

The Netty Project 3.0 User Guide

**The Proven Approach
to Rapid Network
Application Development**

3.0.2.GA

Preface	iii
1. The Problem	iii
2. The Solution	iii
1. Getting Started	1
1.1. Before Getting Started	1
1.2. Writing a Discard Server	1
1.3. Looking into the Received Data	3
1.4. Writing an Echo Server	4
1.5. Writing a Time Server	5
1.6. Writing a Time Client	7
1.7. Dealing with Packet Fragmentation and Assembly	8
1.7.1. What is Packet Fragmentation and Assembly?	8
1.7.2. The First Solution	9
1.7.3. The Second Solution	10
1.8. Speaking in POJO instead of ChannelBuffer	12
1.9. Summary	14

Preface

This guide provides an introduction to [Netty](#) and what it is about.

1. The Problem

Nowadays we use general purpose applications or libraries to communicate with each other. For example, we often use an open source HTTP client library to retrieve information from an open source web server and to invoke a remote procedure call via web services.

However, a general purpose protocol or its implementation sometimes does not scale very well. It is like we don't use a general purpose HTTP server to exchange huge files, e-mail messages, and near-realtime messages such as financial information and multiplayer game data. What's required is a highly optimized protocol implementation which is dedicated to a special purpose. For example, you might want to implement an HTTP server which is optimized for AJAX-based chat application. You could even want to design and implement a whole new protocol which is precisely tailored to your need.

Another inevitable case is when you have to deal with a legacy proprietary protocol to ensure the interoperability with an old system. What matters in this case is how quickly we can implement that protocol while not sacrificing the stability and performance of the resulting application.

2. The Solution

The [Netty project](#) is an effort to provide an asynchronous event-driven network application framework and tooling for the rapid development of maintainable high-performance high-scalability protocol servers and clients.

In other words, Netty is a NIO client server framework which enables quick and easy development of network applications such as protocol servers and clients. It greatly simplifies and streamlines network programming such as TCP/IP socket server development.

'Quick and easy' does not mean that a resulting application will suffer from a maintainability or a performance issue. Netty has been designed carefully with the experiences earned from the implementation of a lot of protocols such as FTP, SMTP, HTTP, and various binary and text-based legacy protocols. As a result, Netty has succeeded to find a way to achieve ease of development, performance, stability, and flexibility without a compromise.

Some users might already have found other network application framework that claims to have the same advantage, and you might want to ask what makes Netty so different from them. The answer is the philosophy where it is built on. Netty is designed to give you the most comfortable experience both in terms of the API and the implementation from the day one. It is not something tangible but you will realize that this philosophy will make your life much easier as you read this guide and play with Netty.

Getting Started

This chapter tours around the core constructs of Netty with simple examples to let you get started with Netty easily. You will be able to write a network application on top of Netty right away when you are at the end of this chapter.

1.1. Before Getting Started

The minimum requirements to run the examples which are introduced in this chapter are only two; the latest version of Netty and JDK 1.5 or above. The latest version of Netty is available in [the project download page](#). To download the right version of JDK, please refer to your preferred JDK vendor's web site.

Is that all? To tell the truth, you should find these two are just enough to implement almost any type of protocols. Otherwise, please feel free to [contact the Netty project community](#) and let us know what's missing.

At last but not least, please refer to the API reference whenever you want to know more about the classes introduced here. All class names in this document are linked to the online API reference for your convenience. Also, please don't hesitate to [contact the Netty project community](#) and let us know if there's any incorrect information, errors in grammar and typo, and if you have a good idea to improve the documentation.

1.2. Writing a Discard Server

The most simplistic protocol in the world is not 'Hello, World!' but **DISCARD**. It's a protocol which discards any received data without any response.

