MINA in Real Life
Schedule

• Introduction to MINA
• Simple Use Cases
• A more complex Use Case
• Do's and Don'ts
• Summary
• Q&A
Introduction
• A framework on top of NIO 1.0
  – Asynchronous
  – Non-blocking
  – Event-Driven
  – TCP, UDP, APR, Serial ...
  – Extensible through Filters
  – Comes with a protocol framework
Built for ADS

• ADS needed a SEDA based network framework on top of NIO
  – Netty-1 sound ok, but...
  – Needed a full rewrite
  – It became MINA 1.0
  – And later, a TLP!
Key concepts

- PERFORMS I/O
- FILTERS INCOMING AND OUTGOING MESSAGES
- APPLICATION LAYER

MINA
How it works...
Simple use cases
• A Simple TCP server: EchoServer
  – Based on TCP
  – Multi-Users
  – Should be fast
  – Returns what the users sent without modification

• Ok, let's code it!
public static void main(String[] args) throws Exception {
    SocketAccepter acceptor = new NioSocketAccepter();

    // Bind
    acceptor.setHandler(new EchoProtocolHandler());
    acceptor.bind(new InetSocketAddress(PORT));

    System.out.println("Listening on port " + PORT);
}
The “Business” part

```java
public class EchoProtocolHandler extends IoHandlerAdapter {

    /**
     * This is where we handle incoming messages
     */
    public void messageReceived(IoSession session, Object message)
        throws Exception {
        // Write the received data back to remote peer
        session.write(((IoBuffer) message).duplicate());
    }
}
```
And that's it!

- We have created a SocketAcceptor
- Then we associated a handler to it
- And accepted incoming connections
- Last, we implemented the logic in the Handler, in the messageReceived() method.
What do we have?

• A multithreaded server
• Accepting many parallel clients
• Roughly 4 lines of code!
• We can extended the server easily
  – For instance, adding a logger
  – Handling more messages
  – Or adding SSL support
public static void main(String[] args) throws Exception {
    SocketAcceptor acceptor = new NioSocketAccepter();

    // Add a logging filter
    acceptor.getFilterChain().addLast( "Logger", new LoggingFilter() );

    // Bind
    acceptor.setHandler(new EchoProtocolHandler());
    acceptor.bind(new InetSocketAddress(PORT));

    System.out.println("Listening on port ", PORT);
}
Another simple Use Case

• NTP Server
  – UDP (port 123)
  – Fixed Message size
  – Binary protocol
  – Stateless

• The code...
A more complex use case
A more complex Use Case

- Apache Directory Server
  - TCP and UDP
  - Simple or Two levels protocols
  - Binary messages
  - Multiple handlers
  - Potentially hundred of thousands connections
  - Has to be fast
Handle many protocols

- LDAP (TCP)
- Kerberos (TCP and UDP)
- NTP (UDP)
- DHCP (UDP)
- DNS (TCP and UDP)
- ChangePassword
LDAP protocol

- Binary protocol
  - Defined using ASN.1
  - BER encoded
- TCP based
- Connected
- More than one message type
Constraints

• Support LDAP and LDAPS
• Session can last forever
  – Small memory footprint
• Messages can be quite big
  – Images
• We can receive more than one message in an incoming buffer
• It should be Client and Server side
Decoding

• Problem: it's a 2 level protocol
  – TLVs
  – Ldap

• TLV means Type/Length/Value
  – Each of those three elements can be longer than one byte
  – A Value can contains other TLVs
LDAP messages

• 10 different requests
  – Bind, Unbind, Abandon, Add, Compare, Delete, Modify, ModifyDN, Search, Extended

• 11 different responses
  – Bind, SearchResEntry, SearchResDone, SearchResRef, Add, Compare, Delete, Modify, ModifyDN, Extended, Intermediate
Server Side

- The chain will contain the SSL filter, plus an executor, and the Ldap protocol codec
- We may have expensive requests
- We want more than one handler
- Each session contains user's data
SocketAcceptor acceptor = new NioSocketAcceptor( nbThreads );

