NON-BLOCKING IO : CONCEPT AND FRAMEWORKS

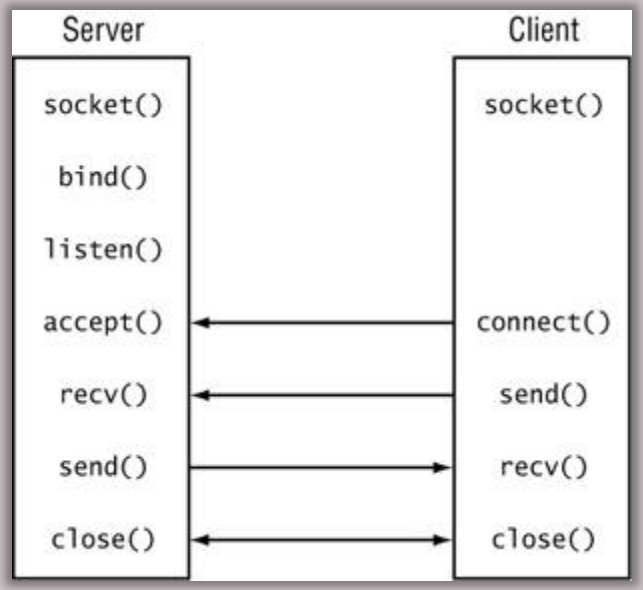
Ehab Ababneh

Outline

- The Problem: Thousands of Clients.
- Solution: Non-Blocking IO.
- The Reactor Pattern: Down to the roots of NBIO.
- NBIO is hard, just like multithreading.
- Frameworks are a bliss! … Apache MINA included.
- Example servers using Apache MINA and some of the benefits we get out of it:
  - Performance.
  - Separating low-level IO handling from the protocol from the business logic.
- More on MINA features.

The Problem

- Client/Server model is widely used computation model.
  - WWW, FTP, E-Mail, ….
- Servers nowadays need to serve thousands of users simultaneously.
- The major issue here is the time it takes a server to read data from the client and the time it takes the server to send the response back to the client.
  - Tens of milliseconds.



Typical steps in client/server communication

The Problem

- Typically the thread reading from a network socket blocks until all the data is received or a timeout is reached. Thus, the term *blocking IO*.
- A simple single-thread server using blocking IO can handle very few hundreds of client requests in one second.
  - Not good enough!
- Solution Approaches:
  - Multi-threading.
  - Non-blocking IO (Referred to as NBIO hereafter).
    - There are variations that fall under this category, but they are OS/Programming languages specific.

Solution Approaches – Multi-Threading

- Recall Homework Three.
- Several threads handling client requests.
- Once a connection between a client and a server is established a thread can read the request process it and send the response back to the client.
  - One-thread-per-client scheme.
- The number of threads created per-core is determined by the blocking coefficient.
  - Number of Threads = Number of Cores / (1 - Blocking Coefficient)