To implement the DISCARD protocol, you only need to log the received data. Let us start straight from the handler implementation, which handles I/O events generated by Netty.

```
package org.jboss.netty.example.discard;

@ChannelPipelineCoverage("all") ❶
public class DiscardServerHandler extends SimpleChannelHandler { ❷

    @Override
    public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) { ❸
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, ExceptionEvent e) { ❹
        e.getCause().printStackTrace();

        Channel ch = e.getChannel();
        ch.close();
    }
}
```

- ① `ChannelPipelineCoverage` annotates a handler type to tell if the handler instance of the annotated type can be shared by more than one `Channel` (and its associated `ChannelPipeline`). `DiscardServerHandler` does not manage any stateful information, and therefore it is annotated with the value "all".
- ② `DiscardServerHandler` extends `SimpleChannelHandler`, which is an implementation of `ChannelHandler`. `SimpleChannelHandler` provides various event handler methods that you can override. For now, it is just enough to extend `SimpleChannelHandler` rather than to implement the handler interfaces by yourself.
- ③ We override the `messageReceived` event handler method here. This method is called with a `MessageEvent`, which contains the received data, whenever new data is received from a client. In this example, we ignore the received data by doing nothing to implement the DISCARD protocol.
- ④ `exceptionCaught` event handler method is called with an `ExceptionEvent` when an exception was raised by Netty due to I/O error or by a handler implementation due to the exception thrown while processing events. In most cases, the caught exception should be logged and its associated channel should be closed here, although the implementation of this method can be different depending on what you want to do to deal with an exceptional situation. For example, you might want to send a response message with an error code before closing the connection.

So far so good. We have implemented the first half of the DISCARD server. What's left now is to write the main method which starts the server with the `DiscardServerHandler`.

```
package org.jboss.netty.example.discard;

import java.net.InetSocketAddress;
import java.util.concurrent.Executors;

public class DiscardServer {

    public static void main(String[] args) throws Exception {
        ChannelFactory factory =
            new NioServerSocketChannelFactory①(
                Executors.newCachedThreadPool(),
                Executors.newCachedThreadPool());

        ServerBootstrap bootstrap = new ServerBootstrap②(factory);

        DiscardServerHandler handler = new DiscardServerHandler();
        ChannelPipeline pipeline = bootstrap.getPipeline();
        pipeline.addLast("handler", handler);③

        bootstrap.setOption("child.tcpNoDelay", true);④
        bootstrap.setOption("child.keepAlive", true);

        bootstrap.bind(new InetSocketAddress(8080));⑤
    }
}
```

- 1 `ChannelFactory` is a factory which creates and manages `Channels` and its related resources. It processes all I/O requests and performs I/O to generate `ChannelEvents`. Netty provides various `ChannelFactory` implementations. We are implementing a server-side application in this example, and therefore `NioServerSocketChannelFactory` was used. Another thing to note is that it does not create I/O threads by itself. It is supposed to acquire threads from the thread pool you specified in the constructor, and it gives you more control over how threads should be managed in the environment where your application runs, such as an application server with a security manager.
- 2 `ServerBootstrap` is a helper class that sets up a server. You can set up the server using a `Channel` directly. However, please note that this is a tedious process and you do not need to do that in most cases.
- 3 Here, we add the `DiscardServerHandler` to the *default* `ChannelPipeline`. Whenever a new connection is accepted by the server, a new `ChannelPipeline` will be created for a newly accepted `Channel` and all the `ChannelHandlers` added here will be added to the new `ChannelPipeline`. It's just like a shallow-copy operation; all `Channel` and their `ChannelPipelines` will share the same `DiscardServerHandler` instance.
- 4 You can also set the parameters which are specific to the `Channel` implementation. We are writing a TCP/IP server, so we are allowed to set the socket options such as `tcpNoDelay` and `keepAlive`. Please note that the `"child."` prefix was added to all options. It means the options will be applied to the accepted `Channels` instead of the options of the `ServerSocketChannel`. You could do the following to set the options of the `ServerSocketChannel`:

```
bootstrap.setOption("reuseAddress", true);
```

- 5 We are ready to go now. What's left is to bind to the port and to start the server. Here, we bind to the port 8080 of all NICs (network interface cards) in the machine. You can now call the `bind` method as many times as you want (with different bind addresses.)

