MINA 2.0 User Guide

Part I - Basics

Chapter 1 - Getting Started

In this chapter, we will give you first sense of what is MINA, what is NIO, why we developped a framework on top of NIO and what you will find inside. We will also show you how to run a very simple example of a server run with MINA

NIO Overview

The NIO API was introduced in Java 1.4 and had since been used for wide number of applications. The NIO API covers IO non-blocking operations.

First of all, it's good to know that MINA is written on top of NIO 1. A new version has been designed in Java 7, NIO-2, we don't yet benefit from the added features this version is carrying.

It's also important to know that the N in NIO means New, but we will use the Non-Blocking term in many places. NIO-2 should be seen as a New New I/O...

The java.nio.* package contains following key constructs

- Buffers - Data Containers
- Chartsets - Containers translators for bytes and Unicode
- Channels - represents connections to entities capable of I/O operations
- Selectors - provide selectable, multiplexed non-blocking IO
- Regexps - provide provide some tools to manipulate regular expressions

We are mostly interested in the Channels, _ Selectors_ and Buffers parts in the MINA framework, except that we want to hide those elements to the user.

This user guide will thus focus on everything built on top of those internal components.

NIO vs BIO

It's important to understand the difference between those two APIs. BIO, or Blocking IO, relies on

plain sockets used in a blocking mode : when you read, write or do whatever operation on a socket, the called operation will blcok the caller until the operation is completed.
In some cases, it's critical to be able to call the operation, and to expect the called operation to inform the caller when the operation is done : the caller can then do something else in the mean time.
This is also where NIO offers a better way to handle IO when you have numerous connected sockets : you dn't have to create a specific thread for each connection, you can just use a few threads to do the same job.
If you want to get more information about what covers NIO, there is a lot of good articles around the web, and a few books cobering this matter.

Why MINA ?

Writing some network application is generrally seen as a burden, and a low level development. It's a area which is not frequently studied or known by developpers, either because it has ben studied in school a long time ago and everything has been forgotten, or because the complexity of this network layer is frequently hidden by higher level layers, so you never get deep into it.
Add that when it comes to asynchronous IO, an extra layer of complexity comes into play : time.
The big difference between BIO (Blocking IO) and NIO (Non-Blocking IO) is that in BIO, you send a request, and you wait until you get the response. On the server side, it means one thread wil be associated with any incoming connection, so you won't have to deal with the complexity of multiplexing the connections. In NIO, on the other hand, you have to deal with the synchronous nature of a non-blocking system, wich means that your application will be invoked when some events occur. In NIO, you don't call and wait for a result, you send a command and you are informed when the result is ready.

The need of a framework

Considering those differences, and teh fact that most of the applications are usually expecting a blocking mode when invoking the network layer, the best solution is to hide this aspect by writing a framework that mimic a blocking mode. This is what MINA does !
But MINA does more. It provides a common IO vision to an application that needs to communicate over TCP, UDP or whatever mechanism. If we consider only TCP and UDP, one is a connected protocol (TCP) when the other one is connectionless (UDP). MINA masks this difference, and make you focus on the two parts that are important for your application : the applicive code and the application protocol encoding/decoding.
MINA does not only handles TCP and UDP, it's also offering a layer on top of serial communication (RSC232), over VmpPipe or APR.
Last, not least, MINA is a network framework that has been specifically designed to work either on the client side and on teh server side. Writing a server make it critical to have a scalable system, which is tunnable to fit the server needs, in term of performance and memory usage : this is what MINA is good for, still making it easy to devlop you server.

When to use MINA ?

This is a intersting question ! MINA does not expect to be the best possible choice in any case. There are a few elements to consider when considering using MINA. Let's list them :

- Ease of use When you have no special performance requirements, MINA is probably a good choice as it allows you to dvelop a server or a client easily, without having to deal with the various parameters and use cases to handle when writing the same application on top of BIO or NIO. You can probably write your server with a few tens of lines, and there are a few pitfalls in which you are likely to fall
- A high number of connected users BIO is definitively faster that NIO. The difference is something like 30% in favor of BIO. This is true for up to a few thousands of connected users, but up to a point, the BIO approach just stop scaling : you won't e able to handle millions of connected users using one thread per user ! NIO can. Now, one other aspect is that the time spent in the MINA part of your code is probably non significant, compared to whatever your application will consumme. At some point, it's probably not worthful to spend many times more energy writing a faster network layer on your own for a gain which will be barely noticable.
- A proven system MINA is used by tens of applications all over the world. There are some Apache projects based on MINA, and they are working pretty well. This is some kind of guarantee that ypu won't have to spend hours on some cryptic errors in your own implementation of the network layer.
- Existing supportd protocols MINA comes with various implemented existing protocols : HTTP, XML, TCP, LDAP, DHCP, NTP, DNS, XMPP, SSH, FTP... At some point, MINA can be seen not only as a NIO framework, but as a network layer with some protocol implementation. One of MINA feature in the near future is to offer a collection of existing protocol you can use.

Features

MINA is a simple yet full-featured network application framework which provides:

- Unified API for various transport types:
  - TCP/IP & UDP/IP via Java NIO
  - Serial communication (RS232) via RXTX
  - In-VM pipe communication
  - You can implement your own!
- Filter interface as an extension point; similar to Servlet filters
- Low-level and high-level API:
  - Low-level: uses ByteBuffers
  - High-level: uses user-defined message objects and codecs
- Highly customizable thread model:
  - Single thread
  - One thread pool
  - More than one thread pools (i.e. SEDA)
- Out-of-the-box SSL · TLS · StartTLS support using Java 5 SSLEngine

- Overload shielding & traffic throttling
- Unit testability using mock objects
- JMX managability
- Stream-based I/O support via StreamIoHandler
- Integration with well known containers such as PicoContainer and Spring
- Smooth migration from Netty, an ancestor of Apache MINA.

First Steps

We will show you how easy it is to use MINA, running a very simple example provided with the MINA package.
The first thing you have to do is to setup your evnvironment when you want to use MINA in your application. We will describe what you need to install and how to run a MINA program. Nothing fancy, just a first taste of MINA...

Download

First, you have to download the latest MINA release from Downloads Section. Just take the latest version, unless you have very good reasons not to do so...
Generally speaking, if you are going to use Maven to build your project, you won't even have to download anything, as soon as you will depend on a repository which already contains the MINA libraries : you just tell your Maven poms that you want to use the MINA jars you need.

What's inside

After the download is complete, extract the content of tar.gz or zip file to local hard drive. The downloaded compressed file has following contents
On UNIX system, type :

```
$ tar xzpf apache-mina-2.0.7-tar.gz
```

In the apache-mina-2.0.7 directory, you will get :

```
 |
 +- dist
 +- docs
 +- lib
 +- src
 +- LICENSE.txt
 +- LICENSE.jzlib.txt
 +- LICENSE.ognl.txt
 +- LICENSE.slf4j.txt
 +- LICENSE.springframework.txt
 +- NOTICE.txt
```

Content Details

- dist - Contains jars for the MINA library code
- docs - Contains API docs and Code xrefs
- lib - Contains all needed jars for all the libraries needed for using MINA

Additional to these, the base directory has couple of license and notice files

Running your first MINA program

Well, we have downloaded the release, let's run our first MINA example, shipped with the release.
Put the following jars in the classpath

- mina-core-2.0.7.jar
- mina-example-2.0.7.jar
- slf4j-api-1.6.6.jar
- slf4j-log4j12-1.6.6.jar
- log4j-1.2.17.jar

Logging Tip * Log4J 1.2 users: slf4j-api.jar, slf4j-log4j12.jar, and Log4J 1.2.x * Log4J 1.3 users: slf4j-api.jar, slf4j-log4j13.jar, and Log4J 1.3.x * java.util.logging users: slf4j-api.jar and slf4j-jdk14.jar IMPORTANT: Please make sure you are using the right slf4j-*.jar that matches to your logging framework. For instance, slf4j-log4j12.jar and log4j-1.3.x.jar can not be used together, and will malfunction. If you don't need a logging framework you can use slf4j-nop.jar for no logging or slf4j-simple.jar for very basic logging.

On the command prompt, issue the following command :

```
$ java org.apache.mina.example.gettingstarted.timeserver.MinaTimeServer
```

This shall start the server. Now telnet and see the program in action
Issue following command to telnet

```
telnet 127.0.0.1 9123
```

Well, we have run our first MINA program. Please try other sample programs shipped with MINA as examples.

Summary

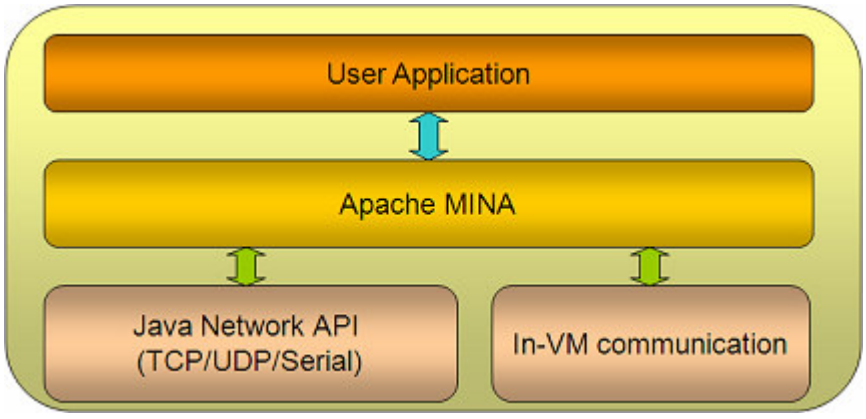
In this chapter, we looked at MINA based Application Architecture, for Client as well as Server. We also touched upon the implementation of Sample TCP Server/Client, and UDP Server and Client.
In the chapters to come we shall discuss about MINA Core constructs and advanced topics

Chapter 2 - Basics

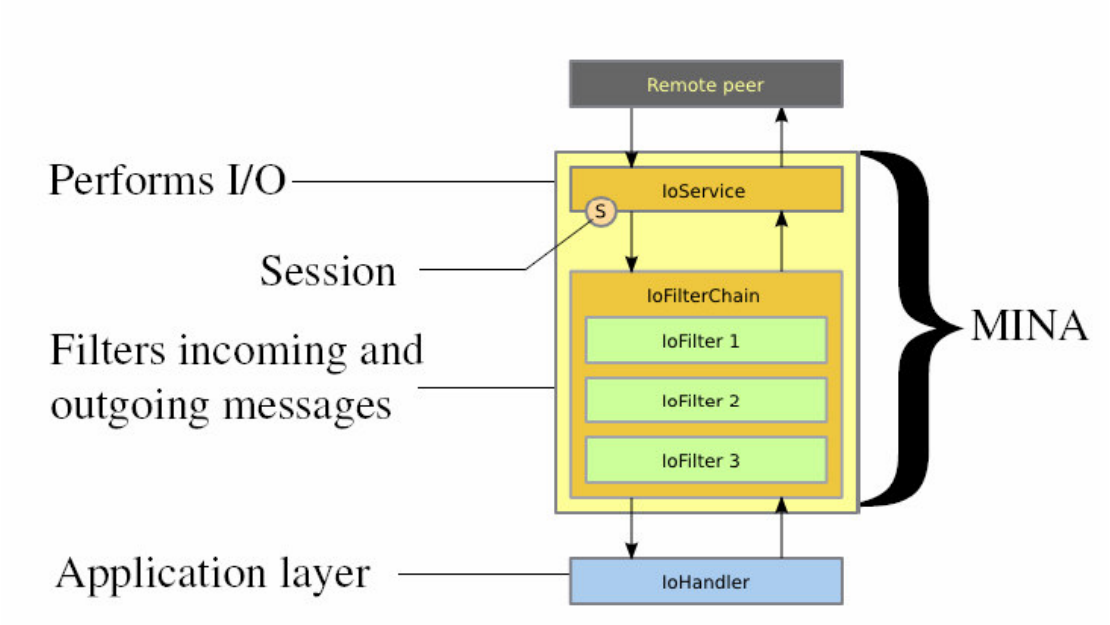
In Chapter 1, we had a brief glimpse of Apache MINA. In this chapter we shall have a look at Client/Server Architecture and details on working out a MINA based Server and Client.
We will also expose some very simple Servers and Clients, based on TCP and UDP.

MINA based Application Architecture

It's the question most asked : 'How does a MINA based application look like'? In this article lets see what's the architecture of MINA based application. Have tried to gather the information from presentations based on MINA.
A Bird's Eye View :



Here, we can see that MINA is the glue between your application (be it a client or a server) and the underlying network layer, which can be based on TCP, UDP, in-VM comunication or even a RS-232C serial protocol for a client.
You just have to design your application on top of MINA without having to handle all the complexity of the newtork layer.
Lets take a deeper dive into the details now. The following image shows a bit more the internal of MINA, and what are each of the MINA components doing :



(The image is from Emmanuel Lécharny presentation MINA in real life (ApacheCon EU 2009))
Broadly, MINA based applications are divided into 3 layers

- I/O Service - Performs actual I/O
- I/O Filter Chain - Filters/Transforms bytes into desired Data Structures and vice-versa
- I/O Handler - Here resides the actual business logic

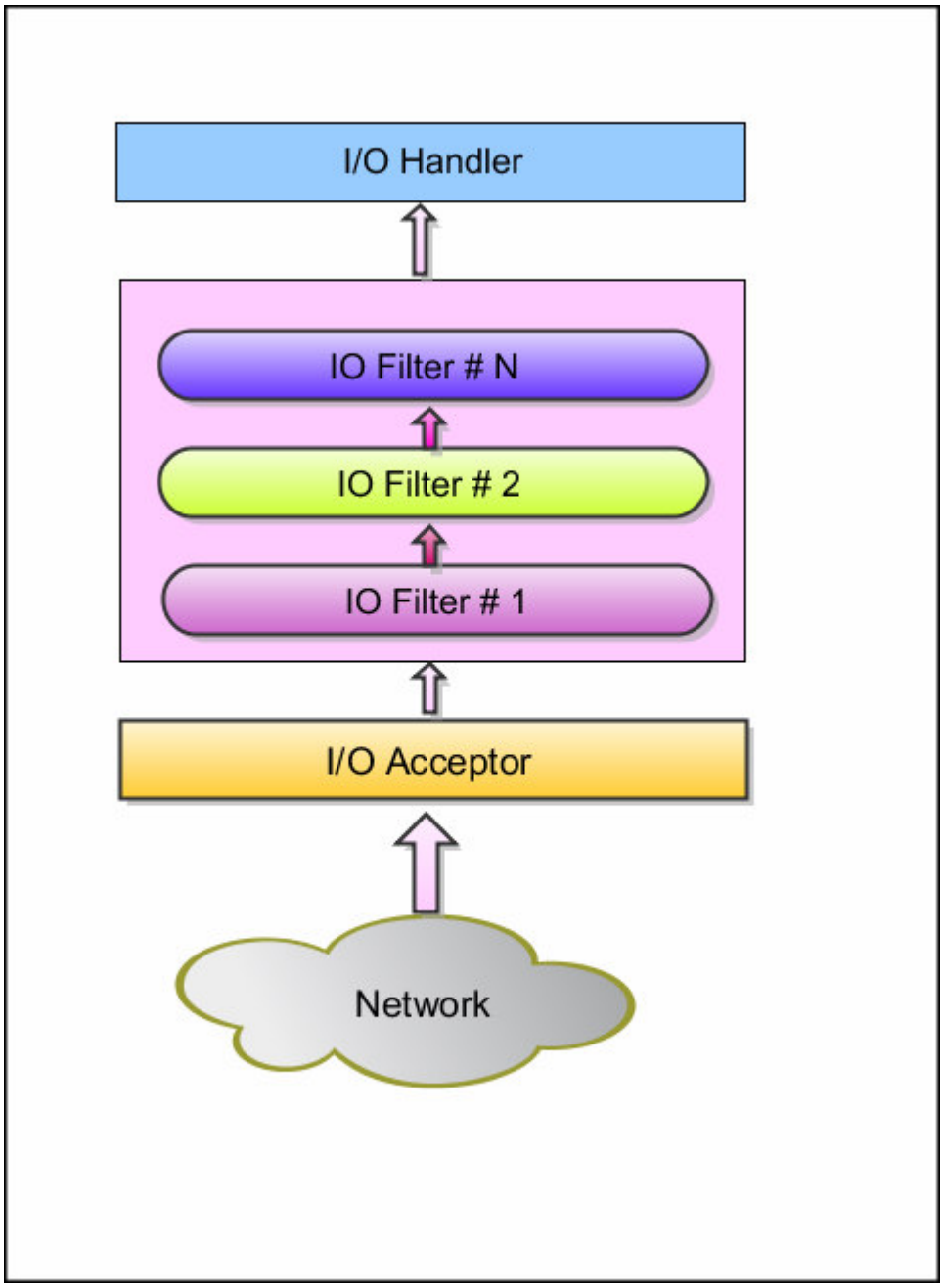
So, in order to create a MINA based Application, you have to :

1. Create an I/O service - Choose from already available Services (*Acceptor) or create your own
2. Create a Filter Chain - Choose from already existing Filters or create a custom Filter for transforming request/response
3. Create an I/O Handler - Write business logic, on handling different messages

This is pretty much it.
You can get a bit deeper by reading those two pages :
Server Architecture
Client Architecture
Of course, MINA offers more than just that, and you will robably have to take care of many oher aspects, like the messages encoding/decoding, the network configuration how to scale up, etc... We will have a further look at those aspects in the next chapters.