Solution Approaches – Multi-Threading

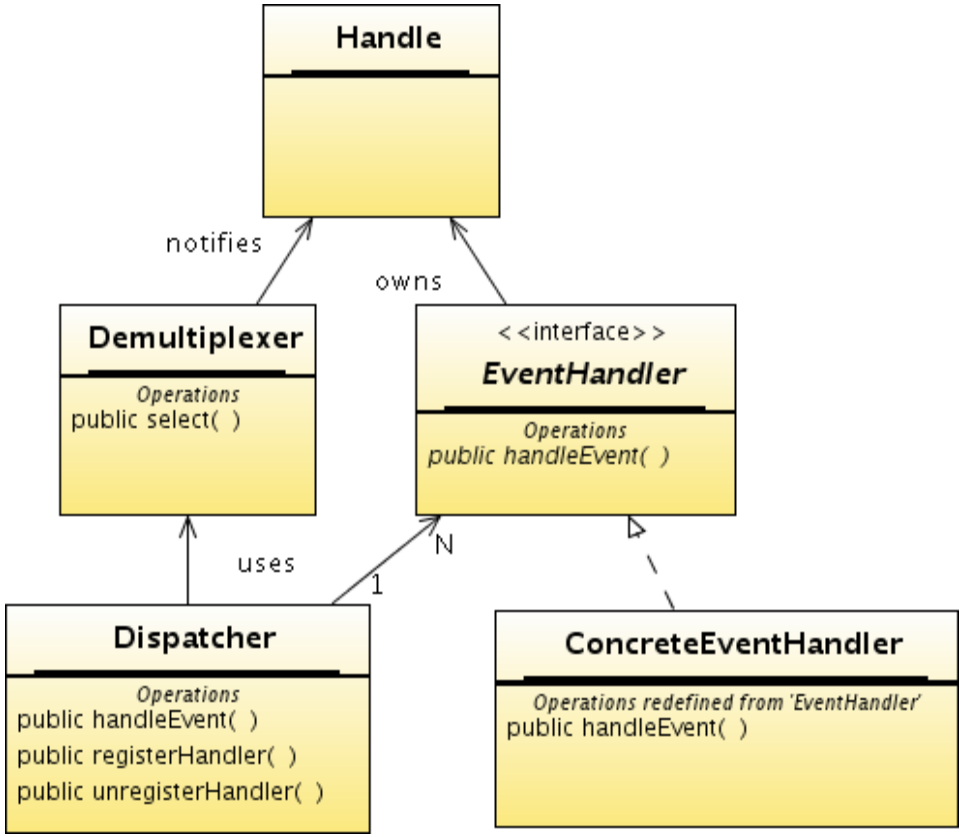
- In this approach a large number of threads (possibly hundreds of them) are created when the blocking coefficient is very close to 1.
  - Large files are sent over a network or high network latency.
- Operating systems may not do a good job when handling large number of threads.
  - Scheduling overhead and wasted CPU cycles in context switches.
- Significant amount of memory is invested in the threads' stack frames (2MB is a common default).
- This approach has many advocates (e.g. Eric Brewer creator of Inktomi [3]).

Solution Approaches – NBIO

- In this approach, as the name suggests, a thread is not blocked while it is performing an IO operation. Instead, …
  - it registers its interest in an IO operation and the operating system (OS) will handle performing that operation.
  - The OS will notify the thread of any events that occurred in that operation through a call back function supplied by the thread.
- As with threads, the support of the software stack between the user code and hardware (i.e. OS , JVM … etc) is necessary.
  - Supported in: windows NT v3.5 and later, Linux v2.6.x and later, Solaris 10…etc. And in Java 1.4 (code name Merlin) and later.
- We'll see an example server (Echo Server) written in Java using NBIO.

But, How Is It Possible?