Congratulations! You've just finished your first server on top of Netty.

1.3. Looking into the Received Data

Now that we have written our first server, we need to test if it really works. The easiest way to test it is to use the `telnet` command. For example, you could enter `"telnet localhost 8080"` in the command line and type something.

However, can we say that the server is working fine? We cannot really know that because it is a discard server. You will not get any response at all. To prove it is really working, let us modify the server to print what it has received.

We already know that `MessageEvent` is generated whenever data is received and the `messageReceived` handler method will be invoked. Let us put some code into the `messageReceived` method of the `DiscardServerHandler`:

```

@Override
public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {
    ChannelBuffer ❶ buf = (ChannelBuffer) e.getMessage();
    while(buf.readable()) {
        System.out.println((char) buf.readByte());
    }
}

```

- ❶ It is safe to assume the message type in socket transports is always `ChannelBuffer`. `ChannelBuffer` is a fundamental data structure which stores a sequence of bytes in Netty. It's similar to NIO `ByteBuffer`, but it is easier to use and more flexible. For example, Netty allows you to create a composite `ChannelBuffer` which combines multiple `ChannelBuffers` reducing the number of unnecessary memory copy.

Although it resembles to NIO `ByteBuffer` a lot, it is highly recommended to refer to the API reference. Learning how to use `ChannelBuffer` correctly is a critical step in using Netty without difficulty.

If you run the `telnet` command again, you will see the server prints what has received.

The full source code of the discard server is located in the `org.jboss.netty.example.discard` package of the distribution.

1.4. Writing an Echo Server

So far, we have been consuming data without responding at all. A server, however, is usually supposed to respond to a request. Let us learn how to write a response message to a client by implementing the `ECHO` protocol, where any received data is sent back.

The only difference from the discard server we have implemented in the previous sections is that it sends the received data back instead of printing the received data out to the console. Therefore, it is enough again to modify the `messageReceived` method:

```

@Override
public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {
    Channel ❶ ch = e.getChannel();
    ch.write(e.getMessage());
}

```

- ❶ A `ChannelEvent` object has a reference to its associated `Channel`. Here, the returned `Channel` represents the connection which received the `MessageEvent`. We can get the `Channel` and call the `write` method to write something back to the remote peer.

If you run the `telnet` command again, you will see the server sends back whatever you have sent to it.

The full source code of the echo server is located in the `org.jboss.netty.example.echo` package of the distribution.

1.5. Writing a Time Server

The protocol to implement in this section is the [TIME](#) protocol. It is different from the previous examples in that it sends a message, which contains a 32-bit integer, without receiving any requests and closes the connection once the message is sent. In this example, you will learn how to construct and send a message, and to close the connection on completion.

Because we are going to ignore any received data but to send a message as soon as a connection is established, we cannot use the `messageReceived` method this time. Instead, we should override the `channelConnected` method. The following is the implementation:

```
package org.jboss.netty.example.time;

@ChannelPipelineCoverage("all")
public class TimeServerHandler extends SimpleChannelHandler {

    @Override
    public void channelConnected(ChannelHandlerContext ctx, ChannelStateEvent e) { ❶
        Channel ch = e.getChannel();

        ChannelBuffer time = ChannelBuffers.buffer(4); ❷
        time.writeInt(System.currentTimeMillis() / 1000);

        ChannelFuture f = ch.write(time); ❸

        f.addListener(new ChannelFutureListener() { ❹
            public void operationComplete(ChannelFuture f) {
                Channel ch = future.getChannel();
                ch.close();
            }
        });
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, ExceptionEvent e) {
        e.getCause().printStackTrace();
        e.getChannel().close();
    }
}
```

- ❶ As explained, `channelConnected` method will be invoked when a connection is established. Let us write the 32-bit integer that represents the current time in seconds here.
- ❷ To send a new message, we need to allocate a new buffer which will contain the message. We are going to write a 32-bit integer, and therefore we need a `ChannelBuffer` whose capacity is 4 bytes. The `ChannelBuffers` helper class is used to allocate a new buffer. Besides the `buffer` method, `ChannelBuffers` provides a lot of useful methods related to the `ChannelBuffer`. For more information, please refer to the API reference.

