Check for 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
#17 ???

After tests for the encoder are in place, it's time to take care of the tests for the decoder. Because we don't know whether the bytes will come in all at once or fragmented, we need to take special care to test both cases. The following listing 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 that matches the structure of the binary protocol
#3 Write the buffer to the EmbeddedChannel and check if a new MemcachedResponse was created by asserting 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 that matches the structure of the binary protocol
#7 Split the Buffer into three fragments
#8 Write the first fragment to the EmbeddedChannel and check that no new MemcachedResponse was created, because not all data is ready yet
#9 Write the second fragment to the EmbeddedChannel and check that no new MemcachedResponse was created, because not all data is ready yet
#10 Write the last fragment to the EmbeddedChannel and check that a new MemcachedResponse was created because 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 serve as an example of how you'd use 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 nonfragmented data
- Test validation of received data/sent 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 vice versa. This chapter showed how you can use a protocol specification and extract the needed information for the implementation.

It also showed you how to complete your work by writing 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'll take a deeper look into the thread model Netty uses and how it's 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

- Overview of 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 or framework. Netty comes with an easy-to-use but powerful thread model that helps you simplify your code. All of your `ChannelHandler`s, 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 nonblocking 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'll 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 multithreaded 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.

Although we make these assumptions, if you're new to multithreaded applications, having a general understanding of the core concepts should be enough for you to gain some insights from this chapter.

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.

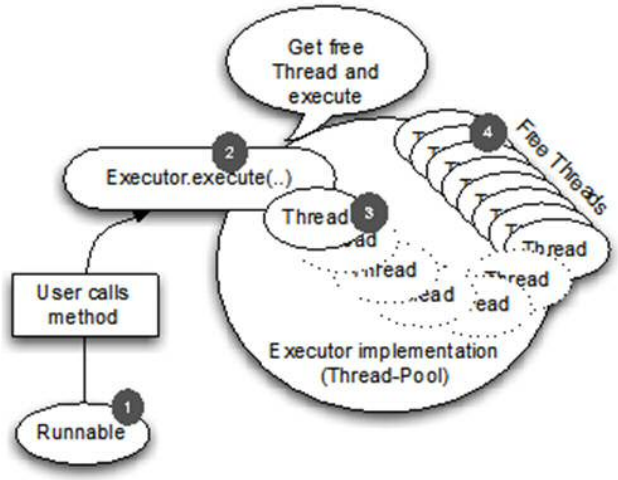
A thread model specifies how code is executed and gives the developer information on how their code will be executed. This is important because it allows the developer to know in advance how to guard their code from side effects by concurrent execution. Without this knowledge the best a developer could do would be 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, we'll provide 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 because most hardware comes with more than one core or even more than one CPU these days. If everything would be executed in only one `Thread`, it would be impossible to make proper use of the provided resource. To fix this problem, many applications execute the running code in multiple `Thread`s. In the early days of Java, this was done by simply creating new `Thread`s 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 `Thread`s and recycling them comes with overhead. In Java 5 we finally got so-called thread pools, which often cache `Thread`s 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 `Thread`s 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 that represents the task to execute. This could be anything from a database call to file system cleanup.
#2 Previous runnable gets handed over to the thread pool.
#3 A free Thread is picked and the task is executed with it. When a thread has finished running, it's 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 `Thread`s to be created and destroyed with each new task.

But using multiple `Thread`s 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 increase. Although 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 you do to solve this problem? You need multiple `Thread`s 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, because 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.

The following listing shows the typical logic of an `EventLoop` that behaves the same way as explained. Be aware this is more to illustrate the idea than 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 you have events that are ready to run
#2 Loop over all of the events and run them

The event loop is represented by the `EventLoop` interface in Netty. Because 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`. It's also possible to schedule tasks for later execution, just as you can do with any `ScheduledExecutorService` implementation. The class/interface hierarchy of `EventExecutor` is shown in figure 15.2 (shown in only one depth).

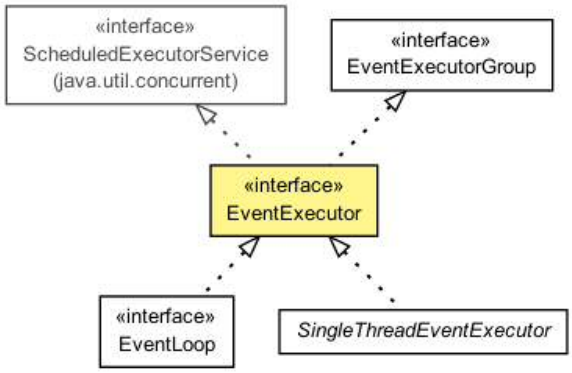


Figure 15.2 EventLoop class hierarchy

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 necessary because otherwise events could be processed out of order and so the bytes that are processed wouldn't be guaranteed to be in the right order. This would lead to corruption and so isn't allowed by design.

15.2.1 I/O and event handling in Netty 4

Netty uses I/O events, which are triggered by various I/O operations on the transport itself. These I/O operations are, for example, 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 some data is read. In contrast, a write event is usually triggered by the user itself, for example, 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, transfer 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's assigned to the `EventLoop`.

We should note that Netty didn't always use the thread model (abstracted via the `EventLoop`) that we described. 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 why it 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 would be executed in the I/O `Thread` (the I/O `Thread` was what's now 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 because handling of outbound events in the `ChannelHandler`s needed careful synchronization, because it wasn't guaranteed that only one `Thread` would operate on them at the same time. This could happen if you fired downstream events at the same time for one `Channel`; for example, you called `Channel.write(..)` in different threads.