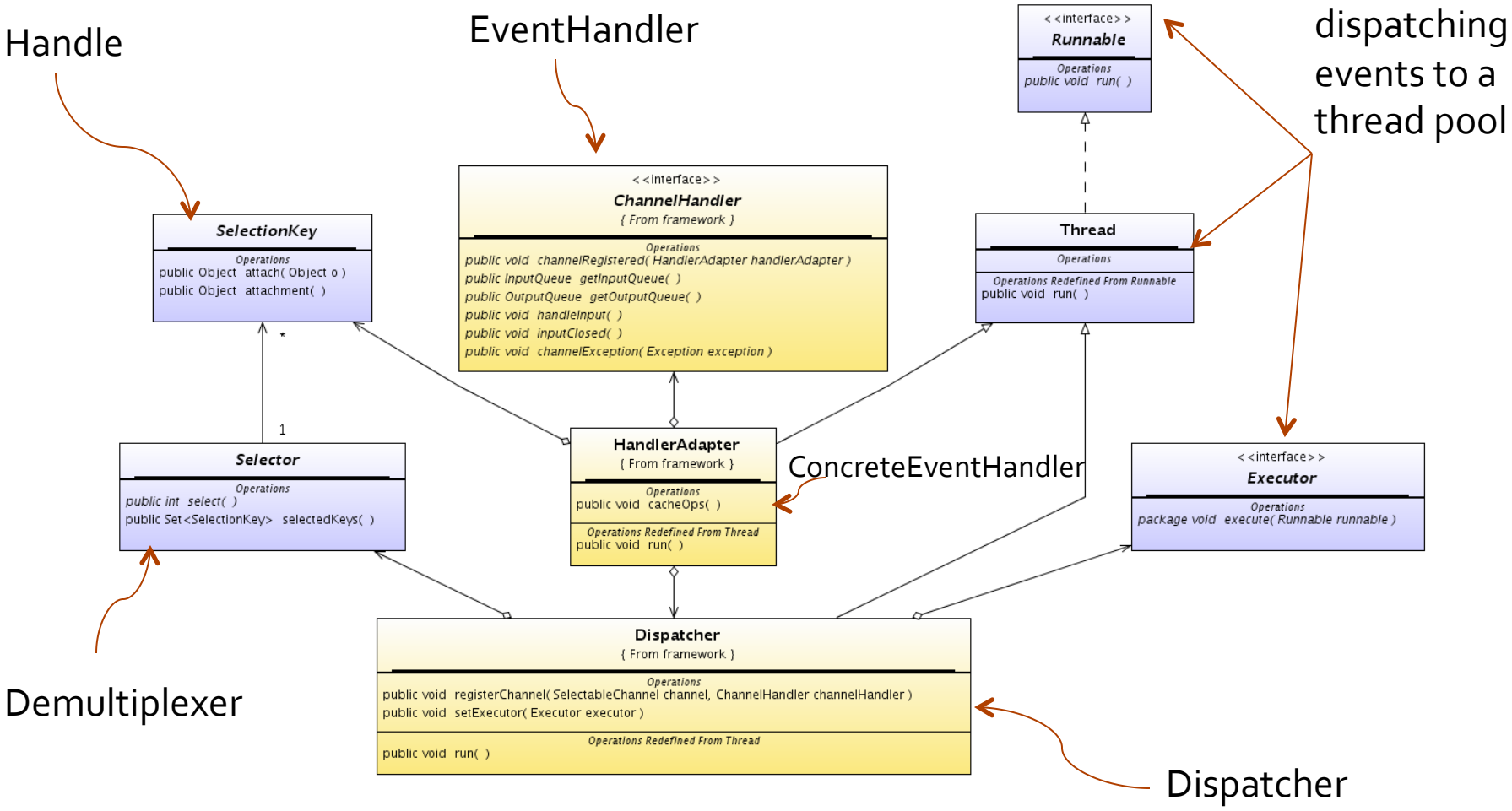
- The Reactor Design Pattern [7] :
  - Used to decouple the threads from the IO operations
  - Listens to events on sockets, files, … etc. and sends those events to interested threads.



But, How Is It Possible? – The Reactor

- The Reactor Design Pattern Actors and Dynamics:
  - Handle: a resource (file, socket, ..etc.)
  - Demultiplexer: Listens to events on Handles (data written, data arrived, timeouts, ..etc.). Exposes the select() method.
  - Dispatcher: uses the select() method from the demultiplexer to get events that happened on all handles and delivers each event to its appropriate Handler.
  - EventHandler: reacts to an event (e.g. defines a network protocol for data arrived on a network socket).

Mapping NBIO to Reactor (I)



Mapping NIO to Reactor (II) – Reactor Operation

The main thread of the reactor performs the following:

1. Create a new thread pool (an executor).
2. Create a new ServerSocketChannel, and bind it to a port.
3. Create a new Selector.
4. Register the ServerSocketChannel in the Selector, asking for accept readiness.
5. While(true)
   wait for notifications from the selector. For each notification arrived check:
   1. Accept notification: the server socket is ready to accept a new connection so call accept. Now a new socket was created so register this socket in the Selector.
   2. Write notification: For each socket which is ready for writing, check if the protocol asked to write some bytes. If so, try to write some bytes to the socket.
   3. Read notification: For each socket which is ready for reading, read some bytes and pass them down to the protocol handler. The actual work done by the protocol will be achieved with the use of the thread pool; *e.g.*, protocol processing is assigned as a task for the pool.