Server Architecture

We have exposed the MINA Application Architecture in the previous section. Let's now focus on the Server Architecture. Basically, a Server listens on a port for incoming requests, process them and send replies. It also creates and handles a session for each client (whenever we have a TCP or UDP based protocol), this will be explain more extensively in the chapter 4.



- IOAcceptor listens on the network for incoming connections/packets
- For a new connection, a new session is created and all subsequent request from IP Address/Port combination are handled in that Session
- All packets received for a Session, traverses the Filter Chain as specified in the diagram. Filters can be used to modify the content of packets (like converting to Objects, adding/removing information etc). For converting to/from raw bytes to High Level Objects, PacketEncoder/Decoder are particularly useful
- Finally the packet or converted object lands in IOHandler. IOHandlers can be used to fulfill business needs.

Session creation

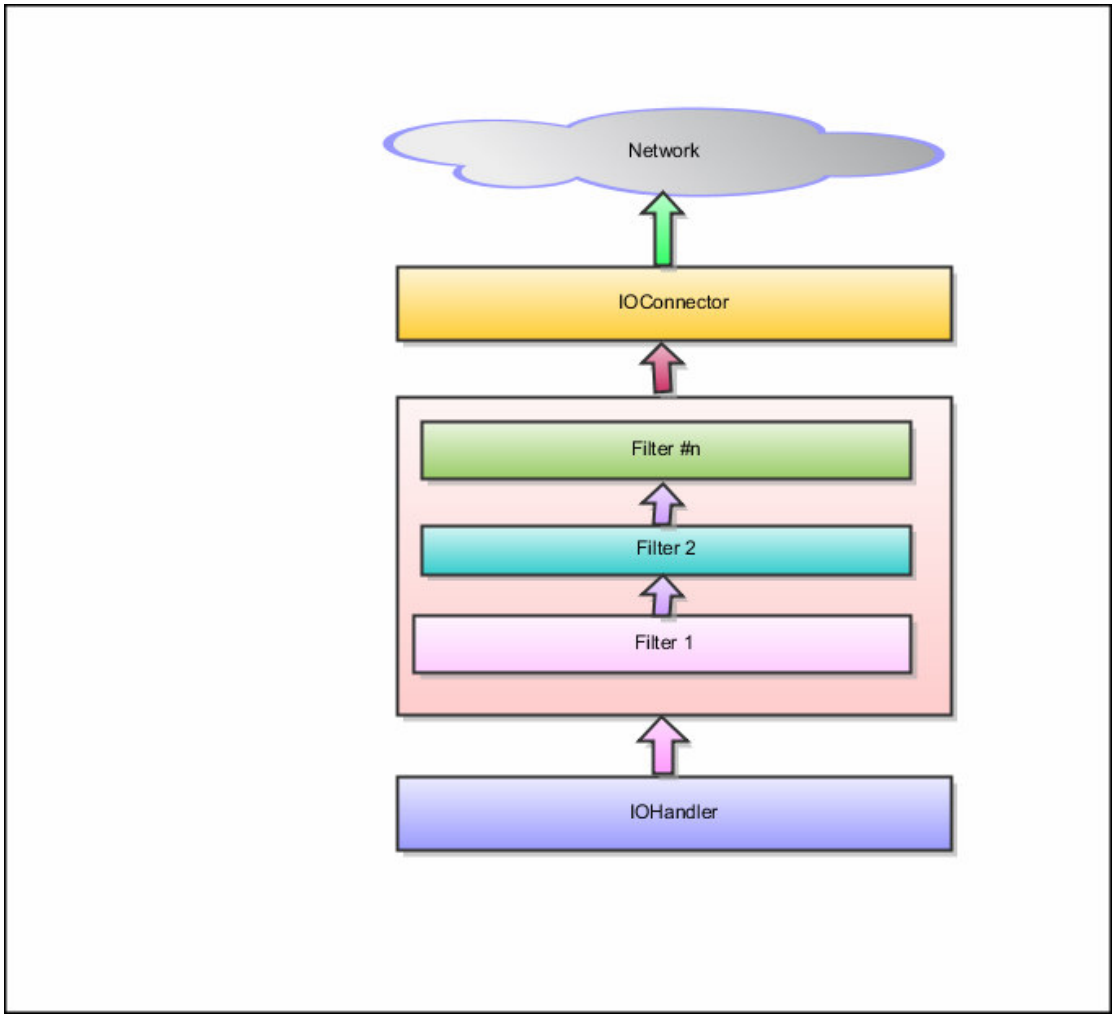
Whenever a client connects on a MINA server, we will create a new session to store persistent data into it. Even if the protocol is not connected, this session will be created. The following schema shows how MINA handles incoming connections :

Incoming connections handling

We will now explain how MINA processes incoming messages.
Assuming that a session has been created, any new incoming message will result in a selector being waken up

Client Architecture

We had a brief look at MINA based Server Architecture, lets see how Client looks like. Clients need to connect to a Server, send message and process the responses.



- Client first creates an IOConnector (MINA Construct for connecting to Socket), initiates a bind with Server
- Upon Connection creation, a Session is created and is associated with Connection

- Application/Client writes to the Session, resulting in data being sent to Server, after traversing the Filter Chain
- All the responses/messages received from Server are traverses the Filter Chain and lands at IOHandler, for processing

Sample TCP Server

This tutorial will walk you through the process of building a MINA based program. This tutorial will walk through building a time server. The following prerequisites are required for this tutorial:

- MINA 2.x Core
- JDK 1.5 or greater
- SLF4J 1.3.0 or greater

    Log4J 1.2 users: slf4j-api.jar, slf4j-log4j12.jar, and Log4J 1.2.x
    Log4J 1.3 users: slf4j-api.jar, slf4j-log4j13.jar, and Log4J 1.3.x
    java.util.logging users: slf4j-api.jar and slf4j-jdk14.jar
    IMPORTANT: Please make sure you are using the right slf4j-*.jar that matches to your logging framework.

For instance, slf4j-log4j12.jar and log4j-1.3.x.jar can not be used together, and will malfunction.
We have tested this program on both Windows? 2000 professional and linux. If you have any problems getting this program to work, please do not hesitate to contact us in order to talk to the MINA developers. Also, this tutorial has tried to remain independent of development environments (IDE, editors..etc). This tutorial will work with any environment that you are comfortable with. Compilation commands and steps to execute the program have been removed for brevity. If you need help learning how to either compile or execute java programs, please consult the Java tutorial.

Writing the MINA time server

We will begin by creating a file called MinaTimeServer.java. The initial code can be found below:

```
public class MinaTimeServer {
    public static void main(String[] args) {
        // code will go here next
    }
}
```

This code should be straightforward to all. We are simply defining a main method that will be used to kick off the program. At this point, we will begin to add the code that will make up our server. First off, we need an object that will be used to listen for incoming connections. Since this program will be TCP/IP based, we will add a SocketAcceptor to our program.

```
import org.apache.mina.transport.socket.nio.NioSocketAcceptor;

public class MinaTimeServer
{
```

```
    public static void main( String[] args )
    {
        IoAcceptor acceptor = new NioSocketAcceptor();
    }
}
```

With the NioSocketAcceptor class in place, we can go ahead and define the handler class and bind the NioSocketAcceptor to a port :

```
import java.net.InetSocketAddress;

import org.apache.mina.core.service.IoAcceptor;
import org.apache.mina.transport.socket.nio.NioSocketAcceptor;

public class MinaTimeServer
{
    private static final int PORT = 9123;
    public static void main( String[] args ) throws IOException
    {
        IoAcceptor acceptor = new NioSocketAcceptor();
        acceptor.bind( new InetSocketAddress(PORT) );
    }
}
```

As you see, there is a call to acceptor.setLocalAddress(new InetSocketAddress(PORT));. This method defines what host and port this server will listen on. The final method is a call to IoAcceptor.bind(). This method will bind to the specified port and start processing of remote clients.

Next we add a filter to the configuration. This filter will log all information such as newly created sessions, messages received, messages sent, session closed. The next filter is a ProtocolCodecFilter. This filter will translate binary or protocol specific data into message object and vice versa. We use an existing TextLine factory because it will handle text base message for you (you don't have to write the codec part)

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.charset.Charset;

import org.apache.mina.core.service.IoAcceptor;
import org.apache.mina.filter.codec.ProtocolCodecFilter;
import org.apache.mina.filter.codec.textline.TextLineCodecFactory;
import org.apache.mina.filter.logging.LoggingFilter;
import org.apache.mina.transport.socket.nio.NioSocketAcceptor;

public class MinaTimeServer
```

```
{
    public static void main( String[] args )
    {
        IoAcceptor acceptor = new NioSocketAcceptor();
        acceptor.getFilterChain().addLast( "logger", new LoggingFilter() );
        acceptor.getFilterChain().addLast( "codec", new ProtocolCodecFilter( new
TextLineCodecFactory( Charset.forName( "UTF-8" ))));
        acceptor.bind( new InetSocketAddress(PORT) );
    }
}
```

At this point, we will define the handler that will be used to service client connections and the requests for the current time. The handler class is a class that must implement the interface IoHandler. For almost all programs that use MINA, this becomes the workhorse of the program, as it services all incoming requests from the clients. For this tutorial, we will extend the class IoHandlerAdapter. This is a class that follows the adapter design pattern which simplifies the amount of code that needs to be written in order to satisfy the requirement of passing in a class that implements the IoHandler interface.

```
import java.net.InetSocketAddress;
import java.nio.charset.Charset;

import org.apache.mina.core.service.IoAcceptor;
import org.apache.mina.filter.codec.ProtocolCodecFilter;
import org.apache.mina.filter.codec.textline.TextLineCodecFactory;
import org.apache.mina.filter.logging.LoggingFilter;
import org.apache.mina.transport.socket.nio.NioSocketAcceptor;

public class MinaTimeServer
{
    public static void main( String[] args ) throws IOException
    {
        IoAcceptor acceptor = new NioSocketAcceptor();
        acceptor.getFilterChain().addLast( "logger", new LoggingFilter() );
        acceptor.getFilterChain().addLast( "codec", new ProtocolCodecFilter( new
TextLineCodecFactory( Charset.forName( "UTF-8" ))));
        acceptor.setHandler( new TimeServerHandler() );
        acceptor.bind( new InetSocketAddress(PORT) );
    }
}
```

We will now add in the NioSocketAcceptor configuration. This will allow us to make socket-specific settings for the socket that will be used to accept connections from clients.

```
import java.net.InetSocketAddress;
import java.nio.charset.Charset;
```

```
import org.apache.mina.core.session.IdleStatus;
import org.apache.mina.core.service.IoAcceptor;
import org.apache.mina.filter.codec.ProtocolCodecFilter;
import org.apache.mina.filter.codec.textline.TextLineCodecFactory;
import org.apache.mina.filter.logging.LoggingFilter;
import org.apache.mina.transport.socket.nio.NioSocketAcceptor;

public class MinaTimeServer
{
    public static void main( String[] args ) throws IOException
    {
        IoAcceptor acceptor = new NioSocketAcceptor();
        acceptor.getFilterChain().addLast( "logger", new LoggingFilter() );
        acceptor.getFilterChain().addLast( "codec", new ProtocolCodecFilter( new
TextLineCodecFactory( Charset.forName( "UTF-8" ))));
        acceptor.setHandler( new TimeServerHandler() );
        acceptor.getSessionConfig().setReadBufferSize( 2048 );
        acceptor.getSessionConfig().setIdleTime( IdleStatus.BOTH_IDLE, 10 );
        acceptor.bind( new InetSocketAddress(PORT) );
    }
}
```

There are 2 new lines in the MinaTimeServer class. These methods set the set the IoHandler, input buffer size and the idle property for the sessions. The buffer size will be specified in order to tell the underlying operating system how much room to allocate for incoming data. The second line will specify when to check for idle sessions. In the call to setIdleTime, the first parameter defines what actions to check for when determining if a session is idle, the second parameter defines the length of time in seconds that must occur before a session is deemed to be idle.

The code for the handler is shown below:

```
import java.util.Date;

import org.apache.mina.core.session.IdleStatus;
import org.apache.mina.core.service.IoHandlerAdapter;
import org.apache.mina.core.session.IoSession;

public class TimeServerHandler extends IoHandlerAdapter
{
    @Override
    public void exceptionCaught( IoSession session, Throwable cause ) throws Exception
    {
        cause.printStackTrace();
    }
```

```
    @Override
    public void messageReceived( IoSession session, Object message ) throws Exception
    {
        String str = message.toString();
        if( str.trim().equalsIgnoreCase("quit") ) {
            session.close();
            return;
        }
        Date date = new Date();
        session.write( date.toString() );
        System.out.println("Message written...");
    }
    @Override
    public void sessionIdle( IoSession session, IdleStatus status ) throws Exception
    {
        System.out.println( "IDLE " + session.getIdleCount( status ));
    }
}
```

The methods used in this class are exceptionCaught, messageReceived and sessionIdle. exceptionCaught should always be defined in a handler to process and exceptions that are raised in the normal course of handling remote connections. If this method is not defined, exceptions may not get properly reported.

The exceptionCaught method will simply print the stack trace of the error and close the session. For most programs, this will be standard practice unless the handler can recover from the exception condition.

The messageReceived method will receive the data from the client and write back to the client the current time. If the message received from the client is the word "quit", then the session will be closed. This method will also print out the current time to the client. Depending on the protocol codec that you use, the object (second parameter) that gets passed in to this method will be different, as well as the object that you pass in to the session.write(Object) method. If you do not specify a protocol codec, you will most likely receive a IoBuffer object, and be required to write out a IoBuffer object.

The sessionIdle method will be called once a session has remained idle for the amount of time specified in the call acceptor.getSessionConfig().setIdleTime(IdleStatus.BOTH_IDLE, 10);.

All that is left to do is define the socket address that the server will listen on, and actually make the call that will start the server. That code is shown below:

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.charset.Charset;
```

```
import org.apache.mina.core.service.IoAcceptor;
import org.apache.mina.core.session.IdleStatus;
import org.apache.mina.filter.codec.ProtocolCodecFilter;
import org.apache.mina.filter.codec.textline.TextLineCodecFactory;
import org.apache.mina.filter.logging.LoggingFilter;
import org.apache.mina.transport.socket.nio.NioSocketAcceptor;

public class MinaTimeServer
{
    private static final int PORT = 9123;
    public static void main( String[] args ) throws IOException
    {
        IoAcceptor acceptor = new NioSocketAcceptor();
        acceptor.getFilterChain().addLast( "logger", new LoggingFilter() );
        acceptor.getFilterChain().addLast( "codec", new ProtocolCodecFilter( new
TextLineCodecFactory( Charset.forName( "UTF-8" ))));
        acceptor.setHandler( new TimeServerHandler() );
        acceptor.getSessionConfig().setReadBufferSize( 2048 );
        acceptor.getSessionConfig().setIdleTime( IdleStatus.BOTH_IDLE, 10 );
        acceptor.bind( new InetSocketAddress(PORT) );
    }
}
```

Try out the Time server

At this point, we can go ahead and compile the program. Once you have compiled the program you can run the program in order to test out what happens. The easiest way to test the program is to start the program, and then telnet in to the program:

Client Output	Server Output
user@myhost:~> telnet 127.0.0.1 9123 Trying 127.0.0.1... Connected to 127.0.0.1. Escape character is '^]'. hello Wed Oct 17 23:23:36 EDT 2007 quit Connection closed by foreign host. user@myhost:~>	MINA Time server started. Message written...

What's Next?

Please visit our Documentation page to find out more resources. You can also keep reading other tutorials.

Sample TCP Client

We have seen the Client Architecture. Lets explore a sample Client implementation.

We shall use Sumup Client as a reference implementation.

We will remove boiler plate code and concentrate on the important constructs. Below the code for the Client :

```
public static void main(String[] args) throws Throwable {
    NioSocketConnector connector = new NioSocketConnector();
    connector.setConnectTimeoutMillis(CONNECT_TIMEOUT);

    if (USE_CUSTOM_CODEC) {
    connector.getFilterChain().addLast("codec",
        new ProtocolCodecFilter(new SumUpProtocolCodecFactory(false)));
    } else {
        connector.getFilterChain().addLast("codec",
            new ProtocolCodecFilter(new ObjectSerializationCodecFactory()));
    }

    connector.getFilterChain().addLast("logger", new LoggingFilter());
    connector.setHandler(new ClientSessionHandler(values));
    IoSession session;

    for (;;) {
        try {
            ConnectFuture future = connector.connect(new InetSocketAddress(HOSTNAME,
PORT));
            future.awaitUninterruptibly();
            session = future.getSession();
            break;
        } catch (RuntimeIoException e) {
            System.err.println(&amp;quot;Failed to connect.&amp;quot;);
            e.printStackTrace();
            Thread.sleep(5000);
        }
```

```
        }

        // wait until the summation is done
        session.getCloseFuture().awaitUninterruptibly();
        connector.dispose();
    }
```

To construct a Client, we need to do following

- Create a Connector
- Create a Filter Chain
- Create a IOHandler and add to Connector
- Bind to Server

Lets examine each one in detail

Create a Connector

```
NioSocketConnector connector = new NioSocketConnector();
```

Here we have created a NIO Socket connector

Create a Filter Chain

```
if (USE_CUSTOM_CODEC) {
        connector.getFilterChain().addLast("codec",
                new ProtocolCodecFilter(new SumUpProtocolCodecFactory(false)));
} else {
        connector.getFilterChain().addLast("codec",
                new ProtocolCodecFilter(new ObjectSerializationCodecFactory()));
}
```

We add Filters to the Filter Chain for the Connector. Here we have added a ProtocolCodec, to the filter Chain.