In addition to the burden placed on you to synchronize `ChannelHandler`s, another problematic side effect of this thread model was that you may have needed to fire an inbound (upstream) event as a result of an outbound (downstream) event. This would be true, for example, if your `Channel.write(..)` operation caused an exception. In that case, an `exceptionCaught` had to 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 lies. The problem was that you then had a situation where the code was executed in your calling `Thread` but the `exceptionCaught` event had to be handed over to the worker thread for execution and so a context switch was needed again.

In contrast, the new thread model of Netty 4 doesn't have these problems, because everything is executed in the same `EventLoop` and so in the same `Thread`. This eliminates the need for synchronization in the `ChannelHandler` and makes it much easier for the user to understand 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.

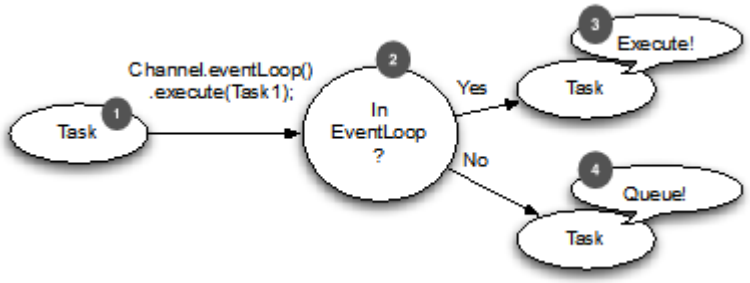
15.2.3 Netty's thread model internals

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's 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 `ChannelHandler`s don't need to worry about concurrent access.

It's important to know that each `EventLoop` has its own task/event queue and so is not affected by other `EventLoop`s 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 be 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.

Because 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 block calls (or execute tasks that can take long periods to complete), you must 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, which 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 shows 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, it provides more details about the internals of Netty's implementation and list what advantages and limitations it provides.

15.3.1 Scheduling tasks with plain Java API

To schedule a task you typically use the provided `ScheduledExecutorService` implementations that ships with the JDK. This wasn't always true. Before Java 5 you had to use 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 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 the next listing 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 that 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 oncethe task is 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 thing 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 can 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 the following listing.

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 the next listing.

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 `ScheduledFuture` that's returned for every asynchronous operation. The `ScheduledFuture` provides a method for canceling a scheduled task or checking its state.

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 for the Efficient Implementation of a Timer Facility," by George Varghese and Tony Lauck. This kind of implementation only guarantees an approximated execution, which means that the execution of the task may not be 100% 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% 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 to see if tasks need to get executed then.

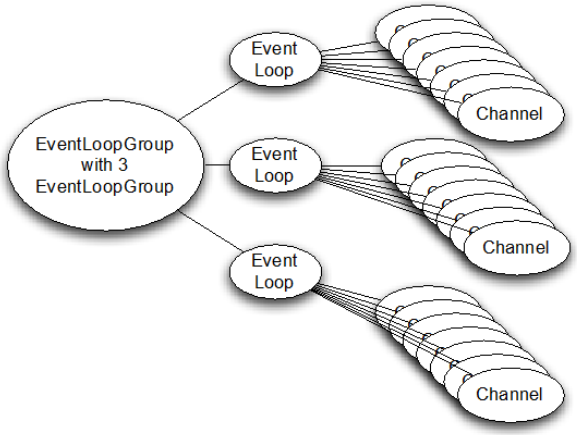
4. If there's a task, the `EventLoop` will execute it right away and remove it from the queue.
5. The `EventLoop` waits for the next run and starts over again with step 4.

Because of this implementation, the scheduled execution may be not 100% accurate. 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 `EventLoop`s are created and assigned varies based on the transport implementation. An asynchronous implementation uses only a few `EventLoop`s (and so `Threads`) that are shared between the `Channel`s. This allows a minimal number of `Thread`s to serve many `Channel`s, 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 EventLoop 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 are bound to the EventLoop, and so all operations are always executed by the same thread during the lifetime 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 `EventLoop`s (and so the `Thread`s) 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 `Channel`s 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.

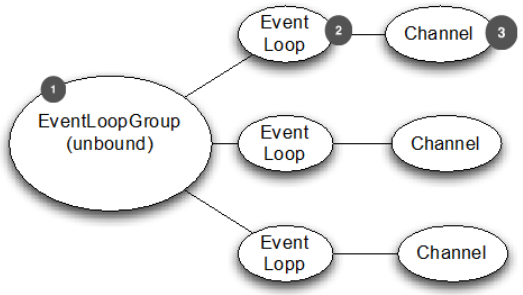
Once a `Channel` is assigned to an `EventLoop`, it will use this `EventLoop` throughout its lifetime and likewise the `Thread` that powers the `EventLoop`. You can, and should, depend on this, because it ensures that you don't need to worry about synchronization (which includes thread safety, visibility, and synchronization) in your `ChannelHandler` implementations.

But this also affects the usage of `ThreadLocal`, for example, which is often used by applications. Because an `EventLoop` usually powers more than one `Channel`, the `ThreadLocal` will be the same for all `Channel`s that are assigned to the `EventLoop`. Thus, it's a bad fit for state tracking and so on. It can still be useful for sharing heavy or expensive objects between `Channel`s that no longer need to keep state, and so it 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's possible in Netty 4 to deregister a `Channel` from its `EventLoop` and register it with a different `EventLoop` later. This functionality was deprecated because it didn't work out 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 one `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 be handled by only 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 time two limitations still need to be addressed in Netty 5 that are a result of the current thread model (Netty 4). This section will explain these limitations and show 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 fewer `Channel`s assigned than others. This is even more likely if the `EventLoopGroup` (which contains the `EventLoop`s) handles different protocols or different kind of connections, such as long-living and short-living connections.

