High-performance TCP/IP programming on the JVM

Networking API's, languages and programming-models for the JVM

June, 31st. 2009

Matthias Schmidt
Sun Microsystems, Inc.

Table of Contents

Networking API's, languages and programming-models for the JVM

Introduction

The purpose of this document is threefold. Firstly, it should give the reader a historical overview of the TCP/IP networking API's, programming paradigms, and various coding styles on the Java Virtual Machine. Secondly, it presents a couple of new options, programming-models, and demonstrates the same with Scala, one of the new languages available on the JVM. Lastly, a small and simple benchmark was conducted, to relate these different strategies with to each other on the metrics of performance.

The discussion is centred wholly around the Java Virtual Machine for a couple of reasons. First and foremost, one regularly deals with several programming languages and several libraries. Basing everything around the JVM provides the opportunity to eliminate issues such as the language implementation, memory management, interfaces to the host operating-system, etc. Besides this advantage, the JVM is the foundation for a large amount of Open Source libraries, tools, including excellent support from a couple of free IDEs that it became the platform of many web applications. The platform is mature, versatile and widely available.

To showcase the various API's, languages and programming models, and to give the programmer an impression as to how such source code looks like, five very trivial server programs that write a simple HTML page to any client, which tries to open a TCP/IP connection to such a server, were created. Since all five programs run on the Java Virtual Machine, this page-generating method has converged to a single utility-class, which is shared among the five programs. Therefore, risks arising from inconsistent behaviour and performance, because the programs were developed in different programming languages - Java and Scala, were eliminated.

The different programs that demonstrate this are listed below.

		Writing to...	**Connection/Thread ratio**
1	Plain Sockets	*OutputStream*	1:1
2	NIO	*ByteBuffer*	One has to code it oneself. Use Reactor-Pattern, *ThreadPool* etc. 1:n
3	Grizzly	*ByteBuffer*	ThreadPool management done by the Framework. Flexible. 1:n
4	Scala, Actor	*OutputStream*	Actors are implemented using the Fork/Join library from Doug Lea. 1:n
5	Scala, Actor + NIO	*ByteBuffer*	"

Classic Java networking IO:Sockets

The first and still the single most important Networking API for Java is the TCP/IP Socket support, which is available since the very beginning of Java. One has to bear in mind that TCP/IP network communications is being addressed here, rather than the networking protocols other than TCP/IP, such as Novell's IPX/SPX, DECnet, SNA , ISO/OSI or AppleTalk. Therefore the focus would be on TCP/IP, which has by now become synonymous with Networking.

On the one hand, the single most important interface to TCP/IP-networking is the Socket interface, introduced with BSD Unix. Java supported TCP/IP from the very first day by modelling Java interfaces and classes similar to the constructs used in the traditional C-API found on various Unix systems. Java's networking interfaces deal with the usual features - Socket, InetAddress, and the like.

On the other hand, Java I/O used to be centred around the *InputStream* and *OutputStream,* accompanied by the streams - *Reader* and *Writer*, as is the case with that of the File or Pipe interfaces. Extensive usage of the API uncovered the need to extend and to bridge the gaps in the features. The connection between sockets and streams is made, simply by retrieving the input and the output-streams from the sockets. Programmers are now able to treat those networking-streams just as they would with ordinary streams, and wrap them in suitable *Reader*'s and *Writer*'s that they are familiar with.

The Example program #1 demonstrates how TCP/IP server programs have been written for quite some time. A *ServerSocket* is the server side of a socket communication, which waits for a client to connect to the server. One usually creates a *ServerSocket* by providing a TCP/IP port to listen on, and to wait in the blocking call *accept* for clients trying to connect to it. But since *accept* blocks, one has to do the data transmission between client and server in a separate, newly created thread. Otherwise the server would stop listening for additional client connection requests while processing the previous request.

One connection = one Thread?

Threads are lightweight, cheap and constitute the basic building block for pretty much all Java applications. Java comes with Thread support out of the box. Java ,as a Language, and the Java Virtual Machine, as a platform, have been designed with Threading and multi-CPU machines in mind. There isn't anything wrong in using a thread for a connection, except the fact that one has to deal with multitudes of them. Things get quite ugly on some scenarios, especially when one deals with heavy traffic sites serving thousands of concurrent users. When the server and the code has to deal with such a high number of traffic and a constant rate of incoming requests, the price tag for creating a new Thread for each incoming request becomes quite high. It is just not efficient to create a new thread from scratch. Allowing the existing threads to service one single connection, and to tear the connection down once it's no longer used, in a much efficient approach.