IoFilterChainBuilder chain = new DefaultIoFilterChainBuilder();
chain.addLast( "sslFilter", new SslFilter( sslCtx ) );

chain.addLast( "codec", new ProtocolCodecFilter( getProtocolCodecFactory() ) );

chain.addLast( "executor",
    new ExecutorFilter(
        new OrderedThreadPoolExecutor( getNbThreads() ),
        IoEventType.WRITE ) );

acceptor.setFilterChainBuilder( chain );
...
Acceptor configuration

...  
// Disable the disconnection of the clients on unbind
acceptor.setCloseOnDeactivation( false );

// Allow the port to be reused even if the socket is in TIME_WAIT state
acceptor.setReuseAddress( true );

// No Nagle's algorithm
acceptor.getSessionConfig().setTcpNoDelay( true );

// Inject the protocol handler
acceptor.setHandler( getHandler() );

// Bind to the configured address
acceptor.bind();
Handlers

class LdapProtocolHandler extends DemuxingIoHandler
{
...
    public void messageReceived(IoSession session, Object message)
    {
        ... // SSL and controls Handling
        super.messageReceived(session, message);
    }
...}

public void messageReceived(IoSession session, Object message)
{
    MessageHandler<Object> handler = findReceivedMessageHandler(message.getClass());

    if (handler != null) {
        handler.handleMessage(session, message);
    } else {
        throw new UnknownMessageTypeException(...);
    }
}
Back to basic...
What about XML?

- Tagged language
- Size is unknown
- Parser are a bit a pain to use at this point
- Seems like XML is the Lingua Franca those days...
  - “a language used by people of diverse speech to communicate with one another, often a basic form of speech with simplified grammar.”
Issues

• We have to detect tags
• We have to detect text between tags
• We have to keep everything somewhere until we are done with the closing tag
• Java XML decoders don't handle fragmented tags...
An XML stripper server

• We want to extract the message in an XML message, and return it to the user
• The message can be big
• The decoder is the main concern...
• We have to validate the data before sending it to the handler.
public static void main( String[] args ) throws Exception {
    IoHandler xmlStripperProtocolHandler = new XmlStripperProtocolHandler();
    SocketAcceptor acceptor = new NioSocketAcceptor();
    acceptor.setReuseAddress( true );
    acceptor.setHandler( xmlStripperProtocolHandler );

    // Add the codec filter
    acceptor.getFilterChain().addLast( "codec",
        new ProtocolCodecFilter( new XmlStripperProtocolCodecFactory() ) );

    // Start the listener
    acceptor.bind(new InetSocketAddress(IP_PORT_DEFAULT));
}
public void messageReceived(IoSession session, Object message) {
    Document document = (Document)message;

    // Strip the XML from the <tags>
    String result = getChildren(document.getFirstChild());

    session.write(result);
}
XML codec factory

public class XmlStripperProtocolCodecFactory implements ProtocolCodecFactory
{
    public ProtocolEncoder getEncoder(IoSession session)
    {
        // Create a new encoder.
        return new XmlStripperEncoder();
    }

    public ProtocolDecoder getDecoder(IoSession session)
    {
        // Create a new decoder.
        return new XmlStripperDecoder();
    }
}
protected boolean doDecode(IoSession session, IoBuffer ioBuffer,
    ProtocolDecoderOutput decoderOutput ) {

    ...

decoderOutput.write( parserXML( data ) );

    ...
}

public Object parserXML( IoBuffer xmlBuffer ) {
    byte[] data = new byte[xmlBuffer.limit()];
    xmlBuffer.get( data );
    String xml = new String(data).trim();

    Document document = DocumentBuilderFactory.newInstance().
        newDocumentBuilder().parse(
            newDocumentBuilder().parse(
                new ByteArrayInputStream( xml.getBytes() ) ));

    return (document);
}
Do's and Don'ts
Do's !!!

- Follow the KISS principle
- Keep the chain short
- Do not use an executor if not needed
- Tune the number of IoProcessors
- Use only one codec filter
- If you have a problem, then your codec/handler probably sucks...
DON'Ts !!!

• Don't use the logging filter. Use Log4j.
• Your filter must be thread-safe
• Don't expect that you will receive data in one single block
• Don't forget about the negative impact Nagle's algorithm has on performance
• Don't use Direct buffers unless absolutely needed...
Summary
Q&A