A fix may sound easy at first, because you could just choose the `EventLoop` that has the smallest `Channel` count. But taking into account the number of `Channel`s on the `EventLoop` isn't enough for a proper fix, because even if one `EventLoop` holds more `Channel`s than another it doesn't mean automatically that this `EventLoop` is busier. There many things that need to be considered. These include the following:

- Number of queued tasks
- Last execution times
- How saturated the network stack is

The problem is tracked as part of issue 1230[46] and currently targeted for Netty 5.0.0.Final but may be back-ported to Netty 4 if it can be implemented without any API breakage.

15.5.2 Redistributing Channels across the EventLoops

The other problem is how to redistribute `Channel`s across the `EventLoop`s and so better make use of the resources. This is especially important for long-living connections, because chances are better here that some `EventLoop`s will get busier than others over time.

To fix , the Netty team is currently working on introducing a `reregister(....)` method, which allows you to move a `Channel` to another `EventLoop` and so move over its handling to another `Thread`. But this isn't as easy as it sounds, because it brings a few problems with it.

---

[46] https://github.com/netty/netty/issues/1230

First, you must ensure that all events and operations that were triggered while the old `EventLoop` was registered are executed while still on the old `EventLoop`. Failing to do so would break the thread model, which guarantees that each `ChannelHandler` is processed by only one `Thread`. Another issue is how to make sure that you don't encounter any visibility issues when you move from one `Thread` to another.

This problem is currently tracked as issue 1799[47] and targeted for Netty 5.0.0.Final. It's not likely that it will be ported to Netty 4.x because 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 learning the inner workings, you gained insight as to 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 one it replaced.

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.

[47] https://github.com/netty/netty/issues/1799

16

Case studies, part 1: Droplr, Firebase, and Urban Airship

This chapter covers

- Case studies
- Real-world use cases

In previous chapters you learned a lot about the details of Netty and how you can use them. This is great, but often it's helpful to see how others have empowered their applications to use Netty.

This chapter is all about companies that use Netty. In this chapter we cover three different companies that make heavy use of Netty for their internal infrastructure. You'll see how they used Netty to solve their problems and may be able to use their solutions to solve your own.

The companies are

- Droplr
- Firebase
- Urban Airship

Every case study was written by the people who were part of the effort to use Netty within those companies. The case studies are ordered alphabetically by the name of the companies and so the order doesn't reflect any relevance here.

16.1 Droplr—building mobile services

Written by Bruno de Carvalho, lead architect at 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 was 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: you 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, and so on—and the files are stored on Amazon S3.

16.1.3 Creating a faster upload experience

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 time out just waiting for the okay from the server. Back then Droplr could only offer uploads of up to 32 MB 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, shown in figure 16.1:
- Fully receive the file.
- Save to local filesystem and immediately return success to client.
- Schedule an upload to S3 some time in the future.

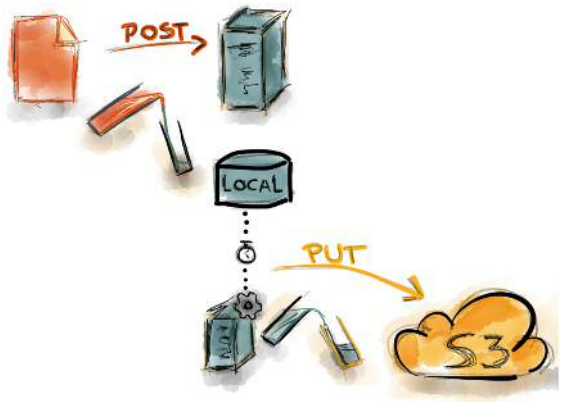


Figure 16.1 Option A, the optimistic and apparently simpler approach

- Option B, the safe but complex approach:
- Pipe the upload from the client directly to S3, in real time (streaming), as shown in figure 16.2.

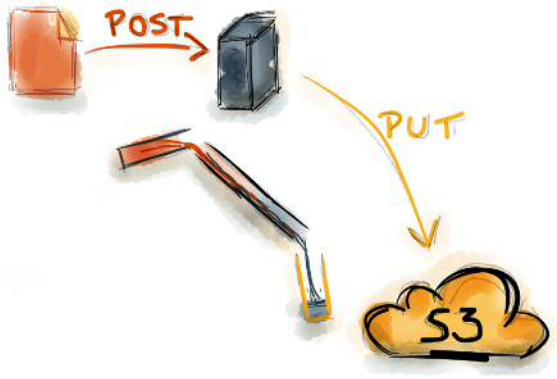


Figure 16.2 Option B, the safe but complex approach

THE OPTIMISTIC AND APPARENTLY SIMPLER APPROACH

Returning a short URL after receiving the file creates expectation—one could even go as far as to call it *an implicit contract*—that the file is immediately available at that URL. Because Droplr was unable to guarantee that the second stage of the upload would *eventually* succeed (that is, actually push the file to S3), the user could end up with a broken link, a link they could have posted on Twitter or sent to an important client. This is unacceptable, even if it happens to only one in every hundred thousand uploads.

Our current numbers show that we have an upload fail rate slightly below 0.01% (1 in every 10,000), the vast majority being connection timeouts between client and server before the upload completes.

We could try to 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 might occur across the cluster (for example, „Where is the file for this drop?").
- Extra, complex logic is required to deal with edge cases, which keeps creating more edge cases.

Thinking through all the pitfalls with every workaround, you'll 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 do the following:

- 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.

This 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 also the flexibility to go *bare metal* (read *down to the bytes*) if required.

16.1.4 The technology stack

When kick-starting a new project for a webserver, you'll end up asking yourself "Okay, so what framework are the cool kids 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:

- *It had to be fast*—I wasn't about to replace a low-performance stack with another low-performance stack.
- *It had to scale*—Whether it had 1 connection or 10,000 connections, each server instance would have to be able to sustain throughput without crashing or leaking

memory over time.