Dispatcher Loop

Reactor Operation

A First NBIO Server … (I)

```
public class EchoServer {
    private InetAddress addr;
    private int port;
    private Selector selector;
    private Map<SocketChannel,List<byte[]>> dataMap;

     public EchoServer(InetAddress addr, int port) throws IOException {
        this.addr = addr;
        this.port = port;
        dataMap = new HashMap<SocketChannel,List<byte[]>>();

        this.selector = Selector.open();
        ServerSocketChannel serverChannel = ServerSocketChannel.open();
        serverChannel.configureBlocking(false);

        InetSocketAddress listenAddr = new InetSocketAddress(this.addr, this.port);
        serverChannel.socket().bind(listenAddr);
        serverChannel.register(this.selector, SelectionKey.OP_ACCEPT);

        runServerLoop();
    }
```

Dispatcher Loop

Reactor Operation

A First NBIO Server … (II)

```
private void runServerLoop() throws IOException {
    while (true) {
        // wait for events
        this.selector.select();

        Iterator keys = this.selector.selectedKeys().iterator();
        while (keys.hasNext()) {
            SelectionKey key = (SelectionKey) keys.next();

            // this is necessary to prevent the same key from
            // coming up  again the next time around.
            keys.remove();

            if (! key.isValid())
                continue;
            if (key.isAcceptable()) {
                this.accept(key);
            }
            else if (key.isReadable()) {
                this.read(key);
            }
            else if (key.isWritable()) {
                this.write(key);
            }
        }
    }
}
```

Get Events from Multiplexer

Dispatch Events to Handlers

Dispatcher Loop

A First NBIO Server … (III)

```
private void accept(SelectionKey key) throws IOException {
    ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();
    SocketChannel channel = serverChannel.accept();
    channel.configureBlocking(false);

    Socket socket = channel.socket();
    SocketAddress remoteAddr = socket.getRemoteSocketAddress();
    log("Connected to: " + remoteAddr);

    // register channel with selector for further IO
    dataMap.put(channel, new ArrayList<byte[]>());
    channel.register(this.selector, SelectionKey.OP_READ);
}
```

* Only Handler for accept event is shown here, other handlers will be in the source code distributed along side the presentation.

Writing AIO Servers May Be Hard (I)

- As with multithreading, there are many intimate details about NBIO that when overlooked can lead to loss of performance.
  - Marking a channel as writeable too early results in the selecting thread spinning because 99% of the time a socket channel is ready for writing. And on win32, can produce disastrous performance problem, like freezing the OS by eating all the CPU.
- If a client sends data that won't fit into the buffer with the channel. Then multiple read events will be dispatched. Likewise, with writes.
  - That means we need to track the state of each read/ write over multiple dispatched events.

Writing NBIO Servers May Be Hard (II)

- The story of Rob Van Behren, adopted from [4]:
  - Set out to write a high-performance asynchronous server system
  - Found that when switching between clients, the code for saving and restoring values/state was difficult
  - Took a step back and wrote a finely-tuned, organized system for saving and restoring state between clients
  - When he was done, he sat back and realized he had written the foundation for a threading package

So, We Can Use Wrapper Frameworks!

- Just like the case with multithreading where many frameworks exist out there that provide many readily available multithreading models.
- NBIO Frameworks:
  - Java: Apache Mina.
  - C++: Boost::asio and POCO C++ Libraries.
  - Perl: IO::Async.
  - Python: Twisted.
- These libraries not just support NBIO for network communication, they also include asynchronous file operations.

Reexamine the Example Server…

- There are part of this server that are similar to any other NBIO server:
  - Setting up multi-channel sockets.
  - Selecting events from the selector and dispatching them in the dispatcher loop.
  - ➢ This is bootstrap logic.
- Other parts vary from one server to another:
  - Logic in all event handlers (accept, read, write, … etc.).
  - ➢ Those methods define the protocol.

Apache MINA

- Is an open source Java network application framework.
- Unified APIs for various transport protocols such as TCP, UDP and serial communication.
- Separates the low-level network handling API from the application logic.

Apache Mina – A First Example

- The Setup:
  - 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.

A Simple EchoServer Built Using MINA

```
public class MinaEchoServer {
    private static final int PORT = 8080;
    public static void main(String [] args) throws IOException
    {
        NioSocketAcceptor acceptor = new NioSocketAcceptor();
        acceptor.getFilterChain().addLast(
                "codec",
                new ProtocolCodecFilter(new  TextLineCodecFactory(Charset
                        .forName("UTF-8"))));

        acceptor.setHandler(new EchoIoHandler());
          acceptor.bind(new InetSocketAddress(PORT));
    }
}
```

Most of the functionalities in the reactor pattern are embedded in this class.

Codec: Converting byte stream into a logical message. The one used here is supplied by MINA.

Processes the messages.