Create IOHandler

```
connector.setHandler(new ClientSessionHandler(values));
```

Here we create an instance of ClientSessionHandler and set it as a handler for the Connector.

Bind to Server

```
IoSession session;
```

```
for (;;) {
    try {
        ConnectFuture future = connector.connect(new InetSocketAddress(HOSTNAME,
PORT));
        future.awaitUninterruptibly();
        session = future.getSession();
        break;
    } catch (RuntimeIoException e) {
        System.err.println("Failed to connect.");
        e.printStackTrace();
        Thread.sleep(5000);
    }
}
```

Here is the most important stuff. We connect to remote Server. Since, connect is an async task, we use the ConnectFuture class to know the when the connection is complete. Once the connection is complete, we get the associated IoSession. To send any message to the Server, we shall have to write to the session. All responses/messages from server shall traverse the Filter chain and finally be handled in IoHandler.

Sample UDP Server

We will begin by looking at the code found in the org.apache.mina.example.udp package. To keep life simple, we shall concentrate on MINA related constructs only.

To construct the server, we shall have to do the following:

1. Create a Datagram Socket to listen for incoming Client requests (See MemoryMonitor.java)
2. Create an IoHandler to handle the MINA framework generated events (See MemoryMonitorHandler.java)

Here is the first snippet that addresses Point# 1:

```
NioDatagramAcceptor acceptor = new NioDatagramAcceptor();
acceptor.setHandler(new MemoryMonitorHandler(this));
```

Here, we create a NioDatagramAcceptor to listen for incoming Client requests, and set the IoHandler.The variable 'PORT' is just an int. The next step is to add a logging filter to the filter chain that this DatagramAcceptor will use. LoggingFilter is a very nice way to see MINA in Action. It generate log statements at various stages, providing an insight into how MINA works.

```
DefaultIoFilterChainBuilder chain = acceptor.getFilterChain();
chain.addLast("logger", new LoggingFilter());
```

Next we get into some more specific code for the UDP traffic. We will set the acceptor to reuse the address

```
DatagramSessionConfig dcfg = acceptor.getSessionConfig();
dcfg.setReuseAddress(true);acceptor.bind(new InetSocketAddress(PORT));
```

Of course the last thing that is required here is to call bind().

IoHandler implementation

There are three major events of interest for our Server Implementation

- Session Created
- Message Received
- Session Closed

Lets look at each of them in detail

Session Created Event

```
@Override
public void sessionCreated(IoSession session) throws Exception {
	SocketAddress remoteAddress = session.getRemoteAddress();
	server.addClient(remoteAddress);
}
```

In the session creation event, we just call addClient() function, which internally adds a Tab to the UI

Message Received Event

```
@Override
public void messageReceived(IoSession session, Object message) throws Exception {
	if (message instanceof IoBuffer) {
		IoBuffer buffer = (IoBuffer) message;
		SocketAddress remoteAddress = session.getRemoteAddress();
		server.recvUpdate(remoteAddress, buffer.getLong());
	}
 }
```

In the message received event, we just dump the data received in the message. Applications that need to send responses, can process message and write the responses onto session in this function.

Session Closed Event

```
@Override
```

```
public void sessionClosed(IoSession session) throws Exception {
    System.out.println("Session closed...");
    SocketAddress remoteAddress = session.getRemoteAddress();
    server.removeClient(remoteAddress);
}
```

In the Session Closed, event we just remove the Client tab from the UI

Sample UDP Client

Lets look at the client code for the UDP Server from previous section.

To implement the Client we need to do following:

- Create Socket and Connect to Server
- Set the IoHandler
- Collect free memory
- Send the Data to the Server

We will begin by looking at the file MemMonClient.java, found in the org.apache.mina.example.udp.client java package. The first few lines of the code are simple and straightforward.

```
connector = new NioDatagramConnector();
connector.setHandler( this );
ConnectFuture connFuture = connector.connect( new InetSocketAddress("localhost", MemoryMonitor.PORT ));
```

Here we create a NioDatagramConnector, set the handler and connect to the server. One gotcha I ran into was that you must set the host in the InetSocketAddress object or else nothing seems to work. This example was mostly written and tested on a Windows XP machine, so things may be different elsewhere. Next we will wait for acknowledgment that the client has connected to the server. Once we know we are connected, we can start writing data to the server. Here is that code:

```
connFuture.addListener( new IoFutureListener(){
                public void operationComplete(IoFuture future) {
                    ConnectFuture connFuture = (ConnectFuture)future;
                    if( connFuture.isConnected() ){
                        session = future.getSession();
                        try {
                            sendData();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
```

```
                } else {
                    log.error("Not connected...exiting");
                }
            }
        });
```

Here we add a listener to the ConnectFuture object and when we receive a callback that the client has connected, we will start to write data. The writing of data to the server will be handled by a method called sendData. This method is shown below:

```
private void sendData() throws InterruptedException {
    for (int i = 0; i < 30; i++) {
        long free = Runtime.getRuntime().freeMemory();
        IoBuffer buffer = IoBuffer.allocate(8);
        buffer.putLong(free);
        buffer.flip();
        session.write(buffer);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
            throw new InterruptedException(e.getMessage());
        }
    }
}
```

This method will write the amount of free memory to the server once a second for 30 seconds. Here you can see that we allocate a IoBuffer large enough to hold a long variable and then place the amount of free memory in the buffer. This buffer is then flipped and written to the server.

Our UDP Client implementation is complete.

Summary

In this chapter, we looked at MINA based Application Architecture, for Client as well as Server. We also touched upon the implementation of Sample TCP Server/Client, and UDP Server and Client.

In the chapters to come we shall discuss about MINA Core constructs and advanced topics

Chapter 3 - IoService

A MINA IoService - as seen in the application architecture chapter, is the base class supporting all the IO services, either from the server side or the client side.

It will handle all the interaction with your application, and with the remote peer, send and receive messages, manage sessions, conections, etc.

It's an interface, which is implemented as an IoAcceptor for the server side, and IoConnector for the client side.

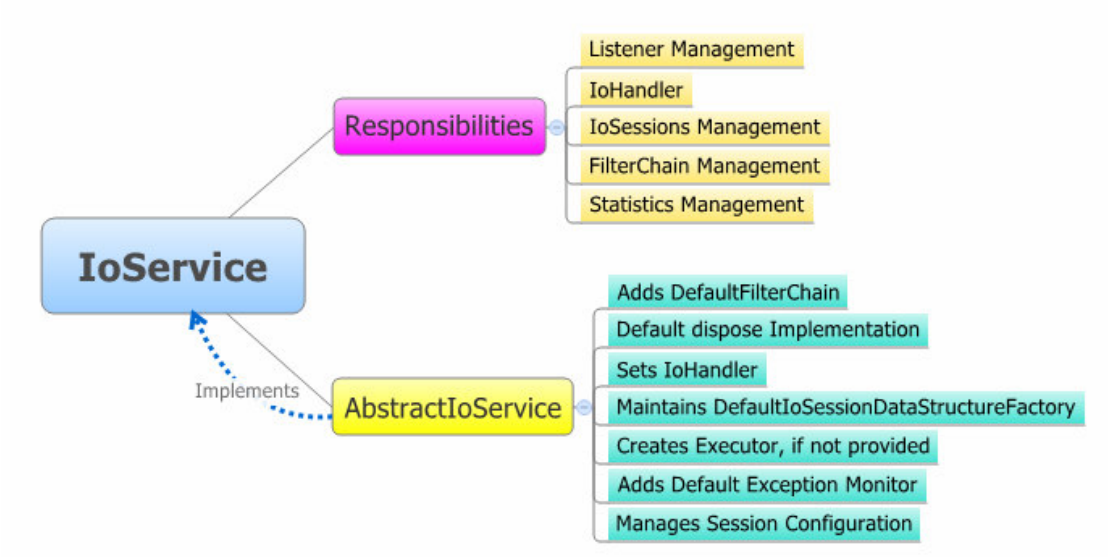
We will expose the interface in those chapters :

IoService Introduction

IoService provides basic I/O Service and manages I/O Sessions within MINA. Its one of the most crucial part of MINA Architecture. The implementing classes of IoService and child interface, are where most of the low level I/O operations are handled.

IoService Mind Map

Let's try to see what are the responsibilities of the IoService and it implementing class AbstractIoService. Let's take a slightly different approach of first using a Mind Map and then jump into the inner working. The Mind Map was created using XMind.



Responsabilities

As seen in the previous graphic, The IoService has many responsabilities :

- sessions management : Creates and deletes sessions, detect idleness.
- filter chain management : Handles the filtr chain, allowing the user to change the chain on the fly
- handler invocation : Calls the handler when some new message is received, etc
- statistics management : Updates the number of messages sent, bytes sent, and many others
- listeners management : Manages the Listeners a suer can set up
- communication management : Handles the transmission of data, in both side

All those aspects will be described in the following chapters.

Interface Details

IoService is the base interface for all the IoConnector's and IoAcceptor's that provides I/O services and manages I/O sessions. The interface has all the functions need to perform I/O related operations.

Lets take a deep dive into the various methods in the interface

- getTransportMetadata()
- addListener()
- removeListener()
- isDisposing()
- isDisposed()
- dispose()

- getHandler()
- setHandler()
- getManagedSessions()
- getManagedSessionCount()
- getSessionConfig()
- getFilterChainBuilder()
- setFilterChainBuilder()
- getFilterChain()
- isActive()
- getActivationTime()
- broadcast()
- setSessionDataStructureFactory()
- getScheduledWriteBytes()
- getScheduledWriteMessages()
- getStatistics()

getTransportMetadata()

This method returns the Transport meta-data the IoAcceptor or IoConnector is running. The typical details include provider name (nio, apr, rxtx), connection type (connectionless/connection oriented) etc.

addListener

Allows to add a IoServiceListener to listen to specific events related to IoService.

removeListener

Removes specified IoServiceListener attached to this IoService.

isDisposing

This method tells if the service is currently being disposed. As it can take a while, it's useful to know the current status of the service.

isDisposed

This method tells if the service has been disposed. A service will be considered as disposed only when all the resources it has allocated have been released.

dispose

This method releases all the resources the service has allocated. As it may take a while, the user should check the service status using the isDisposing() and isDisposed() to know if the service is now disposed completely.

Always call dispose() when you shutdown a service !

getHandler

Returns the IoHandler associated with the service.

setHandler

Sets the IoHandler that will be responsible for handling all the events for the service. The handler contains your application logic !

getManagedSessions

Returns the map of all sessions which are currently managed by this service. A managed session is a session which is added to the service listener. It will be used to process the idle sessions, and other session aspects, depending on the kind of listeners a user adds to a service.

getManagedSessionCount

Returns the number of all sessions which are currently managed by this service.

getSessionConfig

Returns the session configuration.

getFilterChainBuilder

Returns the Filter chain builder. This is useful if one wants to add some new filter that will be injected when the sessions will be created.

setFilterChainBuilder

Defines the Filter chain builder to use with the service.

getFilterChain

Returns the current default Filter chain for the service.

isActive

Tells if the service is active or not.

getActivationTime

Returns the time when this service was activated. It returns the last time when this service was activated if the service is not anymore active.

broadcast

Writes the given message to all the managed sessions.

setSessionDataStructureFactory

Sets the IoSessionDataStructureFactory that provides related data structures for a new session created by this service.

getScheduledWriteBytes

Returns the number of bytes scheduled to be written (ie, the bytes stored in memory waiting for the socket to be ready for write).

getScheduledWriteMessages

Returns the number of messages scheduled to be written (ie, the messages stored in memory waiting for the socket to be ready for write).

getStatistics

Returns the IoServiceStatistics object for this service.

IoService Details

IoService is an interface that is implemented by the two most important classes in MINA :

- IoAcceptor
- IoConnector

In order to build a server, you need to select an implementation of the IoAcceptor interface. For client applications, you need to implement an implementation of the IoConnector interface.

IoAcceptor

Basically, this interface is named because of the accept() method, responsible for the creation of new connections between a client and the server. The server accepts incoming connection requests.

At some point, we could have named this interface 'Server' (and this is the new name in the coming MINA 3.0).

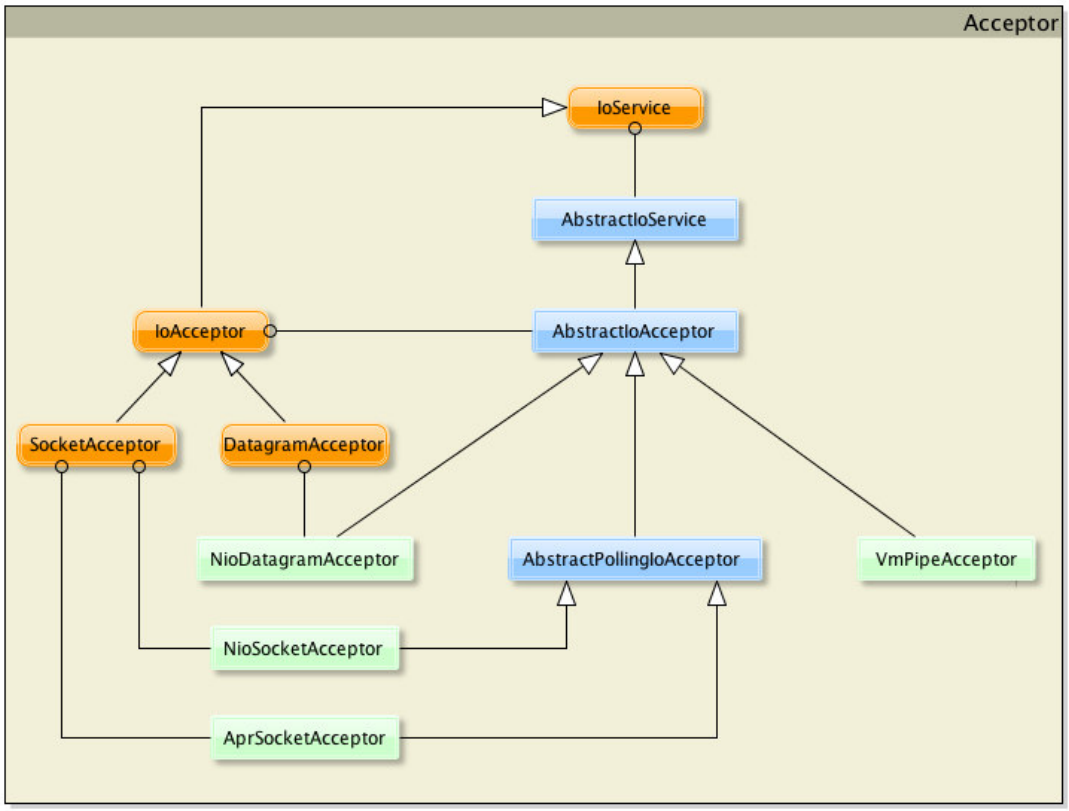
As we may deal with more than one kind of transport (TCP/UDP/...), we have more than one implementation for this interface. It would be very unlikely that you need to implement a new one.

We have many of those implementing classes

- **NioSocketAcceptor** : the non-blocking Socket transport IoAcceptor
- **NioDatagramAcceptor** : the non-blocking UDP transport IoAcceptor
- **AprSocketAcceptor** : the blocking Socket transport IoAcceptor, based on APR
- **VmPipeSocketAcceptor** : the in-VM IoAcceptor

Just pick the one that fit your need.