- *It had to offer me low-level data control*—This means bytes, trigger TCP congestion control, the works.

The first two factors pretty much excluded any noncompiled language. I'm a sucker for Ruby and love lightweight frameworks like Sinatra and Padrino, but I knew the kind of performance I was looking for couldn't be achieved by building upon these blocks.

The second factor, 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 nonblocking I/O was the only option.

The third factor 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 the following listing.

Listing 16.1 Set up 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's responsible for the following:

- Handling load peaks
- HTTP pipeline management

- Setting up a context for request handling
- Spawning new request handlers
- Feeding request handlers
- Handling internal and external errors

The next listing does a quick run-down of the relevant parts of the `RequestController`.

Listing 16.2 Set up 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 the gist of Droplr's server request validation
#2 If there's 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 because they perform any logic that isn't confined to local memory or CPU.

When thread switching occurs, it's imperative that all the blocks of code execute in serial fashion; otherwise we'd risk, for example, for an upload, having an `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're the endpoints behind URIs such as `/account` or `/drops`. They're 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 subclass hierarchy, is a realization of 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 these:

- Determine whether the request handler is stateful or stateless. That is, does it need to be cloned from a prototype or can the prototype be used to handle the request?
- Determine whether the request handler is CPU or non-CPU bound. That is, can it execute on Netty's worker threads or should it be executed in a separate thread pool?
- Create a new, freshly uninitialized ???.
- Roll back current changes.
- Clean up any used resources.

This interface is all the request controller knows about actions. Through its very clear and concise interface, the controller can interact with both with stateful and stateless CPU and non-CPU-bound handlers (or combinations of the previous) in a completely isolated and implementation-agnostic fashion.

HANDLER IMPLEMENTATIONS

Through a subclass 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 that 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 an http client library [48] that's also built with Netty. This asynchronous library exposes an interface that perfectly matches the needs of the server—begin executing an 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.

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 10~12 times faster uploads over the old LAMP stack—a full order of magnitude faster!—and could handle over 1000 times more concurrent uploads, in a total of nearly 10K concurrent uploads (running on a single EC2 large instance).

Factors that contributed to this were the following:

- 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 meant more parallel uploads could take place.
- Thumbnail generation became an asynchronous post process.

[48] You can find the http client library at https://github.com/brunodecarvalho/http-client

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 nonblocking 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 full-stack upgrade (databases, OS, major server, and 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 because I start the process with 1 GB of preallocated memory. (But I do configure the JVM to grow up to 2 GB 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:

- Long polling
- HTTP 1.1 keep-alive and pipelining
- 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 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 figure 16.3.

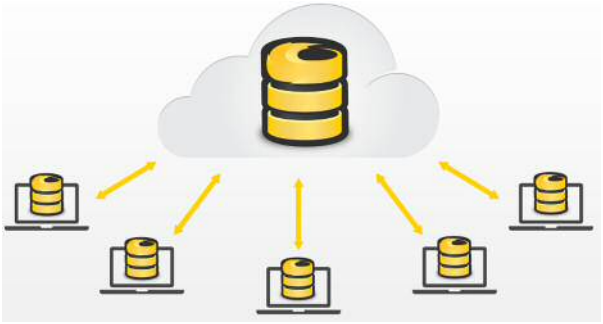


Figure 16.3 Firebase architecture

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 web-socket transports. The long-polling transport is highly reliable across all browsers, networks, and carriers, whereas the web-socket-based transport is faster but not always available. Initially, Firebase connects using long polling and then upgrades to web sockets if possible. For the 5% of Firebase traffic that doesn't support web sockets, 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. Although 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

- Guaranteed in-order delivery of messages
- Close notifications

GUARANTEED IN-ORDER DELIVERY OF MESSAGES

In-order delivery for long polling can be achieved by having only a single request outstanding at a given time. Because 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's 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's always safe to send everything that's buffered up in between. But 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's 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 a single request for at most two requests. The algorithm has two parts:

- Whenever a client has new data to send, it sends a new request unless two are already in flight.
- 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's no data.

This has an important improvement over single request: both the client's buffer time and the server's buffer time are bound to at most a single network round-trip.

Of course, this increase in performance doesn't come without a price, because 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 because 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's 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 two things can cause this to fail: the user can close the browser tab, or the network connection can 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's important to pick your timeout values carefully, because the server is unable to distinguish a slow network from a disconnected client. That is to say, there's 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's 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 in figure 16.4 demonstrates how the Firebase long-- polling transport handles different types of requests.

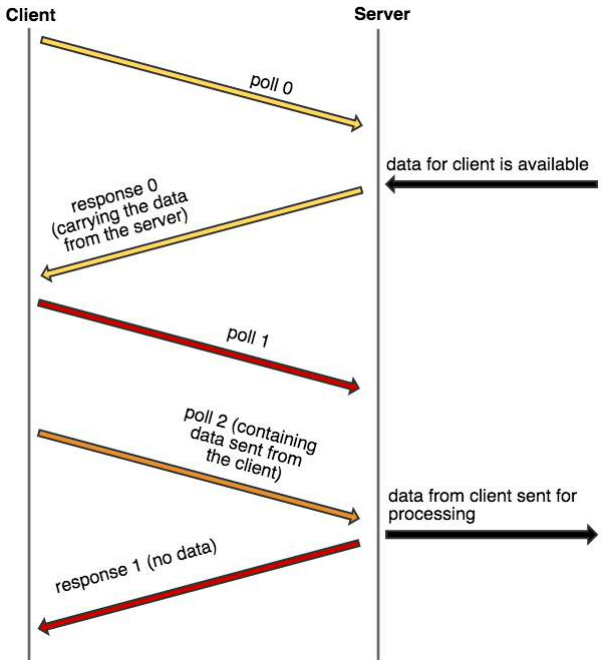


Figure 16.4 Long polling

In this diagram, each long-poll request indicates a different type of scenario. Initially, the client sends a poll (poll 0) to the server. Some time later, the server receives data from elsewhere in the system that's 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), because it currently has none outstanding. A short time later, the client needs to send data to the server. Because it has only 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, you must take some considerations 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's an outstanding long-poll request, the long-polling transport needs to respond with a no-op before handling the REST request.