Bootstrap

```
public class EchoIoHandler extends IoHandlerAdapter {

    @Override
    public void exceptionCaught(IoSession session, Throwable cause) throws Exception {
        session.close();
    }

    @Override
    public void messageReceived(IoSession session, Object message) throws Exception {
        session.write((String) message);
    }
}
```

Provides Several handlers for various events. These handlers can be overridden as we have done here with these two events.

Handler for caught exceptions.

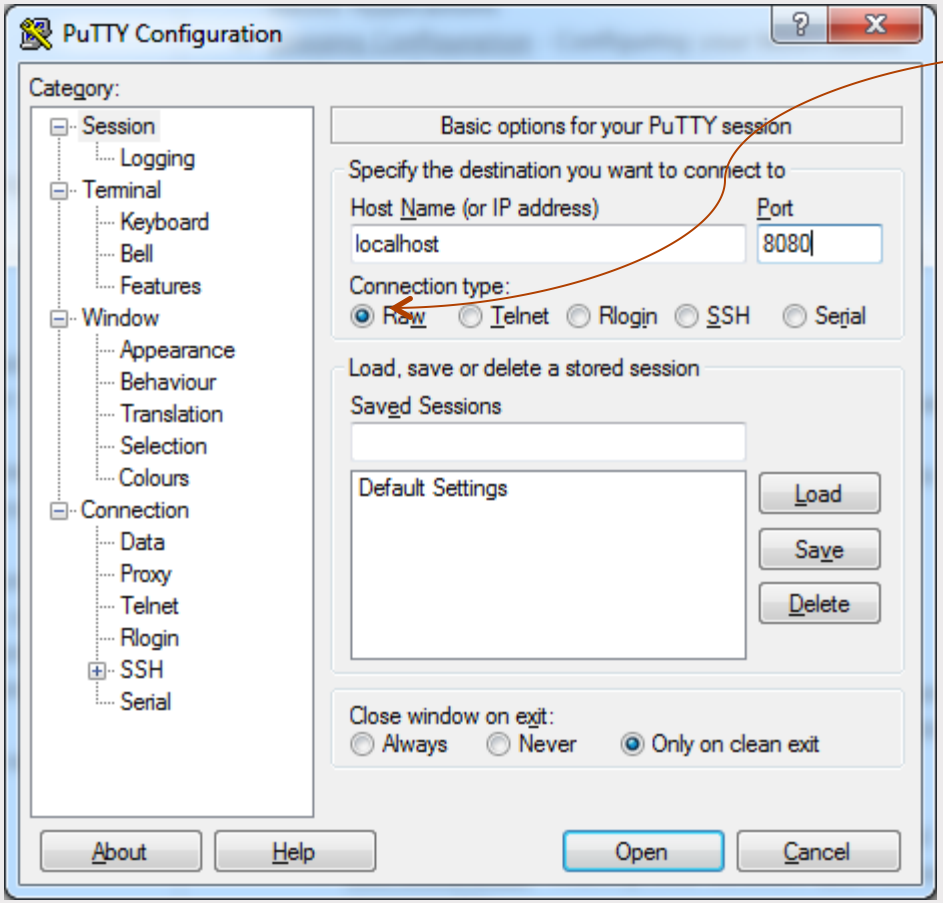
Handler for received messages.

This is an EchoServer. Just send back what ever was received in the message.

Protocol

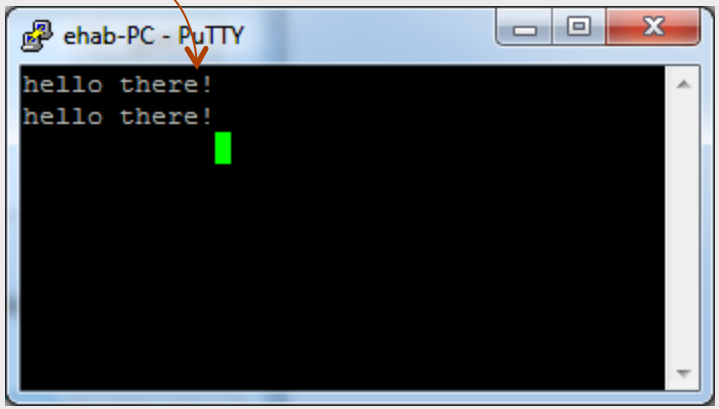
That is it!