On the other hand, it is a good idea to use static imports for `ChannelBuffers`:

```
import static org.jboss.netty.buffer.ChannelBuffers.*;
...
ChannelBuffer dynamicBuf = dynamicBuffer(256);
ChannelBuffer ordinaryBuf = buffer(1024);
```

- 3 As usual, we write the constructed message.

But wait, where's the `flip`? Didn't we used to call `ByteBuffer.flip()` before sending a message in NIO? `ChannelBuffer` does not have such a method because it has two pointers; one for read operations and the other for write operations. The writer index increases when you write something to a `ChannelBuffer` while the reader index does not change. The reader index and the writer index represents where the message starts and ends respectively.

In contrast, NIO buffer does not provide a clean way to figure out where the message content starts and ends without calling the `flip` method. You will be in trouble when you forget to flip the buffer because nothing or incorrect data will be sent. Such an error does not happen in Netty because we have different pointer for different operation types. You will find it makes your life much easier as you get used to it -- a life without flipping out!

Another point to note is that the `write` method returns a `ChannelFuture`. A `ChannelFuture` represents an I/O operation which has not yet occurred. It means, any requested operation might not have been performed yet because all operations are asynchronous in Netty. For example, the following code might close the connection even before a message is sent:

```
Channel ch = ...;
ch.write(message);
ch.close();
```

Therefore, you need to call the `close` method after the `ChannelFuture`, which was returned by the `write` method, notifies you when the write operation has been done. Please note that, `close` might not close the connection immediately, and it returns a `ChannelFuture`.

- 4 How do we get notified when the write request is finished then? This is as simple as adding a `ChannelFutureListener` to the returned `ChannelFuture`. Here, we created a new anonymous `ChannelFutureListener` which closes the `Channel` when the operation is done.

Alternatively, you could simplify the code using a pre-defined listener:

```
f.addListener(ChannelFutureListener.CLOSE);
```

1.6. Writing a Time Client

Unlike DISCARD and ECHO servers, we need a client for the TIME protocol because a human cannot translate a 32-bit binary data into a date on a calendar. In this section, we discuss how to make sure the server works correctly and learn how to write a client with Netty.

The biggest and only difference between a server and a client in Netty is that different [Bootstrap](#) and [ChannelFactory](#) are required. Please take a look at the following code:

```
package org.jboss.netty.example.time;

import java.net.InetSocketAddress;
import java.util.concurrent.Executors;

public class TimeClient {

    public static void main(String[] args) throws Exception {
        String host = args[0];
        int port = Integer.parseInt(args[1]);

        ChannelFactory factory =
            new NioClientSocketChannelFactory1(
                Executors.newCachedThreadPool(),
                Executors.newCachedThreadPool());

        ClientBootstrap bootstrap = new ClientBootstrap2(factory);

        TimeClientHandler handler = new TimeClientHandler();
        bootstrap.getPipeline().addLast("handler", handler);

        bootstrap.setOption("tcpNoDelay"3, true);
        bootstrap.setOption("keepAlive", true);

        bootstrap.connect4(new InetSocketAddress(host, port));
    }
}
```

- ¹ [NioClientSocketChannelFactory](#), instead of [NioServerSocketChannelFactory](#) was used to create a client-side [Channel](#).
- ² [ClientBootstrap](#) is a client-side counterpart of [ServerBootstrap](#).
- ³ Please note that there's no "child." prefix. A client-side [SocketChannel](#) does not have a parent.
- ⁴ We should call the `connect` method instead of the `bind` method.