Here is the class diagram for the IoAcceptor interfaces and classes :



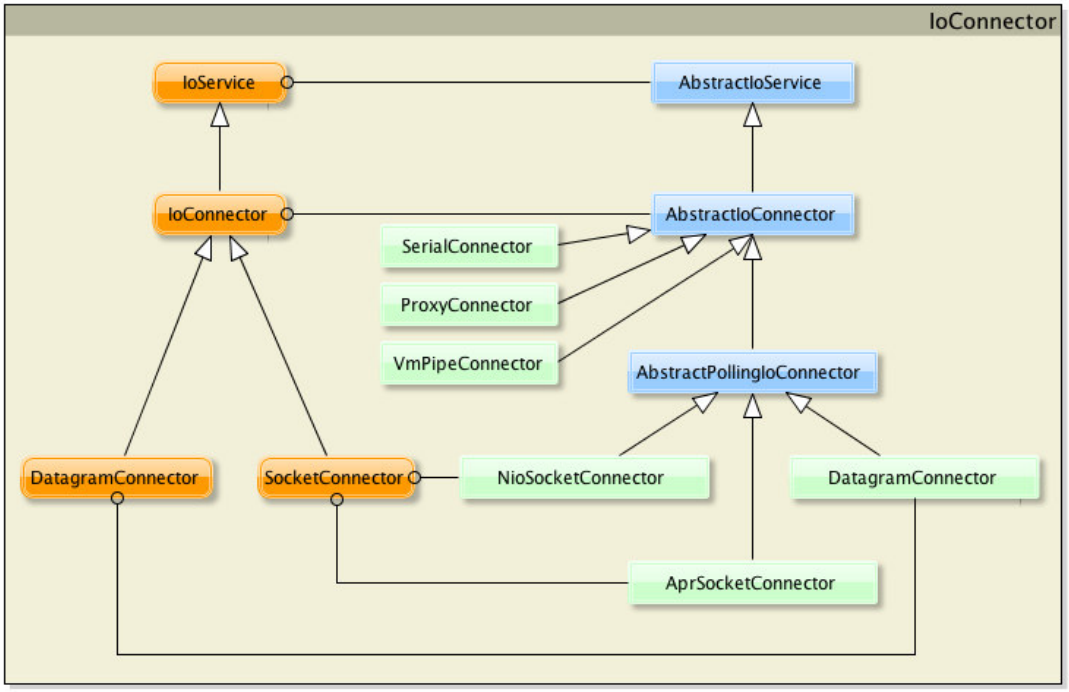
IoConnector

As we have to use an IoAcceptor for servers, you have to implement the IoConnector for clients. Again, we have many implementation classes :

- **NioSocketConnector** : the non-blocking Socket transport IoConnector
- **NioDatagramConnector** : the non-blocking UDP transport IoConnector
- **AprSocketConnector** : the blocking Socket transport IoConnector, based on APR
- **ProxyConnector** : a IoConnector providing proxy support
- **SerialConnector** : a IoConnector for a serial transport
- **VmPipeConnector** : the in-VM IoConnector

Just pick the one that fit your need.

Here is the class diagram for the IoConnector interfaces and classes :



Acceptor

In order to build a server, you need to select an implementation of the IoAcceptor interface.

IoAcceptor

Basically, this interface is named because of the accept() method, responsible for the creation of new connection between a client and the server. The server accepts incoming connection request.

At some point, we could have named this interface 'Server'.

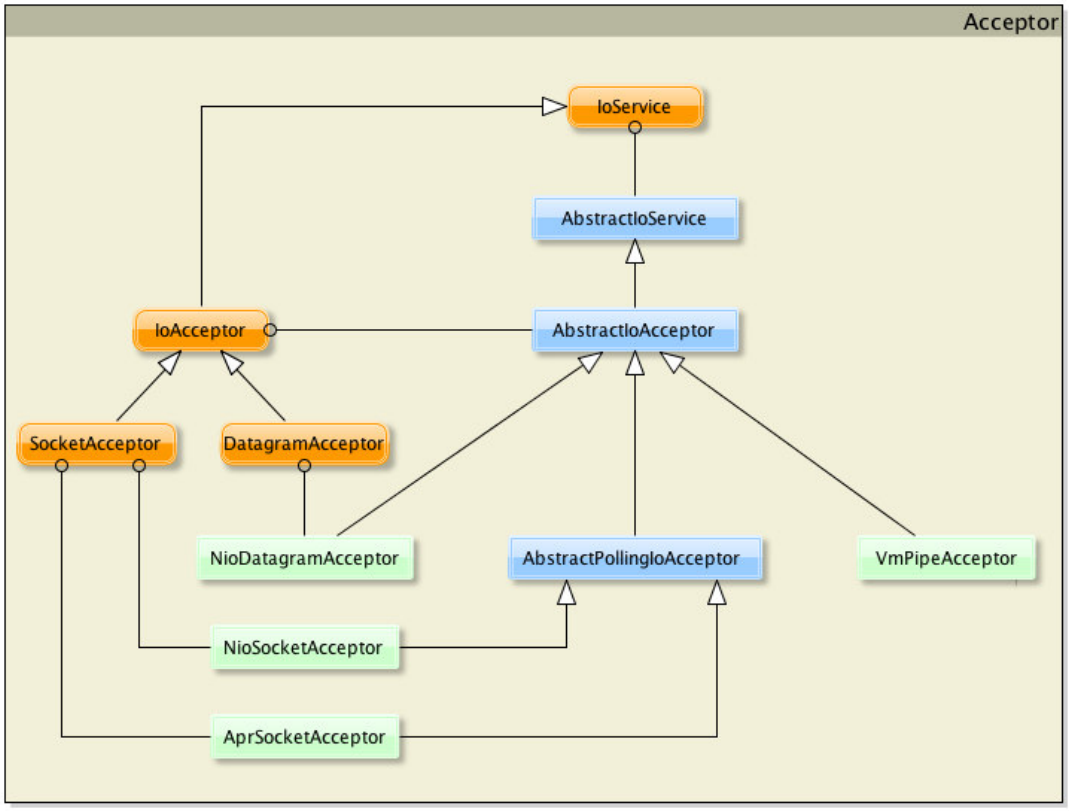
As we may deal with more than one kind of transport (TCP/UDP/...), we have more than one implementation for this interface. It would be very unlikely that you need to implement a new one.

We have many of those implementing classes

- **NioSocketAcceptor** : the non-blocking Socket transport IoAcceptor
- **NioDatagramAcceptor** : the non-blocking UDP transport IoAcceptor
- **AprSocketAcceptor** : the blocking Socket transport IoAcceptor, based on APR
- **VmPipeSocketAcceptor** : the in-VM IoAcceptor

Just pick the one that fit your need.

Here is the class diagram for the IoAcceptor interfaces and classes :



Creation

You first have to select the type of IoAcceptor you want to instanciate. This is a choice you will made early in the process, as it all boils down to which network protocol you will use. Let's see with an example how it works :

```
public TcpServer() throws IOException {
    // Create a TCP acceptor
    IoAcceptor acceptor = new NioSocketAcceptor();

    // Associate the acceptor to an IoHandler instance (your application)
    acceptor.setHandler(this);

    // Bind : this will start the server...
    acceptor.bind(new InetSocketAddress(PORT));

    System.out.println("Server started...");
}
```

That's it ! You have created a TCP server. If you want to start an UDP server, simply replace the first line of code :

```
...
// Create an UDP acceptor
IoAcceptor acceptor = new NioDatagramAcceptor();
...
```

Disposal

The service can be stopped by calling the dispose() method. The service will be stopped only when all the pending sessions have been processed :

```
// Stop the service, waiting for the pending sessions to be inactive
acceptor.dispose();
```

You can also wait for every thread being executed to be properly completed by passing a boolean parameter to this method :

```
// Stop the service, waiting for the processing session to be properly completed
acceptor.dispose( true );
```

Status

You can get the IoService status by calling one of the following methods :

- isActive() : true if the service can accept incoming requests
- isDisposing() : true if the dispose() method has been called. It does not tell if the service is actually stopped (some sessions might be processed)
- isDisposed() : true if the dispose(boolean) method has been called, and the executing threads have been completed.

Managing the IoHandler

You can add or get the associated IoHandler when the service has been instanciated. Youjust have to call the setHandler(IoHandler) or getHandler() methods.

Managing the Filters chain

if you want to manage the filters chain, you will have to call the getFilterChain() method. Here is an example :

```
// Add a logger filter
```

```
DefaultIoFilterChainBuilder chain = acceptor.getFilterChain();
chain.addLast("logger", new LoggingFilter());
```
You can also create the chain before and set it into the service :

```
// Add a logger filter
DefaultIoFilterChainBuilder chain = new DefaultIoFilterChainBuilder();
chain.addLast("logger", new LoggingFilter());

// And inject the created chain builder in the service
acceptor.setFilterChainBuilder(chain);
```

Connector

For client applications, you need to implement an implementation of the IoConnector interface.

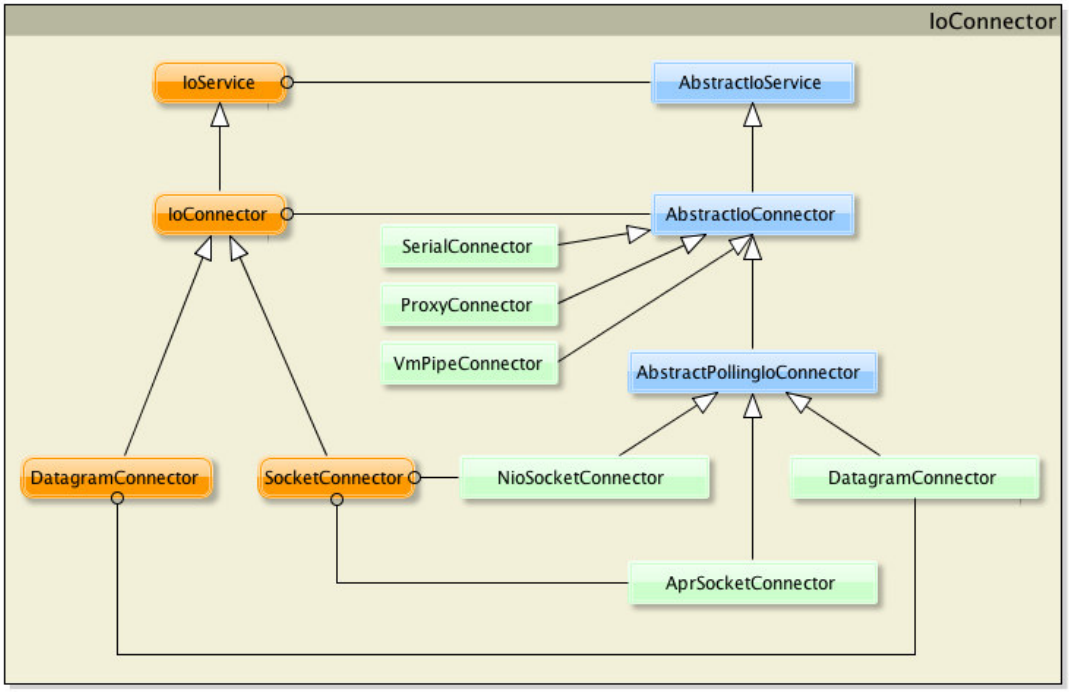
IoConnector

As we have to use an IoAcceptor for servers, you have to implement the IoConnector. Again, we have many implementation classes :

- **NioSocketConnector** : the non-blocking Socket transport Connector
- **NioDatagramConnector** : the non-blocking UDP transport * Connector*
- **AprSocketConnector** : the blocking Socket transport * Connector*, based on APR
- **ProxyConnector** : a Connector providing proxy support
- **SerialConnector** : a Connector for a serial transport
- **VmPipeConnector** : the in-VM * Connector*

Just pick the one that fit your need.

Here is the class diagram for the IoConnector interfaces and classes :


IoConnector
IoService
AbstractIoService
IoConnector
AbstractIoConnector
SerialConnector
ProxyConnector
VmPipeConnector
AbstractPollingIoConnector
DatagramConnector
SocketConnector
NioSocketConnector
DatagramConnector
AprSocketConnector

Chapter 4 - Session

The Session is at the heart of MINA : every time a client connects to the server, a new session is created, and will be kept in memory until the client is disconnected.

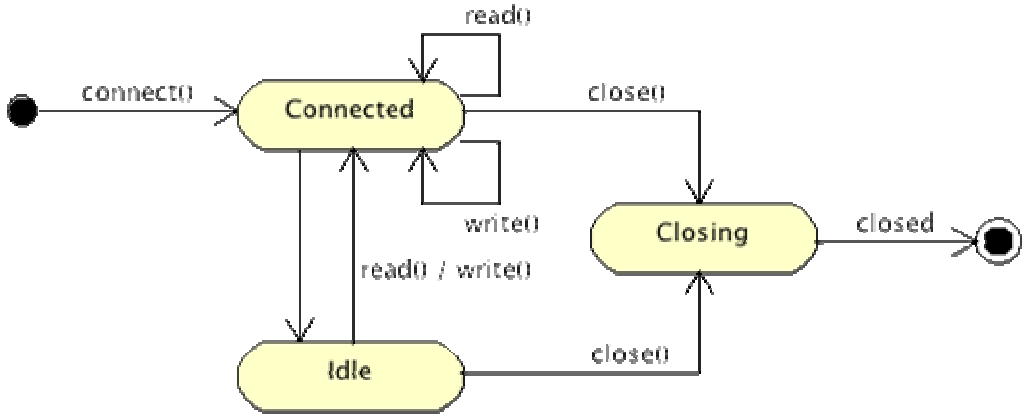
A session is used to store persistent informations about the connection, plus any kind of information the server might need to use during the request processing, and eventually during the whole session life.

Session state

A session has a state, which will evolve during time.

- Connected : the session has been created and is available
- Idle : the session hasn't processed any request for at least a period of time (this period is configurable)
  - Idle for read : no read has actually been made for a period of time
  - Idle for write : no write has actually been made for a period of time
  - Idle for both : no read nor write for a period of time
- Closing : the session is being closed (the remaining messages are being flushed, cleaning up is not terminated)
- Closed : The session is now closed, nothing else can be done to revive it.

The following state diagram exposes all the possible states and transitions :



Configuration

Many different parameters can be set for a specific session :

- receive buffer size
- sending buffer size
- Idle time
- Write timeOut

plus other configuration, depending on the transport type used (see Chapter 6 - Transports)

Managing user-defined attributes

It might be necessary to store some data which may be used later. This is done using the dedicated data structure associated which each session. This is a key-value association, which can store any type of data the developer might want to keep remanent.

For instance, if you want to track the number of request a user has sent since the session has been created, it's easy to store it into this map: just create a key that will be associated with this value.

```
...
int counterValue = session.getAttribute( "counter" );
session.setAttribute( "counter", counterValue + 1 );
...
```

We have a way to handle stored Attributes into the session : an Attribute is a key/value pair, which can be added, removed and read from the session's container.

This container is created automatically when the session is created, and will be disposed when the session is terminated.

Defining the container

As we said, this container is a key/value container, which default to a Map, but it's also possible to define another data structure if one want to handle long lived data, or to avoid storing all those data in memory if they are large : we can implement an interface and a factory that will be used to create this container when the session is created.

This snippet of code shows how the container is created during the session initialization :

```
protected final void initSession(IoSession session,
                IoFuture future, IoSessionInitializer sessionInitializer) {
```

```
        ...
        try {
                ((AbstractIoSession) session).setAttributeMap(session.getService()
                                .getSessionDataStructureFactory().getAttributeMap(session));
        } catch (IoSessionInitializationException e) {
                throw e;
        } catch (Exception e) {
                throw new IoSessionInitializationException(
                                "Failed to initialize an attributeMap.", e);
        }
        ...
```

and here is the factory interface we can implement if we want to define another kind of container :

```
public interface IoSessionDataStructureFactory {
        /**
         * Returns an {@link IoSessionAttributeMap} which is going to be associated
         * with the specified <tt>session</tt>.    Please note that the returned
         * implementation must be thread-safe.
         */
        IoSessionAttributeMap getAttributeMap(IoSession session) throws Exception;
 }
```

Filter chain

Each session is associated with a chain of filters, which will be processed when an incoming request or an outgoing message is received or emitted. Those filters are specific for each session individually, even if most of the cases, we will use the very same chain of filters for all the existing sessions.

However, it's possible to dynamically modify the chain for a single session, for instance by adding a Logger Filter in the chain for a specific session.

Statistics

Each session also keep a track of records about what has been done for the session :

- number of bytes received/sent
- number of messages received/sent
- Idle status
- throughput

and many other useful informations.

Handler

Last, not least, a session is attached to a Handler, in charge of dispatching the messages to your application. This handler will also send pack response by using the session, simply by calling the write() method :

```
...
session.write( <your message> );
...
```

Chapter 5 - Filters

IoFilter is one of the MINA core constructs that serves a very important role. It filters all I/O events and requests between IoService and IoHandler. If you have an experience with web application programming, you can safely think that it's a cousin of Servlet filter. Many out-of-the-box filters are provided to accelerate network application development pace by simplifying typical cross-cutting concerns using the out-of-the-box filters such as:

- LoggingFilter logs all events and requests.
- ProtocolCodecFilter converts an incoming ByteBuffer into message POJO and vice versa.
- CompressionFilter compresses all data.
- SSLFilter adds SSL - TLS - StartTLS support.
- and many more!

In this tutorial, we will walk through how to implement an IoFilter for a real world use case. It's easy to implement an IoFilter in general, but you might also need to know specifics of MINA internals. Any related internal properties will be explained here.

Filters already present

We have many filters already written. The following table list all the existing filters, with a short description of their usage.