The diagram in figure 16.5 illustrates how Netty lets Firebase respond to multiple request types in one socket.

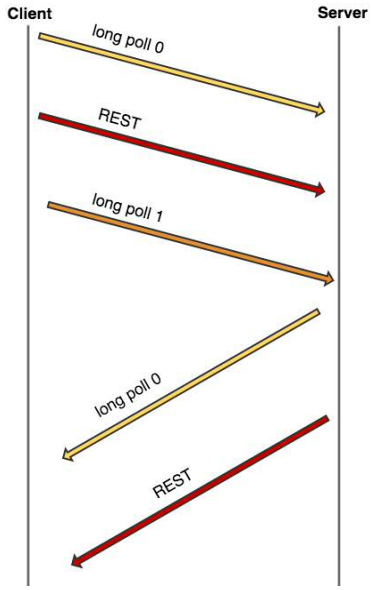


Figure 16.5 Network diagram

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 can't. 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.
- *Web sockets*—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 web sockets 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 were received by the application code. With a multitenant architecture, it's 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. But the account to be charged for a message is typically not available before the SSL decryption has been performed, because it's 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, as shown in the following listing.

Listing 16.3 Set up 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 for these reasons:

- *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 a spike in the number of packets, all of which Netty handles smoothly.
- *It's 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 wouldn't 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 smartphone usage grows across the globe at unprecedented rates, a number of service providers have emerged to assist developers and marketers toward the end of providing amazing end-user experiences. Unlike their feature phone predecessors, these smartphones crave IP connectivity and seek it across a number of channels (3G, 4G, Wi-Fi, 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

Although 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 smartphones. 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 16.6.

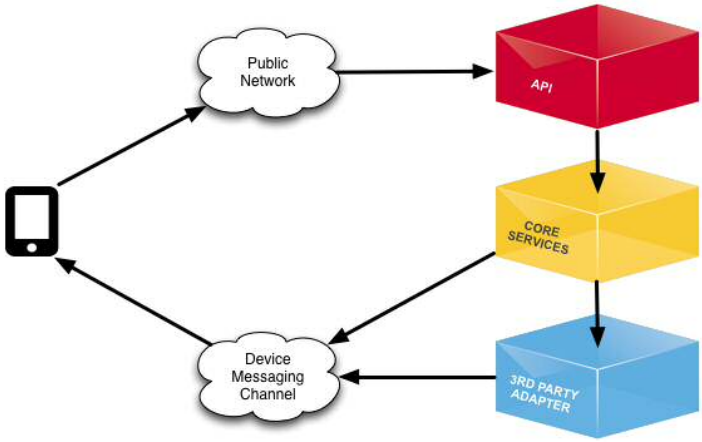


Figure 16.5 High-level mobile messaging platform integration

At a high level, when an application developer desires to send push notifications to a device[49], 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.

Once an application installation has registered its identifier with a back-end service, the delivery of a push message can take either of 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 a 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 third-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, and so on), and capabilities.

[49] Some mobile operating systems allow a form of push notifications called local notifications that would not follow this approach.

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, whereas 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 smooth 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.
2. Format a binary representation of a push message structured according to the format defined by Apple[50].
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's 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.

Although at face value the protocol seems straightforward, there are several nuances to successfully addressing all of the previous concerns, 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 3 in the previous sequence requires one of two solutions. Because the JVM won't allow reading from a closed socket, even if data exists in the output buffer, developers have the options of:

[50] http://bit.ly/189mmpG

- 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.
- Because socket objects can't 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 tens of millions of messages, a 3-second delay between messages isn't acceptable.
- Relying on a socket timeout is an expensive operation. It results in an exception being thrown and several unnecessary system calls.
- Under the Asynchronous I/O 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, as shown in the following listing, Netty makes packing a binary APNS message with correct endian ordering trivial.

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's 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 2-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, as shown here.

Listing 16.5 Set up 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's 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 clear-text bytes to cipher text and the key renegotiation that comes 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 role in all connectivity to third-party push-notification services, including APNS, Google's GCM, and various other 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 can't 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 is 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 don't 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 because it's 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 one 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[51]. 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 one 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 capable of handling even further connections, all on a 4 GB 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's 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 perimeter of the firewall

The previous two sections of this chapter have demonstrated two everyday usages of Netty at the perimeter of the Urban Airship network. Although 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 hundreds of thousands of requests per second with very low latency and exceptional throughput. Nearly every API request fielded by

---

[51] 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.

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[52] instance to test end-to-end message throughput with a minimal client-side footprint.

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[53] and Memcached[54]. 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 got some insight into real-world usage of Netty and how it helped companies to solve their problems and build up their stack. Hopefully this should give you a better idea of how you can solve your problems and use Netty in your next project.

That said, there are also other companies that started to build open-source projects on top of Netty, which itself is used to power their internal needs. In the next chapter you'll learn about two of those open-source projects that were founded by Facebook and Twitter.