Testing the EchoServer



Raw Messages

Type any message and the server will reply with the same message.



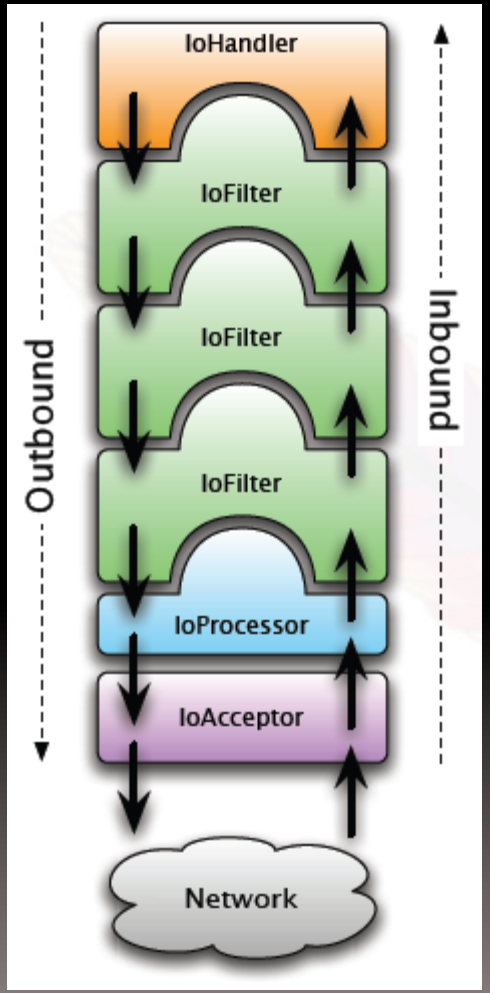
Putty can be downloaded from this link: http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

Comments About MinaEchoServer

- As we have seen, MINA takes care of the generic bootstrap code which usually requires significant optimization.
- You just need to plug in some code for handling the implemented protocol.
- MINA provides implementation for several protocols.
- Separating the protocol handler from the protocol decoder (TextLineCodecFactory) and the bootstrap logic (NioSocketAcceptor) separates the business logic (i.e. EchoIoHandler) from the protocol.
- MINA-based server have an architecture that is very extensible (described next). We could have added for example client black listing feature for example.

MINA-Based Server Architecture

- IoSession:
  - Holder of state for a connection (either client-side or
  - server-side)
  - Passed along with every event
  - Important Methods : (Write, Close, and get/setAttribut)
- IoHandler:
  - Akin to a Servlet
  - Endpoint of a filter chain
  - Important Methods: (sessionOpened, messageReceived, and sessionClosed)
- IO FilterChain:
  - Chain of IoFilter's for each IoSession
  - Can setup template chains per IoConnector/IoAcceptor
  - Dynamic addition/removal



MINA-Based Server Architecture

- IoFilters:
  - Akin to a ServletFilter
  - View/Hack/Slash the event stream
  - Important Methods: (sessionOpened, messageReceived, filterWrite, sessionClosed).
  - Currently built: (Logging, Compression, Blacklist, and SSL).
- IoAcceptor:
  - Server-side entry point.
  - Accepts incoming connections and fires events to an IoHandler.
  - Important Methods: (bind).
- IoProcessor:
  - Internal component
  - Handles reading and writing data to an underlying connection Each connection is associated with a single IoProcessor (shared amongst
  - multiple connections

Another MINA-Based Server

```
public class SWServer {
    public static final int PORT = 8330;

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

We will define our own codec instead of using a built in one.

We will also define our own handler.

```
public class MinaCodecFactory implements ProtocolCodecFactory {

    private final ProtocolEncoder encoder;
    private final ProtocolDecoder decoder;

    public MinaCodecFactory(boolean client) {
        encoder = new MinaResponseEncoder();
        decoder = new MinaRequestDecoder();
    }

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

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

This will define how a response message will be encoded into a byte stream to be transferred over the network.

This will define how a byte stream received from the network will interpreted into a message.