Filter	class	Description
Blacklist	BlacklistFilter	Blocks connections from blacklisted remote addresses
Buffered Write	BufferedWriteFilter	Buffers outgoing requests like the BufferedOutputStream does
Compression	CompressionFilter	
ConnectionThrottle	ConnectionThrottleFilter	
ErrorGenerating	ErrorGeneratingFilter	
Executor	ExecutorFilter	
FileRegionWrite	FileRegionWriteFilter	
KeepAlive	KeepAliveFilter	
Logging	LoggingFilter	Logs event messages, like MessageReceived, MessageSent, SessionOpened, ...
MDC Injection	MdcInjectionFilter	Inject key IoSession properties into the MDC

Noop	NoopFilter	A filter that does nothing. Useful for tests.
Profiler	ProfilerTimerFilter	Profile event messages, like MessageReceived, MessageSent, SessionOpened, ...
ProtocolCodec	ProtocolCodecFilter	A filter in charge of encoding and decoding messages
Proxy	ProxyFilter	
Reference counting	ReferenceCountingFilter	Keeps track of the number of usages of this filter
RequestResponse	RequestResponseFilter	
SessionAttributeInitializing	SessionAttributeInitializingFilter	
StreamWrite	StreamWriteFilter	
SslFilter	SslFilter	
WriteRequest	WriteRequestFilter	

Overriding Events Selectively

You can extend IoAdapter instead of implementing IoFilter directly. Unless overriden, any received events will be forward to the next filter immediately:

```
public class MyFilter extends IoFilterAdapter {
    @Override
    public void sessionOpened(NextFilter nextFilter, IoSession session) throws Exception {
        // Some logic here...
        nextFilter.sessionOpened(session);
        // Some other logic here...
    }
}
```

Transforming a Write Request

If you are going to transform an incoming write request via IoSession.write(), things can get pretty tricky. For example, let's assume your filter transforms HighLevelMessage to LowLevelMessage when IoSession.write() is invoked with a HighLevelMessage object. You could insert appropriate transformation code to your filter's filterWrite() method and think that's all. However, you have to note that you also need to take care of messageSent event because an IoHandler or any filters next to yours will expect messageSent() method is called with HighLevelMessage as a parameter, because it's irrational for the caller to get notified that LowLevelMessage is sent when the caller actually wrote HighLevelMessage. Consequently, you have to implement both filterWrite() and messageSent() if your filter performs transformation.

Please also note that you still need to implement similar mechanism even if the types of the input object and the output object are identical (e.g. CompressionFilter) because the caller of IoSession.write() will expect exactly what he wrote in his or her messageSent() handler method.

Let's assume that you are implementing a filter that transforms a String into a char[]. Your filter's filterWrite() will look like the following:

```
public void filterWrite(NextFilter nextFilter, IoSession session, WriteRequest request) {
    nextFilter.filterWrite(
        session, new DefaultWriteRequest(
                ((String) request.getMessage()).toCharArray(), request.getFuture(),
request.getDestination()));
}
```

Now, we need to do the reverse in messageSent():

```
public void messageSent(NextFilter nextFilter, IoSession session, Object message) {
    nextFilter.messageSent(session, new String((char[]) message));
}
```

What about String-to-ByteBuffer transformation? We can be a little bit more efficient because we don't need to reconstruct the original message (String). However, it's somewhat more complex than the previous example:

```
public void filterWrite(NextFilter nextFilter, IoSession session, WriteRequest request) {
    String m = (String) request.getMessage();
    ByteBuffer newBuffer = new MyByteBuffer(m, ByteBuffer.wrap(m.getBytes());

    nextFilter.filterWrite(
            session, new WriteRequest(newBuffer, request.getFuture(),
request.getDestination()));
}

public void messageSent(NextFilter nextFilter, IoSession session, Object message) {
    if (message instanceof MyByteBuffer) {
        nextFilter.messageSent(session, ((MyByteBuffer) message).originalValue);
    } else {
        nextFilter.messageSent(session, message);
    }
}

private static class MyByteBuffer extends ByteBufferProxy {
    private final Object originalValue;
    private MyByteBuffer(Object originalValue, ByteBuffer encodedValue) {
        super(encodedValue);
        this.originalValue = originalValue;
```

```
    }
}
```

If you are using MINA 2.0, it will be somewhat different from 1.0 and 1.1. Please refer to CompressionFilter and RequestResponseFilter meanwhile.

Be Careful When Filtering sessionCreated Event

sessionCreated is a special event that must be executed in the I/O processor thread (see Configuring Thread Model). Never forward sessionCreated event to the other thread.

```
public void sessionCreated(NextFilter nextFilter, IoSession session) throws Exception {
    // ...
    nextFilter.sessionCreated(session);
}

// DON'T DO THIS!
public void sessionCreated(final NextFilter nextFilter, final IoSession session) throws Exception {
    Executor executor = ...;
    executor.execute(new Runnable() {
        nextFilter.sessionCreated(session);
        });
    }
```

Watch out the Empty Buffers!

MINA uses an empty buffer as an internal signal at a couple of cases. Empty buffers sometimes become a problem because it's a cause of various exceptions such as IndexOutOfBoundsException. This section explains how to avoid such a unexpected situation.

ProtocolCodecFilter uses an empty buffer (i.e. buf.hasRemaining() = 0) to mark the end of the message. If your filter is placed before the ProtocolCodecFilter, please make sure your filter forward the empty buffer to the next filter if your filter implementation can throw a unexpected exception if the buffer is empty:

```
public void messageSent(NextFilter nextFilter, IoSession session, Object message) {
    if (message instanceof ByteBuffer && !((ByteBuffer) message).hasRemaining()) {
        nextFilter.messageSent(nextFilter, session, message);
        return;
    }
    ...
}
```

```
public void filterWrite(NextFilter nextFilter, IoSession session, WriteRequest request) {
    Object message = request.getMessage();
    if (message instanceof ByteBuffer && !((ByteBuffer) message).hasRemaining()) {
        nextFilter.filterWrite(nextFilter, session, request);
        return;
    }
    ...
}
```

Do we always have to insert the if block for every filters? Fortunately, you don't have to. Here's the golden rule of handling empty buffers:

- If your filter works without any problem even if the buffer is empty, you don't need to add the if blocks at all.
- If your filter is placed after ProtocolCodecFilter, you don't need to add the if blocks at all.
- Otherwise, you need the if blocks.

If you need the if blocks, please remember you don't always need to follow the example above. You can check if the buffer is empty wherever you want as long as your filter doesn't throw a unexpected exception.

Chapter 6 - Transports

APR Transport

Introduction

APR (Apache Portable Runtime) provide superior scalability, performance, and better integration with native server technologies. APR transport is supported by MINA. In this section, we shall touch base upon how to use APR transport with MINA. We shall the Time Server example for this.

Pre-requisite

APR transport depends following components
APR library - Download/install appropriate library for the platform from http://www.apache.org/dist/tomcat/tomcat-connectors/native/
JNI wrapper (tomcat-apr-5.5.23.jar) The jar is shipped with release
Put the native library in PATH

Using APR Transport

Refer Time Server example for complete source

Lets see how NIO based Time server implementation looks like

```
IoAcceptor acceptor = new NioSocketAcceptor();

acceptor.getFilterChain().addLast( "logger", new LoggingFilter() );
acceptor.getFilterChain().addLast( "codec", new ProtocolCodecFilter( new TextLineCodecFactory( Charset.forName( "UTF-8" ))));

acceptor.setHandler( new TimeServerHandler() );

acceptor.getSessionConfig().setReadBufferSize( 2048 );
acceptor.getSessionConfig().setIdleTime( IdleStatus.BOTH_IDLE, 10 );

acceptor.bind( new InetSocketAddress(PORT) );
```

Lets see how to use APR Transport

```
IoAcceptor acceptor = new AprSocketAcceptor();

acceptor.getFilterChain().addLast( "logger", new LoggingFilter() );
acceptor.getFilterChain().addLast( "codec", new ProtocolCodecFilter( new TextLineCodecFactory( Charset.forName( "UTF-8" ))));

acceptor.setHandler(  new TimeServerHandler() );

acceptor.getSessionConfig().setReadBufferSize( 2048 );
acceptor.getSessionConfig().setIdleTime( IdleStatus.BOTH_IDLE, 10 );

acceptor.bind( new InetSocketAddress(PORT) );
```

We just change the NioSocketAcceptor to AprSocketAcceptor. That's it, now our Time Server shall use APR transport.

Rest complete process remains same.

Serial Transport

With the MINA 2.0 you are able to connect to serial port like you use to connect to a TCP/IP port with MINA.

Getting MINA 2.0

You you can download the latest built version (2.0.2).

If you prefer to build the code from the trunk, and need assistance to do so, please consult the Developer Guide.

Prerequisite

Useful Information

Before accessing serial port from a Java program you need a native library (.DLL or .so depending of your OS). MINA use the one from RXTX.org : ftp://ftp.qbang.org/pub/rxtx/rxtx-2.1-7-bins-r2.zip.

Just put the good .dll or .so in the jre/lib/i386/ path of your JDK/JRE or use the -Djava.library.path= argument for specify where you placed the native libraries

Useful Information

The mina-transport-serial jar is not included in the full distribution. You can download it from here

Connecting to a serial port

Serial communication for MINA provide only an IoConnector, due to the point-to-point nature of the communication media.

At this point you are supposed to have already read the MINA tutorial.

Now for connecting to a serial port you need a SerialConnector :

```
// create your connector
IoConnector connector = new SerialConnector()
connector.setHandler( ... here your buisness logic IoHandler ... );
```

Nothing very different of a SocketConnector.

Let's create an address for connecting to our serial port.

```
SerialAddress portAddress=new SerialAddress( "/dev/ttyS0", 38400, 8, StopBits.BITS_1, Parity.NONE, FlowControl.NONE );
```

The first parameter is your port identifier. For Windows computer, the serial ports are called "COM1", "COM2", etc... For Linux and some other Unix : "/dev/ttyS0", "/dev/ttyS1", "/dev/ttyUSB0".

The remaining parameters are depending of the device you are driving and the supposed communications characteristics.

- the baud rate
- the data bits
- the parity
- the flow control mecanism

Once it's done, connect the connector to the address :

```
ConnectFuture future = connector.connect( portAddress );
future.await();
IoSession sessin = future.getSession();
```

And voila ! Everything else is as usual, you can plug your filters and codecs. for learn more about RS232 : http://en.wikipedia.org/wiki/RS232

Chapter 7 - Handler

Handles all I/O events fired by MINA. The interface is hub of all activities done at the end of the Filter Chain.

IoHandler has following functions

- sessionCreated
- sessionOpened
- sessionClosed
- sessionIdle
- exceptionCaught
- messageReceived
- messageSent

sessionCreated Event

Session Created event is fired when a new connection is created. For TCP its the result of connection accept, and for UDP this is generated when a UDP packet is received. This function can be used to initialize session attributes, and perform one time activities for a particular connection.

This function is invoked from the I/O processor thread context, hence should be implemented in a way that it consumes minimal amount of time, as the same thread handles multiple sessions.

sessionOpened Event

Session opened event is invoked when a connection is opened. Its is always called after sessionCreated event. If a thread model is configured, this function is called in a thread other than the I/O processor thread.

sessionClosed Event

Session Closed event is closed, when a session is closed. Session cleaning activities like cash cleanup can be performed here.

sessionIdle Event

Session Idle event is fired when a session becomes idle. This function is not invoked for UDP transport.

exceptionCaught Event

This functions is called, when an Exception is thrown by user code or by MINA. The connection is closed, if its an IOException.

messageReceived Event

Message Received event is fired whenever a message is received. This is where the most of the processing of an application happens. You need to take care of all the message type you expect here.

messageSent Event

Message Sent event is fired, whenever a message aka response has been sent(calling IoSession.write()).

Part II - MINA Core

Chapter 8 - IoBuffer

A byte buffer used by MINA applications.

This is a replacement for ByteBuffer. MINA does not use NIO ByteBuffer directly for two reasons:

- It doesn't provide useful getters and putters such as fill, get/putString, and get/putAsciiInt() .
- It is difficult to write variable-length data due to its fixed capacity

This will change in MINA 3. The main reason why MINA has its own wrapper on top of nio ByteBuffer is to have extensible buffers. This was a very bad decision. Buffers are just buffers : a temporary place to store temporary data, before it is used. Many other solutions exist, like defining a wrapper which relies on a list of NIO ByteBuffers, instead of copying the existing buffer to a bigger one just because we want to extend the buffer capacity.
It might also be more comfortable to use an InputStream instead of a byte buffer all along the filters, as it does not imply anything about the nature of the stored data : it can be a byte array, strings, messages...

Last, not least, the current implementation defeat one of the target : zero-copy strategy (ie, once we have read the data from the socket, we want to avoid a copy being done later). As we use extensible byte buffers, we will most certainly copy those data if we have to manage big messages.

Assuming that the MINA ByteBuffer is just a wrapper on top of NIO ByteBuffer, this can be a real problem when using direct buffers.

IoBuffer Operations

Allocating a new Buffer

IoBuffer is an abstract class, hence can't be instantiated directly. To allocate IoBuffer, we need to use one of the two allocate() methods.

```
// Allocates a new buffer with a specific size, defining its type (direct or heap)
public static IoBuffer allocate(int capacity, boolean direct)

// Allocates a new buffer with a specific size
public static IoBuffer allocate(int capacity)
```

The allocate() method takes one or two arguments. The first form takes two arguments :

- capacity - the capacity of the buffer
- direct - type of buffer. true to get direct buffer, false to get heap buffer

The default buffer allocation is handled by SimpleBufferAllocator

Alternatively, following form can also be used

```
// Allocates heap buffer by default.
IoBuffer.setUseDirectBuffer(false);
// A new heap buffer is returned.
IoBuffer buf = IoBuffer.allocate(1024);
```

When using the second form, don't forget to set the default buffer type before, otherwise you will get Heap buffers by default.

Creating Auto Expanding Buffer

Creating auto expanding buffer is not very easy with java NIO API's, because of the fixed size of the buffers. Having a buffer, that can auto expand on needs is a big plus for networking applications. To address this, IoBuffer has introduced the autoExpand property. It automatically expands its capacity and limit value.

Lets see how to create an auto expanding buffer :

```
IoBuffer buffer = IoBuffer.allocate(8);
buffer.setAutoExpand(true);
```

```
buffer.putString("12345678", encoder);
// Add more to this buffer
buffer.put((byte)10);
```

The underlying ByteBuffer is reallocated by IoBuffer behind the scene if the encoded data is larger than 8 bytes in the example above. Its capacity will double, and its limit will increase to the last position the string is written. This behavior is very similar to the way StringBuffer class works.

This mechanism is very likely to be removed from MINA 3.0, as it's not really the best way to handle increased buffer size. It should be replaced by something like a InputStream hiding a list or an array of fixed sized ByteBuffers.

Creating Auto Shrinking Buffer

There are situations which calls for releasing additionally allocated bytes from the buffer, to preserve memory. IoBuffer provides autoShrink property to address the need. If autoShrink is turned on, IoBuffer halves the capacity of the buffer when compact() is invoked and only 1/4 or less of the current capacity is being used. To manually shrink the buffer, use shrink() method.

Lets see this in action :

```
IoBuffer buffer = IoBuffer.allocate(16);
buffer.setAutoShrink(true);
buffer.put((byte)1);
System.out.println("Initial Buffer capacity = "+buffer.capacity());
buffer.shrink();
System.out.println("Initial Buffer capacity after shrink = "+buffer.capacity());
buffer.capacity(32);
System.out.println("Buffer capacity after incrementing capacity to 32 = "+buffer.capacity());
buffer.shrink();
System.out.println("Buffer capacity after shrink= "+buffer.capacity());
```

We have initially allocated a capacity as 16, and set the autoShrink property as true.

Lets see the output of this :

```
Initial Buffer capacity = 16
Initial Buffer capacity after shrink = 16
Buffer capacity after incrementing capacity to 32 = 32
Buffer capacity after shrink= 16
```

Lets take a break and analyze the output

Initial buffer capacity is 16, as we created the buffer with this capacity. Internally this becomes the minimum capacity of the buffer

After calling shrink(), the capacity remains 16, as capacity shall never be less than minimum capacity
After incrementing capacity to 32, the capacity becomes 32
Call to shrink(), reduces the capacity to 16, thereby eliminating extra storage
Again, this mechanism should be a default one, without needing to explicitely tells the buffer that it can shrink.

Buffer Allocation

IoBufferAllocater is responsible for allocating and managing buffers. To have precise control on the buffer allocation policy, implement the IoBufferAllocater interface.

MINA ships with following implementations of IoBufferAllocater

- **SimpleBufferAllocator (default)** - Create a new buffer every time
- **CachedBufferAllocator** - caches the buffer which are likely to be reused during expansion

With the new available JVM, using cached IoBuffer is very unlikely to improve performances.
You can implement you own implementation of IoBufferAllocator and call setAllocator() on IoBuffer to use the same.

Chapter 9 - Codec Filter

This tutorial tries to explain why and how to use a ProtocolCodecFilter.

Why use a ProtocolCodecFilter?

- TCP guarantess delivery of all packets in the correct order. But there is no guarantee that one write operation on the sender-side will result in one read event on the receiving side. see http://en.wikipedia.org/wiki/IPv4#Fragmentation_and_reassembly and http://en.wikipedia.org/wiki/Nagle%27s_algorithm In MINA terminology: without a ProtocolCodecFilter one call of IoSession.write(Object message) by the sender can result in multiple messageReceived(IoSession session, Object message) events on the receiver; and multiple calls of IoSession.write(Object message) can lead to a single messageReceived event. You might not encounter this behavior when client and server are running on the same host (or an a local network) but your applications should be able to cope with this.
- Most network applications need a way to find out where the current message ends and where the next message starts.
- You could implement all this logic in your IoHandler, but adding a ProtocolCodecFilter will make your code much cleaner and easier to maintain.
- It allows you to separate your protocol logic from your business logic (IoHandler).

How ?

Your application is basically just receiving a bunch of bytes and you need to convert these bytes into messages (higher level objects).