As you can see, it is not really different from the server side startup. What about the [ChannelHandler](#) implementation? It should receive a 32-bit integer from the server, translate it into a human readable format, print the translated time, and close the connection:

```

package org.jboss.netty.example.time;

@ChannelPipelineCoverage("all")
public class TimeClientHandler extends SimpleChannelHandler {

    @Override
    public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {
        ChannelBuffer buf = (ChannelBuffer) e.getMessage();
        long currentTimeMillis = buf.readInt() * 1000L;
        System.out.println(new Date(currentTimeMillis));
        e.getChannel().close();
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, ExceptionEvent e) {
        e.getCause().printStackTrace();
        e.getChannel().close();
    }
}

```

It looks very simple and does not look any different from the server side example. However, this handler sometimes will refuse to work raising an `IndexOutOfBoundsException`. We discuss why this happens in the next section.

1.7. Dealing with Packet Fragmentation and Assembly

1.7.1. What is Packet Fragmentation and Assembly?

In a stream-based transport such as TCP/IP, packets can be fragmented and reassembled during transmission even in a LAN environment. For example, let us assume you have received three packets:

```

+-----+-----+-----+
| ABC | DEF | GHI |
+-----+-----+-----+

```

because of the packet fragmentation, a server can receive them as follows:

```

+-----+-----+-----+
| AB | CDEFG | H | I |
+-----+-----+-----+

```

Therefore, a receiving part, regardless it is server-side or client-side, should defrag the received packets into one or more meaningful *frames* that could be easily understood by the application logic. In case of the example above, the received packets should be defragmented back like the following:

```

+-----+-----+-----+
| ABC | DEF | GHI |
+-----+-----+-----+

```

```
+-----+-----+-----+
```

1.7.2. The First Solution

Now let us get back to the TIME client example. We have the same problem here. A 32-bit integer is a very small amount of data, and it is not likely to be fragmented often. However, the problem is that it *can* be fragmented, and the possibility of fragmentation will increase as the traffic increases.

The simplistic solution is to create an internal cumulative buffer and wait until all 4 bytes are received into the internal buffer. The following is the modified `TimeClientHandler` implementation that fixes the problem:

```
package org.jboss.netty.example.time;

import static org.jboss.netty.buffer.ChannelBuffers.*;

@ChannelPipelineCoverage("one") ❶
public class TimeClientHandler extends SimpleChannelHandler {

    private final ChannelBuffer buf = dynamicBuffer(); ❷

    @Override
    public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {
        ChannelBuffer m = (ChannelBuffer) e.getMessage();
        buf.writeBytes(m); ❸

        if (buf.readableBytes() >= 4) { ❹
            long currentTimeMillis = buf.readInt() * 1000L;
            System.out.println(new Date(currentTimeMillis));
            e.getChannel().close();
        }
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, ExceptionEvent e) {
        e.getCause().printStackTrace();
        e.getChannel().close();
    }
}
```

- ❶ This time, "one" was used as the value of the `ChannelPipelineCoverage` annotation. It's because the new `TimeClientHandler` has to maintain the internal buffer and therefore cannot serve multiple `Channels`. If an instance of `TimeClientHandler` is shared by multiple `Channels` (and consequently multiple `ChannelPipelines`), the content of the `buf` will be corrupted.
- ❷ A *dynamic buffer* is a `ChannelBuffer` which increases its capacity on demand. It's very useful when you don't know the length of the message.

- ③ First, all received data should be cumulated into buf.
- ④ And then, the handler must check if buf has enough data, 4 bytes in this example, and proceed to the actual business logic. Otherwise, Netty will call the `messageReceived` method again when more data arrives, and eventually all 4 bytes will be cumulated.