Another MINA-Based Server

```
public abstract class Message {

    protected int       flags         = 0; // The flag. Indicates if the message is zipped of flat, in addition to other info.
    protected int       sessionId     = 0; // The session Id assigned by the server to the custom client.
    protected int       messageLength = 0; // message length, when deflated (i.e. not zipped).
    protected int       packetLength  = 0; // packet length. If message is not zipped will same as message length.
    protected String    messageString = ""; // the actual XML message.

    // …….. Omitted from here are regular setters and getters.
}
```

Message class exchanged in our custom protocol.

```
public class MinaResponseEncoder extends ProtocolEncoderAdapter{

   public void encode(IoSession session, Object message, ProtocolEncoderOutput out) throws Exception {
      Message minaResponse = (Message) message;

      ByteBuffer buffer = ByteBuffer.allocate(0).setAutoExpand(true);

      buffer.putInt(minaResponse.getFlags());
      buffer.putInt(minaResponse.getSessionId());
      buffer.putInt(0);
      buffer.putInt(minaResponse.getMessageLength());
      buffer.putInt(minaResponse.getPacketLength());

      buffer.put(minaResponse.getMessageString().getBytes());
      buffer.flip();
      out.write(buffer);
   }
}
```

Encoder: Writes the message into a byte buffer to be sent out over he network.

Write the header info.

Write the XML part of the message.

Give the byte buffer to MINA for it to handling sending it out.

Another MINA-Based Server

```
public class MinaRequestDecoder extends CumulativeProtocolDecoder {
   protected boolean doDecode(IoSession session, ByteBuffer in, ProtocolDecoderOutput out) throws Exception {
      if (in.remaining() >= 12) {
         int flags = in.getInt();
         int sessionId = in.getInt();
         int packetLength = in.getInt();
         int messageLength = in.getInt();

         byte[] packet = new byte[packetLength];
         in.get(packet);

         Message request = new RequestMessage(flags, sessionId, packetLength, messageLength, packet);
         out.write(request);
         return true;
      } else {
         return false;
      }
   }
}
```

The decoder: will handle converting the byte stream received from the network into a message.

Read the header information.

Read the XML part of the message.

Create a message object.

MINA will send this decoder output to the protocol (IO Handler).

Another MINA-Based Server

```
public class SWServerIoHandler extends IoHandlerAdapter {

    public void sessionOpened(IoSession session) throws Exception {
    }

    public void exceptionCaught(IoSession session, Throwable cause) throws Exception {
        SessionLog.warn(session, cause.getMessage(), cause);
        session.close();
    }

    public void messageReceived(IoSession session, Object message) throws Exception {
        RequestMessage request = (RequestMessage) message;
        Random randomGenerator = new Random();
        int sessionId = request.getSessionId();

        Document doc = RequestParser.parseRequest(request.getMessageString());

        ResponseMessage response = RequestHandlerFactory.getRequestHandler(request).processRequest(request);
        session.write(response);
    }
}
```

Our Protocol Handler

Nothing interesting when session is created.

Close the session when something wrong happens.