There are three common techniques for splitting the stream of bytes into messages:

- use fixed length messages
- use a fixed length header that indicates the length of the body
- using a delimiter; for example many text-based protocols append a newline (or CR LF pair) after every message (http://www.faqs.org/rfcs/rfc977.html)

In this tutorial we will use the first and second method since they are definitely easier to implement. Afterwards we will look at using a delimiter.

Example

We will develop a (pretty useless) graphical chargen server to illustrate how to implement your own protocol codec (ProtocolEncoder, ProtocolDecoder, and ProtocolCodecFactory). The protocol is really simple. This is the layout of a request message:

4 bytes	4 bytes	4 bytes
width	height	numchars

- width: the width of the requested image (an integer in network byte-order)
- height: the height of the requested image (an integer in network byte-order)
- numchars: the number of chars to generate (an integer in network byte-order)

The server responds with two images of the requested dimensions, with the requested number of characters painted on it. This is the layout of a response message:

4 bytes	variable length body	4 bytes	variable length body
length1	image1	length2	image2

Overview of the classes we need for encoding and decoding requests and responses:

- **ImageRequest**: a simple POJO representing a request to our ImageServer.
- **ImageRequestEncoder**: encodes ImageRequest objects into protocol-specific data (used by the client)
- **ImageRequestDecoder**: decodes protocol-specific data into ImageRequest objects (used by the server)
- **ImageResponse**: a simple POJO representing a response from our ImageServer.
- **ImageResponseEncoder**: used by the server for encoding ImageResponse objects
- **ImageResponseDecoder**: used by the client for decoding ImageResponse objects
- **ImageCodecFactory**: this class creates the necesarry encoders and decoders

Here is the ImageRequest class :

```
public class ImageRequest {

    private int width;
    private int height;
    private int numberOfCharacters;

    public ImageRequest(int width, int height, int numberOfCharacters) {
        this.width = width;
        this.height = height;
        this.numberOfCharacters = numberOfCharacters;
    }
```

```
    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }

    public int getNumberOfCharacters() {
        return numberOfCharacters;
    }
}
```

Encoding is usually simpler than decoding, so let's start with the ImageRequestEncoder:

```
public class ImageRequestEncoder implements ProtocolEncoder {

    public void encode(IoSession session, Object message, ProtocolEncoderOutput out) throws
Exception {
        ImageRequest request = (ImageRequest) message;
        IoBuffer buffer = IoBuffer.allocate(12, false);
        buffer.putInt(request.getWidth());
        buffer.putInt(request.getHeight());
        buffer.putInt(request.getNumberOfCharacters());
        buffer.flip();
        out.write(buffer);
    }

    public void dispose(IoSession session) throws Exception {
        // nothing to dispose
    }
}
```

Remarks:

- MINA will call the encode function for all messages in the IoSession's write queue. Since our client will only write ImageRequest objects, we can safely cast message to ImageRequest.
- We allocate a new IoBuffer from the heap. It's best to avoid using direct buffers, since generally heap buffers perform better. see http://issues.apache.org/jira/browse/DIRMINA-289)
- You do not have to release the buffer, MINA will do it for you, see http://mina.apache.org/mina-project/apidocs/org/apache/mina/core/buffer/IoBuffer.html
- In the dispose() method you should release all resources acquired during encoding for the specified session. If there is nothing to dispose you could let your encoder inherit from ProtocolEncoderAdapter.

Now let's have a look at the decoder. The CumulativeProtocolDecoder is a great help for writing

your own decoder: it will buffer all incoming data until your decoder decides it can do something with it. In this case the message has a fixed size, so it's easiest to wait until all data is available:

```
public class ImageRequestDecoder extends CumulativeProtocolDecoder {

    protected boolean doDecode(IoSession session, IoBuffer in, ProtocolDecoderOutput out) throws Exception {
        if (in.remaining() >= 12) {
            int width = in.getInt();
            int height = in.getInt();
            int numberOfCharachters = in.getInt();
            ImageRequest request = new ImageRequest(width, height, numberOfCharachters);
            out.write(request);
            return true;
        } else {
            return false;
        }
    }
}
```

Remarks:

- everytime a complete message is decoded, you should write it to the ProtocolDecoderOutput; these messages will travel along the filter-chain and eventually arrive in your IoHandler.messageReceived method
- you are not responsible for releasing the IoBuffer
- when there is not enough data available to decode a message, just return false

The response is also a very simple POJO:

```
public class ImageResponse {

    private BufferedImage image1;

    private BufferedImage image2;

    public ImageResponse(BufferedImage image1, BufferedImage image2) {
        this.image1 = image1;
        this.image2 = image2;
    }

    public BufferedImage getImage1() {
        return image1;
    }

    public BufferedImage getImage2() {
```

```
            return image2;
        }
}
```

Encoding the response is also trivial:

```
public class ImageResponseEncoder extends ProtocolEncoderAdapter {

    public void encode(IoSession session, Object message, ProtocolEncoderOutput out) throws
Exception {
            ImageResponse imageResponse = (ImageResponse) message;
            byte[] bytes1 = getBytes(imageResponse.getImage1());
            byte[] bytes2 = getBytes(imageResponse.getImage2());
            int capacity = bytes1.length + bytes2.length + 8;
            IoBuffer buffer = IoBuffer.allocate(capacity, false);
            buffer.setAutoExpand(true);
            buffer.putInt(bytes1.length);
            buffer.put(bytes1);
            buffer.putInt(bytes2.length);
            buffer.put(bytes2);
            buffer.flip();
            out.write(buffer);
        }

    private byte[] getBytes(BufferedImage image) throws IOException {
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            ImageIO.write(image, "PNG", baos);
            return baos.toByteArray();
        }
}
```

Remarks:

- when it is impossible to calculate the length of the IoBuffer beforehand, you can use an auto-expanding buffer by calling buffer.setAutoExpand(true);

Now let's have a look at decoding the response:

```
public class ImageResponseDecoder extends CumulativeProtocolDecoder {

    private static final String DECODER_STATE_KEY =
ImageResponseDecoder.class.getName() + ".STATE";

    public static final int MAX_IMAGE_SIZE = 5 * 1024 * 1024;

    private static class DecoderState {
            BufferedImage image1;
```

```
    }

    protected boolean doDecode(IoSession session, IoBuffer in, ProtocolDecoderOutput out)
throws Exception {
        DecoderState decoderState = (DecoderState)
session.getAttribute(DECODER_STATE_KEY);
        if (decoderState == null) {
            decoderState = new DecoderState();
            session.setAttribute(DECODER_STATE_KEY, decoderState);
        }
        if (decoderState.image1 == null) {
            // try to read first image
            if (in.prefixedDataAvailable(4, MAX_IMAGE_SIZE)) {
                decoderState.image1 = readImage(in);
            } else {
                // not enough data available to read first image
                return false;
            }
        }
        if (decoderState.image1 != null) {
            // try to read second image
            if (in.prefixedDataAvailable(4, MAX_IMAGE_SIZE)) {
                BufferedImage image2 = readImage(in);
                ImageResponse imageResponse = new ImageResponse(decoderState.image1,
image2);
                out.write(imageResponse);
                decoderState.image1 = null;
                return true;
            } else {
                // not enough data available to read second image
                return false;
            }
        }
        return false;
    }

    private BufferedImage readImage(IoBuffer in) throws IOException {
        int length = in.getInt();
        byte[] bytes = new byte[length];
        in.get(bytes);
        ByteArrayInputStream bais = new ByteArrayInputStream(bytes);
        return ImageIO.read(bais);
    }
}
```

Remarks:

- We store the state of the decoding process in a session attribute. It would also be possible to store this state in the Decoder object itself but this has several disadvantages:
  every IoSession would need its own Decoder instance
  MINA ensures that there will never be more than one thread simultaneously executing the decode() function for the same IoSession, but it does not guarantee that it will always be the same thread. Suppose the first piece of data is handled by thread-1 who decides it cannot yet decode, when the next piece of data arrives, it could be handled by another thread. To avoid visibility problems, you must properly synchronize access to this decoder state (IoSession attributes are stored in a ConcurrentHashMap, so they are automatically visible to other threads).
  a discussion on the mailing list has lead to this conclusion: choosing between storing state in the IoSession or in the Decoder instance itself is more a matter of taste. To ensure that no two threads will run the decode method for the same IoSession, MINA needs to do some form of synchronization => this synchronization will also ensure you can't have the visibility problem described above. (Thanks to Adam Fisk for pointing this out) see http://www.nabble.com/Tutorial-on-ProtocolCodecFilter,-state-and-threads-t3965413.html
- IoBuffer.prefixedDataAvailable() is very convenient when your protocol uses a length-prefix; it supports a prefix of 1, 2 or 4 bytes.
- don't forget to reset the decoder state when you've decoded a response (removing the session attribute is another way to do it)

If the response would consist of a single image, we would not need to store decoder state:

```
protected boolean doDecode(IoSession session, IoBuffer in, ProtocolDecoderOutput out) throws Exception {
    if (in.prefixedDataAvailable(4)) {
        int length = in.getInt();
        byte[] bytes = new byte[length];
        in.get(bytes);
        ByteArrayInputStream bais = new ByteArrayInputStream(bytes);
        BufferedImage image = ImageIO.read(bais);
        out.write(image);
        return true;
    } else {
        return false;
    }
}
```

Now let's glue it all together:

```
public class ImageCodecFactory implements ProtocolCodecFactory {
    private ProtocolEncoder encoder;
```

```
    private ProtocolDecoder decoder;

    public ImageCodecFactory(boolean client) {
        if (client) {
            encoder = new ImageRequestEncoder();
            decoder = new ImageResponseDecoder();
        } else {
            encoder = new ImageResponseEncoder();
            decoder = new ImageRequestDecoder();
        }
    }

    public ProtocolEncoder getEncoder(IoSession ioSession) throws Exception {
        return encoder;
    }

    public ProtocolDecoder getDecoder(IoSession ioSession) throws Exception {
        return decoder;
    }
}
```

Remarks:

- for every new session, MINA will ask the ImageCodecFactory for an encoder and a decoder.
- since our encoders and decoders store no conversational state, it is safe to let all sessions share a single instance.

This is how the server would use the ProtocolCodecFactory:

```
public class ImageServer {
    public static final int PORT = 33789;

    public static void main(String[] args) throws IOException {
        ImageServerIoHandler handler = new ImageServerIoHandler();
        NioSocketAcceptor acceptor = new NioSocketAcceptor();
        acceptor.getFilterChain().addLast("protocol", new ProtocolCodecFilter(new
ImageCodecFactory(false)));
        acceptor.setLocalAddress(new InetSocketAddress(PORT));
        acceptor.setHandler(handler);
        acceptor.bind();
        System.out.println("server is listenig at port " + PORT);
    }
}
```

Usage by the client is identical:

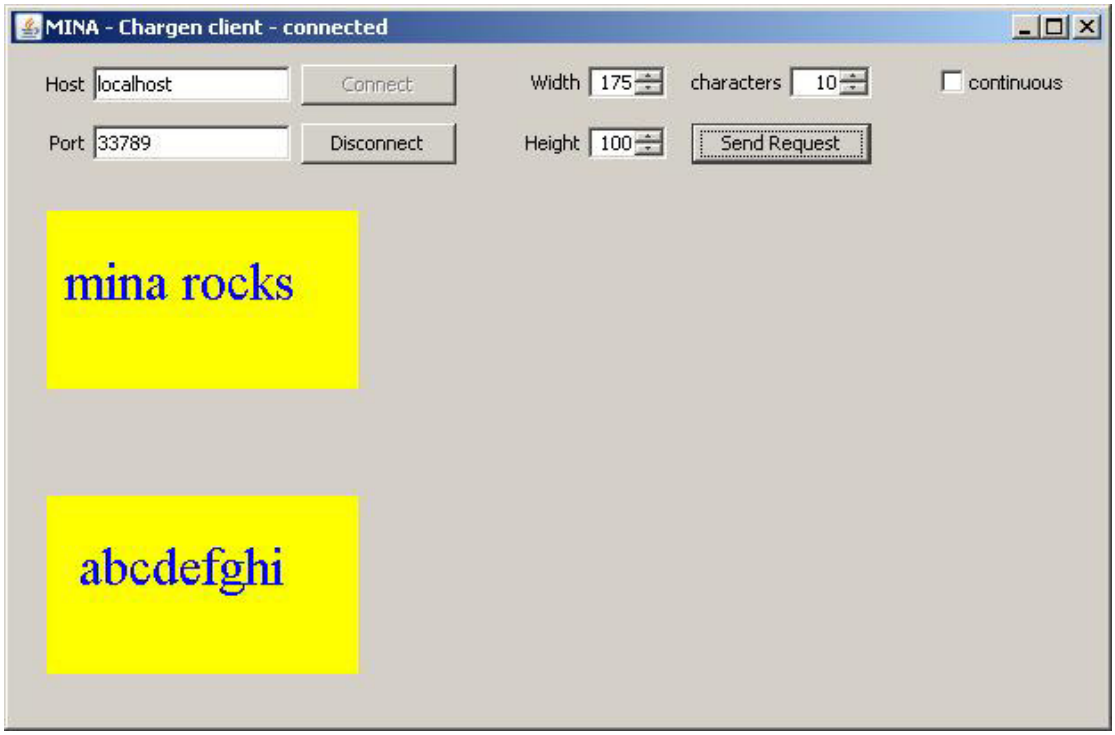
```
public class ImageClient extends IoHandlerAdapter {
```

```
    public static final int CONNECT_TIMEOUT = 3000;

    private String host;
    private int port;
    private SocketConnector connector;
    private IoSession session;
    private ImageListener imageListener;

    public ImageClient(String host, int port, ImageListener imageListener) {
        this.host = host;
        this.port = port;
        this.imageListener = imageListener;
        connector = new NioSocketConnector();
        connector.getFilterChain().addLast("codec", new ProtocolCodecFilter(new
ImageCodecFactory(true)));
        connector.setHandler(this);
    }

    public void messageReceived(IoSession session, Object message) throws Exception {
        ImageResponse response = (ImageResponse) message;
        imageListener.onImages(response.getImage1(), response.getImage2());
    }
    ...
```

For completeness, I will add the code for the server-side IoHandler:

```
public class ImageServerIoHandler extends IoHandlerAdapter {

    private final static String characters = "mina rocks
abcdefghijklmnopqrstuvwxyz0123456789";

    public static final String INDEX_KEY = ImageServerIoHandler.class.getName() +
".INDEX";

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    public void sessionOpened(IoSession session) throws Exception {
        session.setAttribute(INDEX_KEY, 0);
    }

    public void exceptionCaught(IoSession session, Throwable cause) throws Exception {
        IoSessionLogger sessionLogger = IoSessionLogger.getLogger(session, logger);
        sessionLogger.warn(cause.getMessage(), cause);
    }
```

```
    public void messageReceived(IoSession session, Object message) throws Exception {
        ImageRequest request = (ImageRequest) message;
        String text1 = generateString(session, request.getNumberOfCharacters());
        String text2 = generateString(session, request.getNumberOfCharacters());
        BufferedImage image1 = createImage(request, text1);
        BufferedImage image2 = createImage(request, text2);
        ImageResponse response = new ImageResponse(image1, image2);
        session.write(response);
    }

    private BufferedImage createImage(ImageRequest request, String text) {
        BufferedImage image = new BufferedImage(request.getWidth(), request.getHeight(),
BufferedImage.TYPE_BYTE_INDEXED);
        Graphics graphics = image.createGraphics();
        graphics.setColor(Color.YELLOW);
        graphics.fillRect(0, 0, image.getWidth(), image.getHeight());
        Font serif = new Font("serif", Font.PLAIN, 30);
        graphics.setFont(serif);
        graphics.setColor(Color.BLUE);
        graphics.drawString(text, 10, 50);
        return image;
    }

    private String generateString(IoSession session, int length) {
        Integer index = (Integer) session.getAttribute(INDEX_KEY);
        StringBuffer buffer = new StringBuffer(length);

        while (buffer.length() < length) {
            buffer.append(characters.charAt(index));
            index++;
            if (index >= characters.length()) {
                index = 0;
            }
        }
        session.setAttribute(INDEX_KEY, index);
        return buffer.toString();
    }
}
```



Conclusion

There is a lot more to tell about encoding and decoding. But I hope this tutorial already gets you started. I will try to add something about the DemuxingProtocolCodecFactory in the near future. And then we will also have a look at how to use a delimiter instead of a length prefix.

Chapter 10 - Executor Filter

MINA 1.X version let the user define the Thread Model at the Acceptor level. It was part of the Acceptor configuration. This led to complexity, and the MINA team decided to remove this option, replacing it with a much more versatile system, based on a filter : the ExecutorFilter.

The ExecutorFilter class

This class is implementing the IoFilter interface, and basically, it contains an Executor to spread the incoming events to a pool of threads. This will allow an application to use more efficiently the processors, if some task is CPU intensive.

This Filter can be used just before the handlers, assuming that most of the processing will be done in your application, or somewhere before some CPU intensive filter (for instance, a CodecFilter).

More to come ...

Chapter 11 - SSL Filter

To be completed...

Chapter 12 - Logging Filter

Background

The Apache MINA uses a system that allows for the developer of the MINA-base application to use their own logging system.