There's another place that needs a fix. Do you remember that we added a `TimeClientHandler` instance to the *default* `ChannelPipeline` of the `ClientBootstrap`? It means one `TimeClientHandler` is going to handle multiple `Channels` and consequently the data will be corrupted. To create a new `TimeClientHandler` instance per `Channel`, we should implement a `ChannelPipelineFactory`:

```
package org.jboss.netty.example.time;

public class TimeClientPipelineFactory implements ChannelPipelineFactory {

    public ChannelPipeline getPipeline() {
        ChannelPipeline pipeline = Channels.pipeline();
        pipeline.addLast("handler", new TimeClientHandler());
        return pipeline;
    }
}
```

Now let us replace the following lines of `TimeClient`:

```
TimeClientHandler handler = new TimeClientHandler();
bootstrap.getPipeline().addLast("handler", handler);
```

with the following:

```
bootstrap.setPipelineFactory(new TimeClientPipelineFactory());
```

It might look somewhat complicated at the first glance, and it is true that we don't need to introduce `TimeClientPipelineFactory` in this particular case because `TimeClient` creates only one connection.

However, as your application gets more and more complex, you will end up with an implementation of `ChannelPipelineFactory`, which yields much more flexibility.

1.7.3. The Second Solution

Although the first solution has resolved the problem with the `TIME` client, the modified handler does not look that clean. Imagine a more complicated protocol which is composed of multiple fields such as a variable length field. Your `ChannelHandler` implementation will become unmaintainable very quickly.

As you may have noticed, you can add more than one [ChannelHandler](#) to a [ChannelPipeline](#), and therefore, you can split one monolithic [ChannelHandler](#) into multiple modular ones to reduce the complexity of your application. For example, you could split `TimeClientHandler` into two handlers:

- `TimeDecoder` which deals with the packet fragmentation and assembly issue, and
- the initial simple version of `TimeClientHandler`.

Fortunately, Netty provides an extensible class which enables you to write the first one out of the box.

```
package org.jboss.netty.example.time;

❶
public class TimeDecoder extends FrameDecoder {

    @Override
    protected Object decode(
        ChannelHandlerContext ctx, Channel channel, ChannelBuffer buffer) ❷ {

        if (buffer.readableBytes() < 4) {
            return null; ❸
        }

        return buffer.readBytes(4); ❹
    }
}
```

- ❶ There's no [ChannelPipelineCoverage](#) annotation this time because [FrameDecoder](#) is already annotated with "one".
- ❷ [FrameDecoder](#) calls `decode` method with an internally maintained cumulative buffer whenever new data is received.
- ❸ If `null` is returned, it means there's not enough data yet. [FrameDecoder](#) will call again when there is a sufficient amount of data.
- ❹ If non-`null` is returned, it means the `decode` method has decoded a message successfully. [FrameDecoder](#) will discard the read part of its internal cumulative buffer. Please remember that you don't need to decode multiple messages. [FrameDecoder](#) will keep calling the `decode` method until it returns `null`.

If you are an adventurous person, you might want to try the [ReplayingDecoder](#) which simplifies the decoder even more. You will need to consult the API reference for more information though.

```
package org.jboss.netty.example.time;

public class TimeDecoder extends ReplayingDecoder<VoidEnum> {
```

```
@Override
protected Object decode(
    ChannelHandlerContext ctx, Channel channel, ChannelBuffer buffer, VoidEnum state) {

    return buffer.readBytes(4);
}
}
```

Additionally, Netty provides out-of-the-box decoders which enables you to implement most protocols very easily and helps you avoid from ending up with a monolithic unmaintainable handler implementation. Please refer to the following packages for more detailed examples:

- `org.jboss.netty.example.factorial` for a binary protocol, and
- `org.jboss.netty.example.telnet` for a text line-based protocol.

1.8. Speaking in POJO instead of ChannelBuffer

All the examples we have reviewed so far used a `ChannelBuffer` as a primary data structure of a protocol message. In this section, we will improve the TIME protocol client and server example to use a `POJO` instead of a `ChannelBuffer`.

The advantage of using a `POJO` in your `ChannelHandler` is obvious; your handler becomes more maintainable and reusable by separating the code which extracts information from `ChannelBuffer` out from the handler. In the TIME client and server examples, we read only one 32-bit integer and it is not a major issue to use `ChannelBuffer` directly. However, you will find it is necessary to make the separation as you implement a real world protocol.