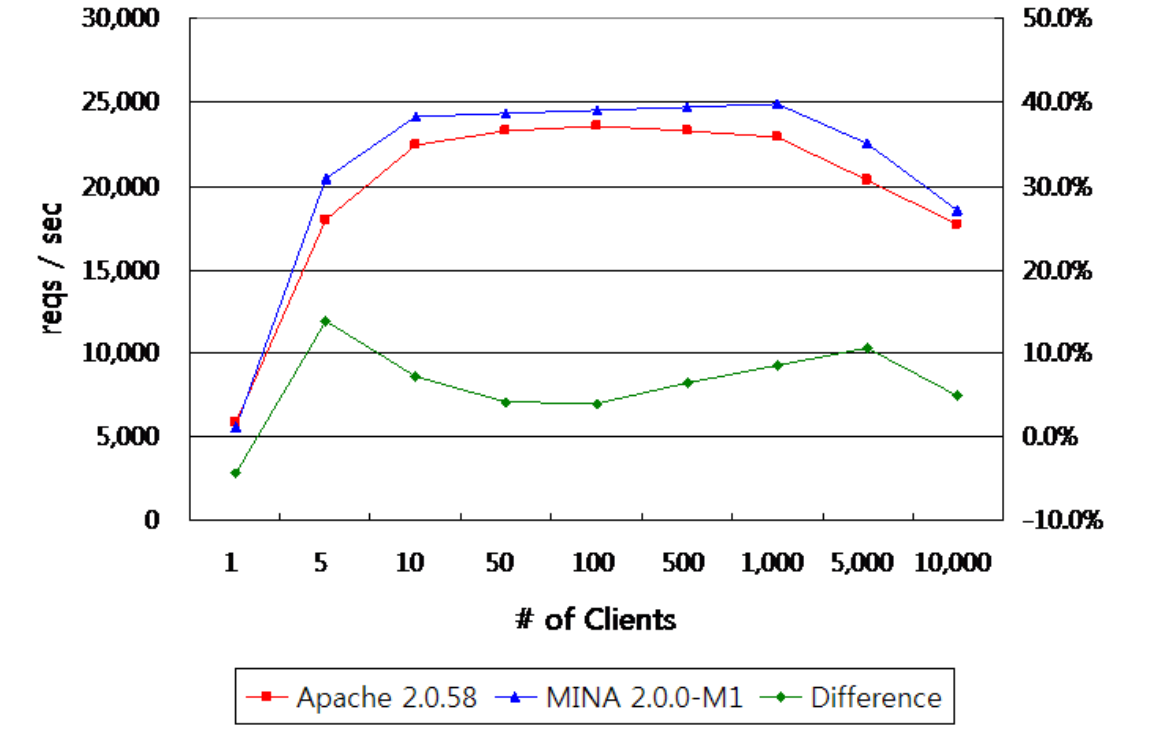
Handle the request and send the response message back.

Few More Features to talk about…

- MINA also:
  - Provides the means to build NBIO clients.
  - Provides 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.
  - Provides SSL Filters to server SSL requests.
  - Provides Customizable logging feature using log4j.
  - Integrates well with JMX and Spring.
  - Provides the means for unit testing through mock object creation for request simulation.

Performance Result

- Here is a well cited performance comparison between Apache MINA and the production ready Apache WebServer.
- Data size used is very small. The gap is expected to be bigger when using larger data sizes.



Who Uses Mina?

- SubEtha SMTP : http://code.google.com/p/subethasmtp/
- EURid: http://www.eurid.eu/ (After one hour MINA had handled more than 0.5 million SSL connections).
- Avis : The Avis event notification router and client library.
- The Apache Directory Project.
- QuickFIX – QuickFIXEngine.org : Financial Information eXchange Protocol.
- JStyx – JStyx.sf.net : Styx, a file sharing NFS-like protocol.
- AsyncWeb: http://mina.apache.org/asyncweb/.

Executive Summary

- The Problem:
  - Servers need to server thousands of clients simultaneously but the server blocks on IO ops.
  - Old solution: Multi-threading still wastes a lot of resources on context switching and may not scale well due to limitation on having 1000s of threads.
- New Solution: Non-Blocking IO…
  - Which Delegates the responsibility of handling IO to the OS and hardware.
  - Thus converting the IO intensive app into CPU intensive. Blocking coefficient goes lower.
  - Requiring less number of threads, impacting the architecture of the software.
- NBIO follows the Reactor Pattern.
- Just Like multithreading is hard, NBIO is hard as well…So, we'll use NBIO frameworks.
- Example NBIO framework: introducing Apache MINA.
- More about Apache MINA's features and benefits and architecture:
  - Separate the protocol from the business logic.

References

- [1] Network Performance: http://en.wikipedia.org/wiki/Network_performance
- [2] The C10K Problem by Dan Kegel: http://www.kegel.com/c10k.html
- [3] Why Events Are A Bad Idea (for high-concurrency servers), by Brewer et.al : http://static.usenix.org/events/hotos03/tech/full_papers/vonbehren/vonbehren.pdf
- [4] Thousands of Threads and Blocking I/O The old way to write Java Servers is New again: http://www.mailinator.com/tymaPaulMultithreaded.pdf
- [5] Apache Mina User Guide: http://mina.apache.org/user-guide.html
- [6] "JAVA NIO FRAMEWORK Introducing a high-performance I/O framework for Java" by Ronny Standtke and Ulrich Ultes-Nitsche.
- [7] "Reactor : An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events" by D. Shmidt: http://www.cs.wustl.edu/~schmidt/PDF/reactor-siemens.pdf