SLF4J

MINA employs the Simple Logging Facade for Java (SLF4J). You can find information on SLF4J here. This logging utility allows for the implementation of any number of logging systems. You may use log4j, java.util.logging or other logging systems. The nice part about this is that if you want to change from java.util.logging to log4j later on in the development process, you do not need to change your source code at all.

Choosing the Right JARs

SLF4J uses a static binding. This means there is one JAR file for each supported logging framework. You can use your favorite logging framework by choosing the JAR file that calls the logging framework you chose statically. The following is the table of required JAR files to use a certain logging framework.

Logging framework	Required JARs
Log4J 1.2.x	slf4j-api.jar, slf4j-log4j12.jar**
Log4J 1.3.x	slf4j-api.jar, slf4j-log4j13.jar
java.util.logging	slf4j-api.jar, slf4j-jdk14.jar**
Commons Logging	slf4j-api.jar, slf4j-jcl.jar

There are a few things to keep in mind:

- slf4j-api.jar is used commonly across any implementation JARs.
- **IMPORTANT** You should not put more than one implementation JAR files in the class path (e.g. slf4j-log4j12.jar and slf4j-jdk14.jar); it might lead your application to a unexpected behavior.
- The version of slf4j-api.jar and slf4j-.jar should be identical.

Once configured properly, you can continue to configure the actual logging framework you chose (e.g. modifying log4j.properties).

Overriding Jakarta Commons Logging

SLF4J also provides a way to convert the existing applications that use Jakarta Commons Logging to use SLF4J without changing the application code. Just remove commons-loggong JAR file from the class path, and add jcl104-over-slf4j.jar to the class path.

log4j example

For this example we will use the log4j logging system. We set up a project and place the following snippet into a file called log4j.properties:

```
# Set root logger level to DEBUG and its only appender to A1.
log4j.rootLogger=DEBUG, A1

# A1 is set to be a ConsoleAppender.
log4j.appender.A1=org.apache.log4j.ConsoleAppender

# A1 uses PatternLayout.
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %c{1} %x - %m%n
```

This file will be placed in the src directory of our project. If you are using an IDE, you essentially want the configuration file to be in the classpath for the JVM when you are testing your code.

Although this shows you how to set up an IoAcceptor to use logging, understand that the SLF4J API may be used anywhere in your program in order to generate proper logging information suitable to your needs.

Next we will set up a simple example server in order to generate some logs. Here we have taken the EchoServer example project and added logging to the class:

```
public static void main(String[] args) throws Exception {
    IoAcceptor acceptor = new SocketAcceptor();
    DefaultIoFilterChainBuilder chain = acceptor.getFilterChain();

LoggingFilter loggingFilter = new LoggingFilter();
    chain.addLast("logging", loggingFilter);

    acceptor.setLocalAddress(new InetSocketAddress(PORT));
    acceptor.setHandler(new EchoProtocolHandler());
    acceptor.bind();

    System.out.println("Listening on port " + PORT);
```

}

As you can see we removed the addLogger method and added in the 2 lines added to the example EchoServer. With a reference to the LoggingFilter, you can set the logging level per event type in your handler that is associated with the IoAcceptor here. In order to specify the IoHandler events that trigger logging and to what levels the logging is performed, there is a method in the LoggingFilter called setLogLevel(IoEventType, LogLevel). Below are the options for this method:

IoEventType	Description
SESSION_CREATED	Called when a new session has been created
SESSION_OPENED	Called when a new session has been opened
SESSION_CLOSED	Called when a session has been closed
MESSAGE_RECEIVED	Called when data has been received
MESSAGE_SENT	Called when a message has been sent
SESSION_IDLE	Called when a session idle time has been reached
EXCEPTION_CAUGHT	Called when an exception has been thrown

Here are the descriptions of the LogLevels:

LogLevel	Description
NONE	This will result in no log event being created regardless of the configuration
TRACE	Creates a TRACE event in the logging system
DEBUG	Generates debug messages in the logging system
INFO	Generates informational messages in the logging system
WARN	Generates warning messages in the logging system
ERROR	Generates error messages in the logging system

With this information, you should be able to get a basic system up and running and be able to expand upon this simple example in order to be generating log information for your system.

Part III - MINA Advanced

Chapter 13 - Debugging

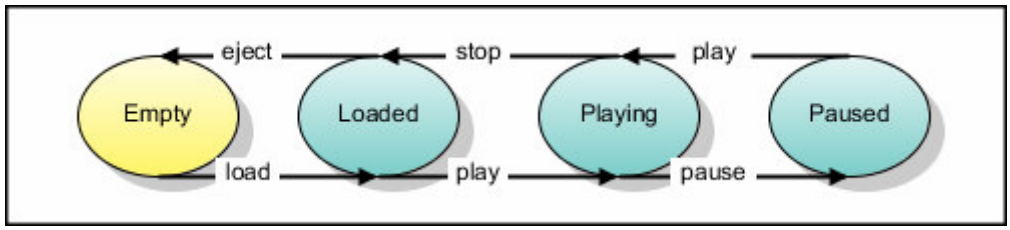
To be completed...

Chapter 14 - State Machine

If you are using MINA to develop an application with complex network interactions you may at some point find yourself reaching for the good old State pattern to try to sort out some of that complexity. However, before you do that you might want to checkout mina-statemachine which tries to address some of the shortcomings of the State pattern.

A simple example

Let's demonstrate how mina-statemachine works with a simple example. The picture below shows a state machine for a typical tape deck. The ellipsis are the states while the arrows are the transitions. Each transition is labeled with an event name which triggers that transition.



Initially, the tape deck is in the Empty state. When a tape is inserted the load event is fired and the tape deck moves to the Loaded state. In Loaded the eject event will trigger a move back to Empty while the play event will trigger a move to the Playing state. And so on... I think you can work out the rest on your own.

Now let's write some code. The outside world (the code interfacing with the tape deck) will only see the TapeDeck interface:

```
public interface TapeDeck {
    void load(String nameOfTape);
    void eject();
    void start();
```

```
    void pause();
    void stop();
}
```

Next we will write the class which contains the actual code executed when a transition occurs in the state machine. First we will define the states. The states are all defined as constant String objects and are annotated using the @State annotation:

```
public class TapeDeckHandler {
    @State public static final String EMPTY   = "Empty";
    @State public static final String LOADED  = "Loaded";
    @State public static final String PLAYING = "Playing";
    @State public static final String PAUSED  = "Paused";
}
```

Now when we have the states defined we can set up the code corresponding to each transition. Each transition will correspond to a method in TapeDeckHandler. Each transition method is annotated using the @Transition annotation which defines the event id which triggers the transition (on), the start state of the transition (in) and the end state of the transition (next):

```
public class TapeDeckHandler {
    @State public static final String EMPTY = "Empty";
    @State public static final String LOADED = "Loaded";
    @State public static final String PLAYING = "Playing";
    @State public static final String PAUSED = "Paused";

    @Transition(on = "load", in = EMPTY, next = LOADED)
    public void loadTape(String nameOfTape) {
        System.out.println("Tape '" + nameOfTape + "' loaded");
    }

    @Transitions({
        @Transition(on = "play", in = LOADED, next = PLAYING),
        @Transition(on = "play", in = PAUSED, next = PLAYING)
    })
    public void playTape() {
        System.out.println("Playing tape");
    }

    @Transition(on = "pause", in = PLAYING, next = PAUSED)
    public void pauseTape() {
        System.out.println("Tape paused");
    }

    @Transition(on = "stop", in = PLAYING, next = LOADED)
    public void stopTape() {
```

```
            System.out.println("Tape stopped");
        }

        @Transition(on = "eject", in = LOADED, next = EMPTY)
        public void ejectTape() {
            System.out.println("Tape ejected");
        }
}
```

Please note that the TapeDeckHandler class does not implement the TapeDeck interface. That's intentional.

Now, let's have a closer look at some of this code. The @Transition annotation on the loadTape method

```
@Transition(on = "load", in = EMPTY, next = LOADED)
public void loadTape(String nameOfTape) {
```

specifies that when the tape deck is in the EMPTY state and the load event occurs the loadTape method will be invoked and then the tape deck will move on to the LOADED state. The @Transition annotations on the pauseTape, stopTape and ejectTape methods should not require any further explanation. The annotation on the playTape method looks slightly different though. As can be seen in the diagram above, when the tape deck is in either the LOADED or in the PAUSED state the play event will play the tape. To have the same method called for multiple transitions the @Transitions annotation has to be used:

```
@Transitions({
        @Transition(on = "play", in = LOADED, next = PLAYING),
        @Transition(on = "play", in = PAUSED, next = PLAYING)
})
public void playTape() {
```

The @Transitions annotation simply lists multiple transitions for which the annotated method will be called.

More about the @Transition parameters

- If you omit the on parameter it will default to "*" which will match any event.
- If you omit the next parameter it will default to "_self_" which is an alias for the current state. To create a loop transition in your state machine all you have to do is to omit the next parameter.
- The weight parameter can be used to define in what order transitions will be searched. Transitions for a particular state will be ordered in ascending order according to their weight value. weight is 0 by default.

Now the final step is to create a StateMachine object from the annotated class and use it to create a proxy object which implements TapeDeck:

```
public static void main(String[] args) {
```

```
        TapeDeckHandler handler = new TapeDeckHandler();
        StateMachine sm = StateMachineFactory.getInstance(Transition.class).create(TapeDeckHandler.EMPTY, handler);
        TapeDeck deck = new StateMachineProxyBuilder().create(TapeDeck.class, sm);

        deck.load("The Knife - Silent Shout");
        deck.play();
        deck.pause();
        deck.play();
        deck.stop();
        deck.eject();
}
```

The lines

```
TapeDeckHandler handler = new TapeDeckHandler();
StateMachine sm = StateMachineFactory.getInstance(Transition.class).create(TapeDeckHandler.EMPTY, handler);
```

creates the StateMachine instance from an instance of TapeDeckHandler. The Transition.class in the call to StateMachineFactory.getInstance(...) tells the factory that we've used the @Transition annotation to build the state machine. We specify EMPTY as the start state. A StateMachine is basically a directed graph. State objects correspond to nodes in the graph while Transition objects correspond to edges. Each @Transition annotation we used in the TapeDeckHandler will correspond to a Transition instance.

Uhhm, what's the difference between @Transition and Transition?

@Transition is the annotation you use to mark a method which should be used when a transition between states occur. Behind the scenes mina-statemachine will create instances of the MethodTransition class for each @Transition annotated method. MethodTransition implements the Transition interface. As a mina-statemachine user you will never use the Transition or MethodTransition types directly.

The TapeDeck instance is created by calling StateMachineProxyBuilder:

```
TapeDeck deck = new StateMachineProxyBuilder().create(TapeDeck.class, sm);
```

The StateMachineProxyBuilder.create() method takes the interfaces the returned proxy object should implement and the StateMachine instance which will receive the events generated by the method calls on the proxy.

When the code is executed the output should be:

```
Tape 'The Knife - Silent Shout' loaded
Playing tape
Tape paused
Playing tape
Tape stopped
```

Tape ejected

What does all this have to do with MINA?

As you might have noticed there's nothing MINA specific about this example. But don't be alarmed. Later on we will see how to create state machines for MINA's IoHandler interface.

How does it work?

Let's walk through what happens when a method is called on the proxy.

Lookup a StateContext object

The StateContext object is important because it holds the current State. When a method is called on the proxy it will ask a StateContextLookup instance to get the StateContext from the method's arguments. Normally, the StateContextLookup implementation will loop through the method arguments and look for a particular type of object and use it to retrieve a StateContext object. If no StateContext has been assigned yet the StateContextLookup will create one and store it in the object.

When proxying MINA's IoHandler we will use a IoSessoinStateContextLookup instance which looks for an IoSession in the method arguments. It will use the IoSession's attributes to store a separate instance of StateContext for each MINA session. That way the same state machine can be used for all MINA sessions without them interfering with each other.

In the example above we never specified what StateContextLookup implementation to use when we created the proxy using StateMachineProxyBuilder. If not specified a SingletonStateContextLookup will be used. SingletonStateContextLookup totally disregards the method arguments passed to it – it'll always return the same StateContext object. Obviously this won't be very useful when the same state machine is used concurrently by many clients as will be the case when we proxy IoHandler later on.

Convert the method invocation into an Event object

All method invocations on the proxy object will be translated into Event objects by the proxy. An Event has an id and zero or more arguments. The id corresponds to the name of the method and the event arguments correspond to the method arguments. The method call deck.load("The Knife - Silent Shout") corresponds to the event {id = "load", arguments = ["The Knife - Silent Shout"]}. The Event object also contains a reference to the StateContext object looked up previously.

Invoke the StateMachine

Once the Event object has been created the proxy will call StateMachine.handle(Event). StateMachine.handle(Event) loops through the Transition objects of the current State in search for a Transition instance which accepts the current Event. This process will stop after a Transition has been found. The Transition objects will be searched in order of weight (typically specified by the @Transition annotation).

Execute the Transition

The final step is to call Transition.execute(Event) on the Transition which matched the Event. After the Transition has been executed the StateMachine will update the current State with the end state defined by the Transition.

Transition is an interface. Every time you use the @Transition annotation a MethodTransition object will be created.

MethodTransition

MethodTransition is very important and requires some further explanation. MethodTransition matches an Event if the event's id matches the on parameter of the @Transition annotation and the annotated method's arguments are assignment compatible with a subset of the event's arguments.

So, if the Event looks like {id = "foo", arguments = [a, b, c]} the method

```
@Transition(on = "foo")
public void someMethod(One one, Two two, Three three) { ... }
```

matches if and only if ((a instanceof One && b instanceof Two && c instanceof Three) == true). On match the method will be called with the matching event arguments bound to the method's arguments:

```
someMethod(a, b, c);
```

Integer, Double, Float, etc also match their primitive counterparts int, double, float, etc.
As stated above also a subset would match:

```
@Transition(on = "foo")
public void someMethod(Two two) { ... }
```

matches if ((a instanceof Two || b instanceof Two || c instanceof Two) == true). In this case the first matching event argument will be bound to the method argument named two when someMethod is called.

A method which takes no arguments always matches if the event id matches:

```
@Transition(on = "foo")
public void someMethod() { ... }
```

To make things even more complicated the first two method arguments also matches against the Event class and the StateContext interface. This means that

```
@Transition(on = "foo")
public void someMethod(Event event, StateContext context, One one, Two two, Three three) { ... }
@Transition(on = "foo")
public void someMethod(Event event, One one, Two two, Three three) { ... }
@Transition(on = "foo")
public void someMethod(StateContext context, One one, Two two, Three three) { ... }
```

also matches the Event {id = "foo", arguments = [a, b, c]} if ((a instanceof One && b instanceof Two && c instanceof Three) == true). The current Event object will be bound to the event method argument and the current StateContext will be bound to context when someMethod is invoked.

As before a subset of the event arguments can be used. Also, a specific StateContext implementation may be specified instead of using the generic interface:

```
@Transition(on = "foo")
public void someMethod(MyStateContext context, Two two) { ... }
```

The order of the method arguments is important. If the method needs access to the current Event it must be specified as the first method argument. StateContext has to be the either the second arguments if the first is Event or the first argument. The event arguments also have to match in the correct order. MethodTransition will not try to reorder the event's arguments in search for a match.

If you've made it this far, congratulations! I realize that the section above might be a little hard to digest. Hopefully some examples could make things clearer:

Consider the Event {id = "messageReceived", arguments = [ArrayList a = [...], Integer b = 1024]}. The following methods match this Event:

```
// All method arguments matches all event arguments directly
@Transition(on = "messageReceived")
public void messageReceived(ArrayList l, Integer i) { ... }

// Matches since ((a instanceof List && b instanceof Number) == true)
@Transition(on = "messageReceived")
public void messageReceived(List l, Number n) { ... }

// Matches since ((b instanceof Number) == true)
@Transition(on = "messageReceived")
```

```
public void messageReceived(Number n) { ... }

// Methods with no arguments always matches
@Transition(on = "messageReceived")
public void messageReceived() { ... }

// Methods only interested in the current Event or StateContext always matches
@Transition(on = "messageReceived")
public void messageReceived(StateContext context) { ... }

// Matches since ((a instanceof Collection) == true)
@Transition(on = "messageReceived")
public void messageReceived(Event event, Collection c) { ... }
```

The following would not match:

```
// Incorrect ordering
@Transition(on = "messageReceived")
public void messageReceived(Integer i, List l) { ... }

// ((a instanceof LinkedList) == false)
@Transition(on = "messageReceived")
public void messageReceived(LinkedList l, Number n) { ... }

// Event must be first argument
@Transition(on = "messageReceived")
public void messageReceived(ArrayList l, Event event) { ... }

// StateContext must be second argument if Event is used
@Transition(on = "messageReceived")
public void messageReceived(Event event, ArrayList l, StateContext context) { ... }

// Event must come before StateContext
@Transition(on = "messageReceived")
public void messageReceived(StateContext context, Event event) { ... }
```

State inheritance

State instances may have a parent State. If StateMachine.handle(Event) cannot find a Transition matching the current Event in the current State it will search the parent State. If no match is found there either the parent's parent will be searched and so on.