[52] http://redis.io/
[53] http://kafka.apache.org/
[54] http://memcached.org/

17

Case studies, part 2: Facebook and Twitter

This chapter covers

- Case studies
- Real-world use cases
- Building your own project on top of Netty

In this chapter you'll see how two of the most popular social networks worldwide 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 possible only because of the very flexible and generic design of Netty. The details in this chapter will serve as inspiration but also to show how some problems can be solved by using Netty. You'll learn from the experience and issues of Facebook and Twitter, and you'll understand how to solve the problems with Netty. Both case studies are written by the engineers themselves, who are responsible for the design and implementation of the various solutions. You'll also understand how their use of Netty and the design choices they made allow them to scale and keep up with future needs.

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, and so on), 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[55] 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, because services and their clients usually can't all be upgraded simultaneously.

Another important feature of Thrift is that it's 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 the back-end services at Facebook communicate with one another, and it's also used for non-RPC serialization tasks, because it provides a common, compact storage format that can be read from a wide selection of languages for later processing.

[55] A now-ancient whitepaper from the original Thrift developers can be found here: http://thrift.apache.org/static/files/thrift-20070401.pdf

Since its development at Facebook, Thrift has been open-sourced as an Apache project[56], where it continues to grow to meet the needs of service developers, not only at Facebook but also at other companies—including Evernote and last.fm[57]—and on major open-source projects such as Apache Cassandra and HBase.

The major components of Thrift are as follows:

- *Thrift Interface Definition Language (IDL)*—Used to define your services and compose any custom types that your services will send and receive
- *Protocols*—Control encoding/decoding elements of data into a common binary format (for example, Thrift binary protocol or JSON)
- *Transports*—Provide a common interface for reading/writing to different media (for example, 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 20 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 languages used for writing Thrift services here at Facebook are C++ and Java.

When I arrived at Facebook, we were already well under way 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 I/O, 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[58] 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 asynchronous task to handle a request and return immediately and then invoke a callback or set a future later when the response is ready).

---

[56] Find more info at http://thrift.apache.org/
[57] Find more examples at http://thrift.apache.org/about/
[58] Folly is Facebook's open-source C++ common library: https://www.facebook.com/notes/facebook-engineering/folly-the-facebook-open-source-library/10150864656793920

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 didn't support pipelining or multiplexing requests, and the servers didn't support asynchronous request handling. Because of these missing features, we Java Thrift service developers 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[59] is an open-source, Apache-licensed Thrift client/server implementation built on top of the Apache Thrift library. It's 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 4-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.

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

[59] https://github.com/facebook/nifty

message (containing optional metadata, client capabilities, and the like). 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 and 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 that 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.

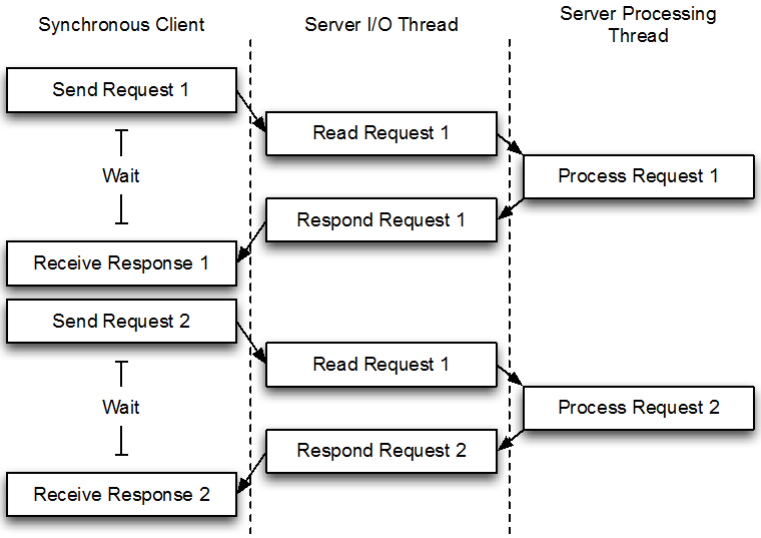


Figure 17.1 Request–response flow

Initial pseudo-asynchronous usages of clients started happening when a few Thrift users took advantage of the fact that for a method, generated client methods `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* handle 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 guarantees 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.

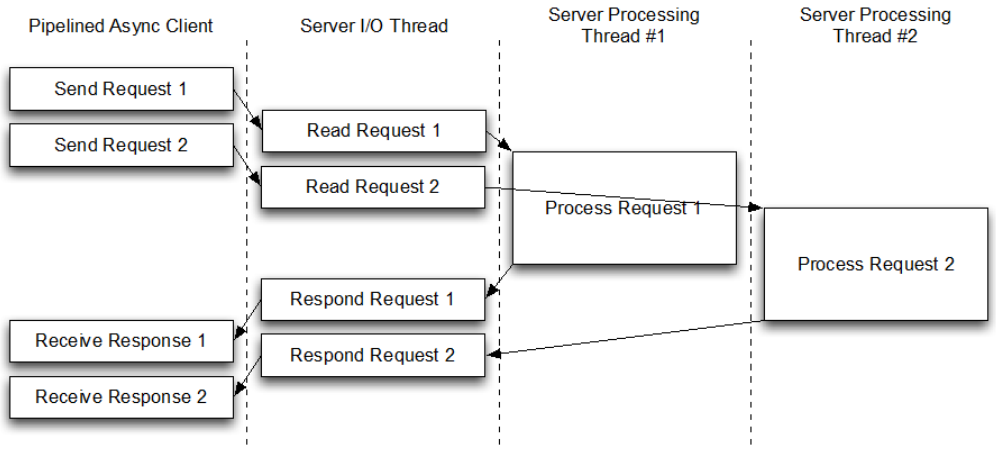


Figure 17.2 Request–response flow

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 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.