First, let us define a new type called `UnixTime`.

```
package org.jboss.netty.example.time;

public class UnixTime {
    private final int value;

    public UnixTime(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }

    @Override
    public String toString() {
        return new Date(value * 1000L).toString();
    }
}
```

We can now revise the `TimeDecoder` to return a `UnixTime` instead of a `ChannelBuffer`.

```

@Override
protected Object decode(
    ChannelHandlerContext ctx, Channel channel, ChannelBuffer buffer) {
    if (buffer.readableBytes() < 4) {
        return null;
    }

    return new UnixTime(buffer.readInt()); ❶
}

```

- ❶ [FrameDecoder](#) and [ReplayingDecoder](#) allow you to return an object of any type. If they were restricted to return only a [ChannelBuffer](#), we would have to insert another [ChannelHandler](#) which transforms a [ChannelBuffer](#) into a [UnixTime](#).

With the updated decoder, the [TimeClientHandler](#) does not use [ChannelBuffer](#) anymore:

```

@Override
public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {
    UnixTime m = (UnixTime) e.getMessage();
    System.out.println(m);
    e.getChannel().close();
}

```

Much simpler and elegant, right? The same technique can be applied on the server side. Let us update the [TimeServerHandler](#) first this time:

```

@Override
public void channelConnected(ChannelHandlerContext ctx, ChannelStateEvent e) {
    UnixTime time = new UnixTime(System.currentTimeMillis() / 1000);
    ChannelFuture f = e.getChannel().write(time);
    f.addListener(ChannelFutureListener.CLOSE);
}

```

Now, the only missing piece is the [ChannelHandler](#) which translates a [UnixTime](#) back into a [ChannelBuffer](#). It's much simpler than writing a decoder because there's no need to deal with packet fragmentation and assembly when encoding a message.

```

package org.jboss.netty.example.time;

import static org.jboss.netty.buffer.ChannelBuffers.*;

@ChannelPipelineCoverage("all") ❶
public class TimeEncoder implements SimpleChannelHandler {

    public void writeRequested(ChannelHandlerContext ctx, MessageEvent ❷ e) {
        UnixTime time = (UnixTime) e.getMessage();
    }
}

```

```

ChannelBuffer buf = buffer(4);
buf.writeInt(time.getValue());

Channels.write(ctx, e.getChannel(), e.getFuture(), buf);3
}
}

```

- 1 The `ChannelPipelineCoverage` value of an encoder is usually "all" because an encoder is stateless in most cases.
- 2 An encoder overrides the `writeRequested` method to intercept a write request. Please note that the `MessageEvent` parameter here is the same type which was specified in `messageReceived` but they are interpreted differently. A `ChannelEvent` can be either an *upstream* or *downstream* event depending on the direction where the event flows. For instance, a `MessageEvent` can be an upstream event when called for `messageReceived` or a downstream event when called for `writeRequested`. Please refer to the API reference to learn more about the difference between a upstream event and a downstream event.
- 3 Once done with transforming a POJO into a `ChannelBuffer`, you should forward the new buffer to the previous `ChannelDownstreamHandler` in the `ChannelPipeline`. `Channels` provides various helper methods which generates and sends a `ChannelEvent`. In this example, `Channels.write(...)` method creates a new `MessageEvent` and sends it to the previous `ChannelDownstreamHandler` in the `ChannelPipeline`.

On the other hand, it is a good idea to use static imports for `Channels`:

```

import static org.jboss.netty.channel.Channels.*;
...
ChannelPipeline pipeline = pipeline();
write(ctx, e.getChannel(), e.getFuture(), buf);
fireChannelDisconnected(ctx, e.getChannel());

```

The last task left is to insert a `TimeEncoder` into the `ChannelPipeline` on the server side, and it is left as a trivial exercise.

1.9. Summary

In this chapter, we had a quick tour of Netty with a demonstration on how to write a network application on top of Netty. More questions you may have will be covered in the upcoming chapters and the revised version of this chapter. Please also note that [the Netty project community](#) is always here to help you.