This feature is useful when you want to add some generic code to all states without having to specify @Transition annotations for each state. Here's how you create a hierarchy of states using

the @State annotation:

```
@State       public static final String A = "A";
@State(A) public static final String B = "A->B";
@State(A) public static final String C = "A->C";
@State(B) public static final String D = "A->B->D";
@State(C) public static final String E = "A->C->E";
```

Error handling using state inheritance

Let's go back to the TapeDeck example. What happens if you call deck.play() when there's no tape in the deck? Let's try:

```
public static void main(String[] args) {
    ...
    deck.load("The Knife - Silent Shout");
    deck.play();
    deck.pause();
    deck.play();
    deck.stop();
    deck.eject();
    deck.play();
}

...
Tape stopped
Tape ejected
Exception in thread "main" o.a.m.sm.event.UnhandledEventException:
Unhandled event: org.apache.mina.statemachine.event.Event@15eb0a9[id=play,...]
    at org.apache.mina.statemachine.StateMachine.handle(StateMachine.java:285)
    at org.apache.mina.statemachine.StateMachine.processEvents(StateMachine.java:142)
    ...
```

Oops! We get an UnhandledEventException because when we're in the Empty state there's no transition which handles the play event. We could add a special transition to all states which handles unmatched Event objects:

```
@Transitions({
    @Transition(on = "*", in = EMPTY, weight = 100),
    @Transition(on = "*", in = LOADED, weight = 100),
    @Transition(on = "*", in = PLAYING, weight = 100),
    @Transition(on = "*", in = PAUSED, weight = 100)
})
public void error(Event event) {
```

```
        System.out.println("Cannot '" + event.getId() + "' at this time");
}
```

Now when you run the main() method above you won't get an exception. The output should be:

```
...
Tape stopped
Tape ejected
Cannot 'play' at this time.
```

Now this seems to work very well, right? But what if we had 30 states instead of only 4? Then we would need 30 @Transition annotations on the error() method. Not good. Let's use state inheritance instead:

```
public static class TapeDeckHandler {
        @State public static final String ROOT = "Root";
        @State(ROOT) public static final String EMPTY = "Empty";
        @State(ROOT) public static final String LOADED = "Loaded";
        @State(ROOT) public static final String PLAYING = "Playing";
        @State(ROOT) public static final String PAUSED = "Paused";

        ...

        @Transition(on = "*", in = ROOT)
        public void error(Event event) {
                System.out.println("Cannot '" + event.getId() + "' at this time");
        }
}
```

The result will be the same but things will be much easier to maintain whith this last approach.

mina-statemachine with IoHandler

Now we're going to convert our tape deck into a TCP server and extend it with some more functionality. The server will receive commands like load , play, stop, etc. The responses will either be positive + or negative - . The protocol is text based, all commands and responses are lines of UTF-8 text terminated by CRLF (i.e. \r\n in Java). Here's an example session:

```
telnet localhost 12345
S: + Greetings from your tape deck!
C: list
S: + (1: "The Knife - Silent Shout", 2: "Kings of convenience - Riot on an empty street")
C: load 1
S: + "The Knife - Silent Shout" loaded
C: play
S: + Playing "The Knife - Silent Shout"
```

```
C: pause
S: + "The Knife - Silent Shout" paused
C: play
S: + Playing "The Knife - Silent Shout"
C: info
S: + Tape deck is playing. Current tape: "The Knife - Silent Shout"
C: eject
S: - Cannot eject while playing
C: stop
S: + "The Knife - Silent Shout" stopped
C: eject
S: + "The Knife - Silent Shout" ejected
C: quit
S: + Bye! Please come back!
```

The complete code for the TapeDeckServer described in this section is available in the org.apache.mina.example.tapedeck package in the mina-example module in the Subversion repository. The code uses a MINA ProtocolCodecFilter to convert bytes from/to Command objects. There is one Command implementation for each type of request the server recognizes. We will not describe the codec implementation here in any detail.

Now, let's have a look at how this server works. The important class which implements the state machine is the TapeDeckServer class. The first thing we do is to define the states:

```
@State public static final String ROOT = "Root";
@State(ROOT) public static final String EMPTY = "Empty";
@State(ROOT) public static final String LOADED = "Loaded";
@State(ROOT) public static final String PLAYING = "Playing";
@State(ROOT) public static final String PAUSED = "Paused";
```

Nothing new there. However, the methods which handle the events now look different. Let's look at the playTape method:

```
@IoHandlerTransitions({
        @IoHandlerTransition(on = MESSAGE_RECEIVED, in = LOADED, next = PLAYING),
        @IoHandlerTransition(on = MESSAGE_RECEIVED, in = PAUSED, next = PLAYING)
})
public void playTape(TapeDeckContext context, IoSession session, PlayCommand cmd) {
        session.write("+ Playing \"" + context.tapeName + "\"");
}
```

This code doesn't use the general @Transition and @Transitions annotations used previously but rather the MINA specific @IoHandlerTransition and @IoHandlerTransitions annotations. This are preferred when creating state machines for MINA's IoHandler interface as they let you use a Java enum for the event ids instead of strings as we used before. There are also corresponding annotations for MINA's IoFilter interface.

We're now using MESSAGE_RECEIVED instead of "play" for the event name (the on attribute in @IoHandlerTransition). This constant is defined in org.apache.mina.statemachine.event.IoHandlerEvents and has the value "messageReceived" which of course corresponds to the messageReceived() method in MINA's IoHandler interface. Thanks to Java5's static imports we don't have to write out the name of the class holding the constant. We just need to put the

```
import static org.apache.mina.statemachine.event.IoHandlerEvents.*;
```

statement in the imports section.

Another thing that has changed is that we're using a custom StateContext implementation, TapeDeckContext. This class is used to keep track of the name of the current tape:

```
static class TapeDeckContext extends AbstractStateContext {
    public String tapeName;
}
```

Why not store tape name in IoSession?

We could have stored the name of the tape as an attribute in the IoSession but using a custom StateContext is recommended since it provides type safety.

The last thing to note about the playTape() method is that it takes a PlayCommand as its last argument. The last argument corresponds to the message argument of IoHandler's messageReceived(IoSession session, Object message) method. This means that playTape() method will only be called if the bytes sent by the client can be decoded as a PlayCommand.

Before the tape deck can play anything a tape has to be loaded. When a LoadCommand is received from the client the supplied tape number will be used to get the name of the tape to load from the tapes array of available tapes:

```
@IoHandlerTransition(on = MESSAGE_RECEIVED, in = EMPTY, next = LOADED)
public void loadTape(TapeDeckContext context, IoSession session, LoadCommand cmd) {
    if (cmd.getTapeNumber() < 1 || cmd.getTapeNumber() > tapes.length) {
        session.write("- Unknown tape number: " + cmd.getTapeNumber());
        StateControl.breakAndGotoNext(EMPTY);
    } else {
        context.tapeName = tapes[cmd.getTapeNumber() - 1];
        session.write("+ \"" + context.tapeName + "\" loaded");
    }
}
```

This code uses the StateControl class to override the next state. If the user specify an unknown tape number we shouldn't move to the LOADED state but instead remain in EMPTY which is what the

```
StateControl.breakAndGotoNext(EMPTY);
```

line does. The StateControl class is described more in a later section.

The connect() method will always be called at the start of a session when MINA calls sessionOpened() on the IoHandler:

```
@IoHandlerTransition(on = SESSION_OPENED, in = EMPTY)
public void connect(IoSession session) {
    session.write("+ Greetings from your tape deck!");
}
```

All it does is to write the greeting to the client. The state machine will remain in the EMPTY state.

The pauseTape(), stopTape() and ejectTape() methods are very similar to playTape() and won't be described in any detail. The listTapes(), info() and quit() methods should be simple enough to understand by now, too. Please note how these last three methods are used for the ROOT state. This means that the list, info and quit commands can be issued in any state.

Now let's have a look at error handling. The error() method will be called when the client sends a Command which isn't legal in the current state:

```
@IoHandlerTransition(on = MESSAGE_RECEIVED, in = ROOT, weight = 10)
public void error(Event event, StateContext context, IoSession session, Command cmd) {
    session.write("- Cannot " + cmd.getName() + " while "
            + context.getCurrentState().getId().toLowerCase());
}
```

error() has been given a higher weight than listTapes(), info() and quit() to prevent it to be called for any of those commands. Notice how error() uses the StateContext object to get hold of the id of the current state. The values of the String constants which are annotated with the @State annotation (Empty, Loaded etc) will be used by mina-statemachine as state id.

The commandSyntaxError() method will be called when a CommandSyntaxException has been thrown by our ProtocolDecoder. It simply prints out that the line sent by the client couldn't be converted into a Command.

The exceptionCaught() will be called for any thrown exception except CommandSyntaxException (it has a higher weight than the commandSyntaxError() method). It closes the session immediately.

The last @IoHandlerTransition method is unhandledEvent() which will be called if none of the other @IoHandlerTransition methods match the Event. We need this since we don't have @IoHandlerTransition annotations for all possible types of events in all states (e.g., we never handle messageSent events). Without this mina-statemachine throws an exception if an Event is handled by the state machine.

The last piece of code we're going to have a look at is the code which creates the IoHandler proxy and the main() method:

```
private static IoHandler createIoHandler() {
    StateMachine sm = StateMachineFactory.getInstance(IoHandlerTransition.class).create(EMPTY, new TapeDeckServer());

    return new StateMachineProxyBuilder().setStateContextLookup(
            new IoSessionStateContextLookup(new StateContextFactory() {
                public StateContext create() {
                    return new TapeDeckContext();
                }
            })).create(IoHandler.class, sm);
}

// This code will work with MINA 1.0/1.1:
public static void main(String[] args) throws Exception {
    SocketAcceptor acceptor = new SocketAcceptor();
    SocketAcceptorConfig config = new SocketAcceptorConfig();
    config.setReuseAddress(true);
    ProtocolCodecFilter pcf = new ProtocolCodecFilter(
            new TextLineEncoder(), new CommandDecoder());
    config.getFilterChain().addLast("codec", pcf);
    acceptor.bind(new InetSocketAddress(12345), createIoHandler(), config);
}

// This code will work with MINA trunk:
public static void main(String[] args) throws Exception {
    SocketAcceptor acceptor = new NioSocketAcceptor();
    acceptor.setReuseAddress(true);
    ProtocolCodecFilter pcf = new ProtocolCodecFilter(
            new TextLineEncoder(), new CommandDecoder());
    acceptor.getFilterChain().addLast("codec", pcf);
    acceptor.setHandler(createIoHandler());
    acceptor.setLocalAddress(new InetSocketAddress(PORT));
    acceptor.bind();
}
```

createIoHandler() creates a StateMachine just like we did before except that we specify IoHandlerTransition.class instead of Transition.class in the call to StateMachineFactory.getInstance(...). This is necessary since we're now using the @IoHandlerTransition annotation. Also, this time we use IoSessionStateContextLookup and a custom StateContextFactory when we create the IoHandler proxy. If we didn't use IoSessionStateContextLookup all clients would share the same state machine which isn't desirable.

The main() method creates the SocketAcceptor and attaches a ProtocolCodecFilter which

decodes/encodes Command objects to its filter chain. Finally, it binds to port 12345 using an IoHandler instance created by the createIoHandler() method.

Advanced topics

Changing state programmatically

To be written...

Calling the state machine recursively

To be written...

Chapter 15 - Proxy

To be completed...

Chapter 16 - JMX Support

Java Management Extensions (JMX) is used for managing and monitoring java applications. This tutorial will provide you with an example as to how you can JMX-enable your MINA based application.

This tutorial is designed to help you get the JMX technology integrated in to your MINA-based application. In this tutorial, we will integrate the MINA-JMX classes into the imagine server example program.

Adding JMX Support

To JMX enable MINA application we have to perform following

- Create/Get MBean server
- Instantiate desired MBeans (IoAcceptor, IoFilter)
- Register MBeans with MBean server

We shall follow \src\main\java\org\apache\mina\example\imagine\step3\server\ImageServer.java, for the rest of our discussion

Create/Get MBean server

```
// create a JMX MBean Server server instance
MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();
```

This lines get the MBean Server instance.

Instantiate MBean(s)

We create an MBean for IoService

```
// create a JMX-aware bean that wraps a MINA IoService object.    In this
// case, a NioSocketAcceptor.
IoServiceMBean acceptorMBean = new IoServiceMBean( acceptor );
```

This creates an IoService MBean. It accepts instance of an acceptor that it exposed via JMX.

Similarly, you can add IoFilterMBean and other custom MBeans as well

Registering MBeans with MBean Server

```
// create a JMX ObjectName.    This has to be in a specific format.
ObjectName acceptorName = new ObjectName( acceptor.getClass().getPackage().getName() +
                ":type=acceptor,name=" + acceptor.getClass().getSimpleName());

// register the bean on the MBeanServer.    Without this line, no JMX will happen for
// this acceptor.
mBeanServer.registerMBean( acceptorMBean, acceptorName );
```

We create an ObjectName that need to be used as logical name for accessing the MBean and register the MBean to the MBean Server. Our application in now JMX enabled. Lets see it in action.

Start the Imagine Server

If you are using Java 5 or earlier:

```
java -Dcom.sun.management.jmxremote -classpath <CLASSPATH>
org.apache.mina.example.imagine.step3.server.ImageServer
```

If you are using Java 6:

```
java -classpath
<CLASSPATH> }}{{{}org.apache.mina.example.imagine.step3.server.ImageServer
```

Start JConsole

Start JConsole using the following command:

```
/bin/jconsole
```

We can see the different attributes and operations that are exposed by the MBeans

Chapter 17 - Spring Integration

This article demonstrates integrating MINA application with Spring. I wrote this article on my blog, and though to put it here, where this information actually belongs to. Can find the original copy at Integrating Apache MINA with Spring.

Application Structure

We shall take a standard MINA application which has following construct

- One Handler
- Two Filter - Logging Filter and a ProtocolCodec Filter
- NioDatagram Socket

Initialization Code

Lets see the code first. For simplicity we have omitted the glue code.

```
public void initialize() throws IOException {

    // Create an Acceptor
    NioDatagramAcceptor acceptor = new NioDatagramAcceptor();

    // Add Handler
    acceptor.setHandler(new ServerHandler());

    acceptor.getFilterChain().addLast("logging",
                    new LoggingFilter());
    acceptor.getFilterChain().addLast("codec",
                    new ProtocolCodecFilter(new SNMPCodecFactory()));

    // Create Session Configuration
    DatagramSessionConfig dcfg = acceptor.getSessionConfig();
```

```
            dcfg.setReuseAddress(true);
            logger.debug("Starting Server.....");
            // Bind and be ready to listen
            acceptor.bind(new InetSocketAddress(DEFAULT_PORT));
            logger.debug("Server listening on "+DEFAULT_PORT);
}
```

Integration Process

To integrate with Spring, we need to do following:

- Set the IO handler
- Create the Filters and add to the chain
- Create the Socket and set Socket Parameters

NOTE: The latest MINA releases doesn't have the package specific to Spring, like its earlier versions. The package is now named Integration Beans, to make the implementation work for all DI frameworks.

Lets see the Spring xml file. Please see that I have removed generic part from xml and have put only the specific things needed to pull up the implementation. This example has been derived from Chat example shipped with MINA release. Please refer the xml shipped with chat example.

Now lets pull things together

Lets set the IO Handler in the spring context file

```
<!-- The IoHandler implementation -->
<bean id="trapHandler" class="com.ashishpaliwal.udp.mina.server.ServerHandler">
```

Lets create the Filter chain

```
<bean id="snmpCodecFilter" class="org.apache.mina.filter.codec.ProtocolCodecFilter">
   <constructor-arg>
      <bean class="com.ashishpaliwal.udp.mina.snmp.SNMPCodecFactory" />
   </constructor-arg>
</bean>

<bean id="loggingFilter" class="org.apache.mina.filter.logging.LoggingFilter" />

<!-- The filter chain. -->
<bean                                                    id="filterChainBuilder"
class="org.apache.mina.core.filterchain.DefaultIoFilterChainBuilder">
   <property name="filters">
      <map>
```

```
            <entry key="loggingFilter" value-ref="loggingFilter"/>
            <entry key="codecFilter" value-ref="snmpCodecFilter"/>
        </map>
    </property>
</bean>
```

Here, we create instance of our IoFilter. See that for the ProtocolCodec factory, we have used Constructor injection. Logging Filter creation is straight forward. Once we have defined the beans for the filters to be used, we now create the Filter Chain to be used for the implementation. We define a bean with id "FilterChainBuidler" and add the defined filters to it. We are almost ready, and we just need to create the Socket and call bind

Lets complete the last part of creating the Socket and completing the chain

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
        <property name="customEditors">
            <map>
                <entry key="java.net.SocketAddress">
                    <bean class="org.apache.mina.integration.beans.InetSocketAddressEditor" />
                </entry>
            </map>
        </property>
</bean>

<!-- The IoAcceptor which binds to port 161 -->
<bean      id="ioAcceptor"      class="org.apache.mina.transport.socket.nio.NioDatagramAcceptor"
init-method="bind" destroy-method="unbind">
    <property name="defaultLocalAddress" value=":161" />
    <property name="handler" ref="trapHandler" />
    <property name="filterChainBuilder" ref="filterChainBuilder" />
</bean>
```

Now we create our ioAcceptor, set IO handler and Filter Chain. Now we have to write a function to read this file using Spring and start our application. Here's the code

```
public void initializeViaSpring() throws Exception {
        new ClassPathXmlApplicationContext("trapReceiverContext.xml");
}
```