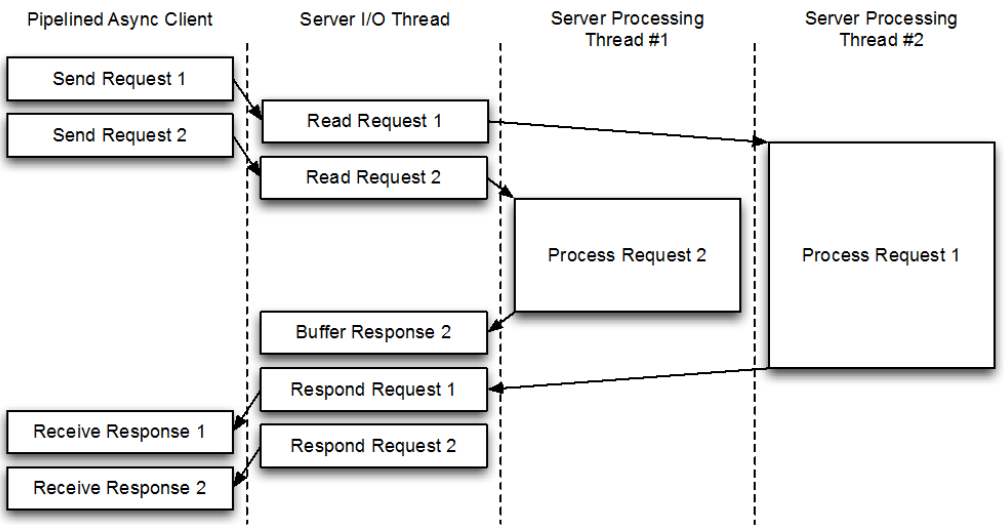


Figure 17.3 Request–response flow

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 Thrift's synchronous transport interface, but its usage is pretty limited because it doesn't provide much improvement 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 it 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 its 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.

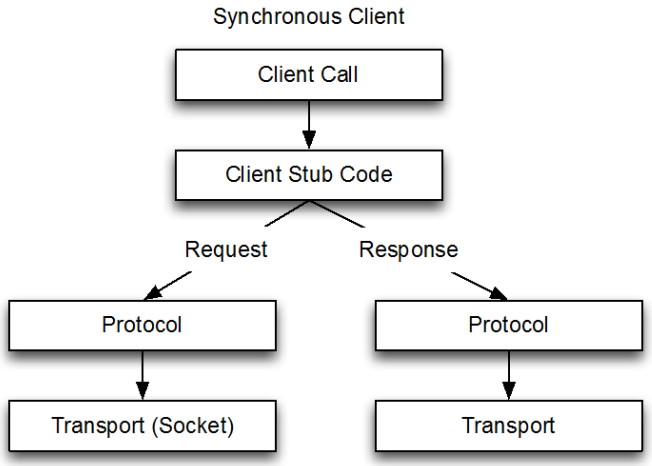


Figure 17.4 Multiplexing/transport layers

That simple flow (shown 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.

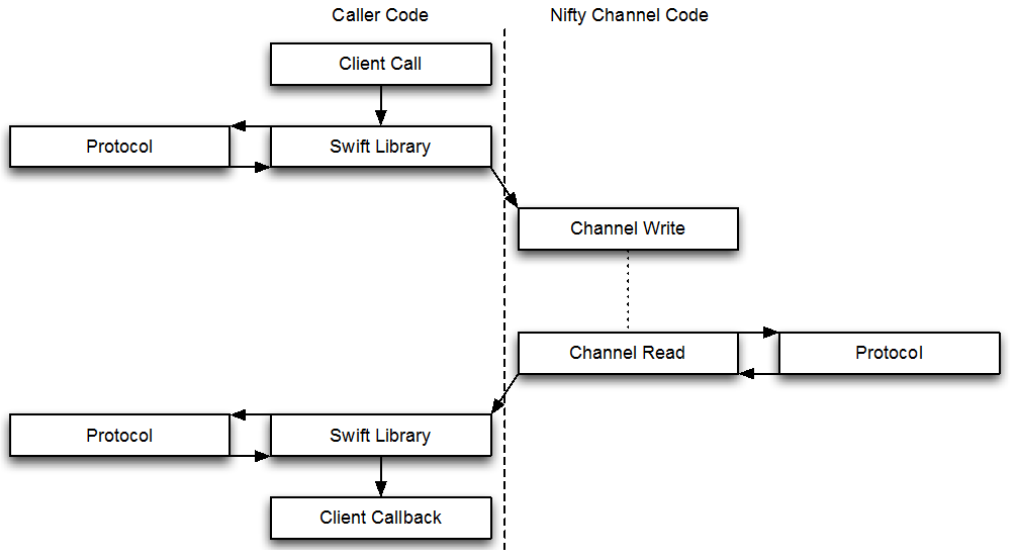


Figure 17.5 Dispatching

17.1.5 Swift: a faster way to build a Java Thrift service

The other key part of our new Java Thrift framework is called Swift. It uses Nifty as its I/O 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 Thrift 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 (for example, asynchronous clients) are supported in Swift first. If you're interested, take a look at the introductory information on Swift's github page[60].

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 empty response back. This measurement isn't a realistic performance estimation of most

[60] https://github.com/facebook/swift

actual Thrift services, but it's 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. See table 17.1.

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 in table 17.1, 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, but 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 although it falls a bit short compared to our next-gen C++ server framework, it's 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 code base that had more time to mature, but because its 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 had 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 because 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, but 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.

But 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 doesn't use timeout sets. Netty solves this problem with its HashedWheelTimer[61]. 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.

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 v4 has been finalized. We're eagerly looking forward to some of the benefits the Netty 4 API will offer us.

---

[61] http://netty.io/4.0/api/io/netty/util/HashedWheelTimer.html

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 I/O 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 I/O worker thread, which is something that wasn't possible in v3.

17.1.6 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 it has made a lot of new features like built-in SOCKS support for Thrift clients simple to add.

But we're not finished yet. We have 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 back ends 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 computing 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 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 teams 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. Although 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[62], PostrgreSQL[63], WebSockets[64], IRC[65], and AWS[66].

Netty's connection management and protocol agnosticism provided an excellent base from which Finagle could be built. But we had a few other requirements Netty couldn't satisfy out of the box because 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, and the like) 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[67]-inspired tracing framework[68]. Finagle was built to solve these problems.

17.2.3 How Finagle works

Internally Finagle is very modular. Components are written independently and 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

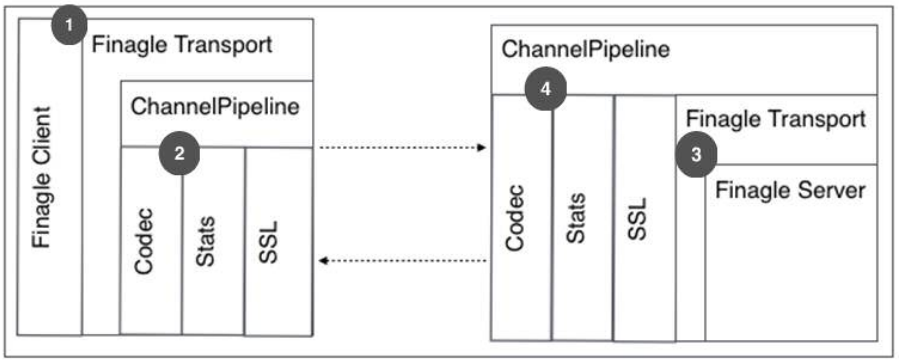
---

[62] https://github.com/twitter/finagle/tree/master/finagle-spdy
[63] https://github.com/mairbek/finagle-postgres
[64] https://github.com/sprsquish/finagle-websocket
[65] https://github.com/sprsquish/finagle-irc
[66] https://github.com/sclasen/finagle-aws
[67] http://research.google.com/pubs/pub36356.html
[68] https://github.com/twitter/zipkin

as Netty ChannelHandlers and inserted into the end of a ChannelPipeline. Messages come in from the wire, where Netty picks them up and runs them through the ChannelPipeline, where they're interpreted by a codec and 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 reuse 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 that Finagle can write to and read from.

On the server side a Netty server is created and 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 abstracts 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

The following listing shows a static `ChannelFactory` with default options.

Listing 17.1 Set up 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 that 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's fulfilled when the channel has either connected or failed to connect.

The next listing shows the `ChannelConnector`, which allows you 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 ruture 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 finishes
#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 it's 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. The next listing shows the details.[69]

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
        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
```

[69] https://github.com/twitter/finagle/blob/master/finagle-core/src/main/scala/com/twitter/finagle/netty3/client.scala

```
    ): 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. The following listing shows the Netty-based implementation of a `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. The next listing shows the code needed.[70]

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 is open and creates 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 a 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's either virtual or a concrete representation of a remote server.

For example, you can create a simple HTTP server by implementing a service that takes an HttpReq and returns a Future[HttpRep] representing an eventual response:

```
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:

[70] https://github.com/twitter/finagle/blob/master/finagle-core/src/main/scala/com/twitter/finagle/netty3/server.scala

```
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. But you 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. But there's 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 because 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.

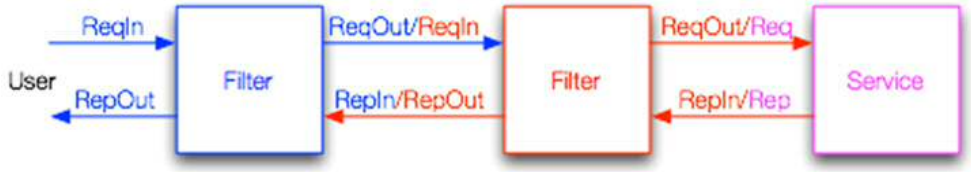


Figure 17.7 Request/response flow

You 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's a service that requires an 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 you 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 will 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 set up to manage failures in a 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 they know about. They do 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 reestablish a connection. Service owners are then free to shut down 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 these are the authentication service, timeline service, and tweet service. These relationships can be expressed succinctly, as shown in the following listing.

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 you create clients for the timeline service, tweet service, and authentication service. A filter is created for authenticating raw requests. Finally the 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 look up the user's timeline via the timeline service. A list of tweet ids is returned and 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 you leave the concurrency to Finagle. You 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 parts of Netty that both Finagle and the wider community can benefit from[71]. 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 onto 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[72]. That number would have been unthinkable on our old architecture.

Finagle was born out of a need to set up Twitter 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. Using Netty as a base, we were able to 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 is not only mission critical but is also needed to give them the right amount of performance and flexibility.

---

[71] https://blog.twitter.com/2013/netty-4-at-twitter-reduced-gc-overhead
[72] https://blog.twitter.com/2013/new-tweets-per-second-record-and-how

Facebook's Nifty project gives insight into how they were 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 using Netty's features.

Twitter's Finagle case study 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 these are possible while still obtaining a high degree of performance by using Netty in its core.

Hopefully this chapter will serve as source of inspiration to you but also as source of information that you can use in your next-generation project.

This chapter should give you an idea how you can build your next-generation framework on top of Netty and take advantage of its performance and flexibility. There's no need to handle all the low-level networking API by yourself when using Netty.