A *ThreadPool* re-uses these pre-created threads, using some smart algorithms to size the pool, allowing some pressure reduction from the servers. Though this does offer some performance gain, there are several other challenges that one has to deal with in those heavy-load environments. These challenges are addressed in the later sections.

```
 3 import java.io.BufferedReader;
 4 import java.io.IOException;
 5 import java.io.InputStreamReader;
 6 import java.io.PrintWriter;
 7 import java.net.ServerSocket;
 8 import java.net.Socket;
 9
10 public class SimpleSocketServer {
11
12     public static void main(String[] args) throws IOException {
13
14         System.out.println("This is SimpleSocketServer running on port " + Utility.PORT);
15
16         ServerSocket serverSocket = null;
17         boolean listening = true;
18
19         try {
20             serverSocket = new ServerSocket(Utility.PORT);
21         } catch (IOException e) {
22             System.err.println("Could not listen on port:" + Utility.PORT);
23             System.exit(-1);
24         }
25
26         while (listening) {
27             new ThreadedHandler(serverSocket.accept()).start();
28         }
29
30         serverSocket.close();
31     }
32 }
33
34 class ThreadedHandler extends Thread {
35
36     private Socket socket = null;
37
38     public ThreadedHandler(Socket socket) {
39         super("ThreadedHandler");
40         this.socket = socket;
41     }
42
43     @Override
44     public void run() {
45
46         try {
47             PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
48
```

```
49              BufferedReader in = new BufferedReader(
                        new InputStreamReader(socket.getInputStream()));
50
51              in.readLine();
52              out.print(Utility.getPage("SimpleSocketServer", 512));
53
54              out.close();
55              in.close();
56              socket.close();
57
58          } catch (IOException e) {
59              e.printStackTrace();
60          }
61      }
62  }
```

References:

1. http://java.sun.com/docs/books/tutorial/networking/sockets/index.html
2. http://java.sun.com/javase/6/docs/api/java/net/package-summary.html
3. http://www.java2s.com/Tutorial/Java/0320__Network/Catalog0320__Network.htm
4. http://www.amazon.com/Network-Programming-Third-Elliotte-Harold/dp/0596007213

Java New I/O: Java NIO

Sun Microsystems introduced Java New I/O (Java NIO) with JDK 1.4.2 to address a couple of shortcomings of the basic Socket API, and to leverage some capabilities of modern operating systems. The main benefits of Java NIO are Buffers/Channels and non-blocking I/O. A short discussion on these aspects is provided here.

Buffers/Channels

The traditional Java API for networking dealt with sockets and streams derived from those sockets. This interface was quite concise and intuitive, which fit in nicely into existing Java APIs. Following the success of Java as a programming language for server-side software, some better alternatives, especially with regard to performance, were needed. As more and more communication on the internet became bulk transfer of media data, such as graphics, audio and video data, rather than small chunks of bytes from HTML pages and form fields which used to dominate the communication in the early days, the need for such alternatives became critical. Java had to cope with the rising levels of data that was increasingly passed around on the internet. Rephrasing the scenario, Java had to provide interfaces for facilities that were already implemented in the operating systems to efficiently transfer all this data.

These changed load patterns already led to some general performance optimization considerations for high-end server systems. One promising performance optimization was to reduce the number of times, a buffer filled with user data had to be copied as it moves up and down the entire software stack. Imagine, for example, a HTTP request hitting one's network interface card. This request hits the system as an ethernet frame, turns into a data-buffer in the device driver for the ethernet card, gets stored in a queue managed by the OS to be handled by the TCP/IP protocol state-machine code, makes it's way up into the Java Virtual Machine, and finally needs to be dispatched to the corresponding handler thread. Data buffers with unchanged payload get copied time and time again. Operation System developers addressed these issues, which is obviously not limited to Java, by streamlining their internal buffer handling, device-driver interfaces, network stacks and the way they handled DMA throughout the system. There was a kind of umbrella term for these technologies - Zero-Copy. No one actually managed to achieve zero-copy, but the long-term goal eventually became the name of the strategy.

These optimizations had one thing in common - they dealt with fixed size blocks, rather than the streams of data. To bring this evolution of operating system design to the Java Virtual Machine, and to give Java developers handles for these blocks to work with, Java NIO was invented. Only that they called the blocks *Buffers*. There is a basic, abstract class Buffer and various implementations for different data-types like longs, chars, floats, and most prominent the *ByteBuffer*. There is a coherent set of operations possible on those Buffers. This clearly established a set of pointers associated with each *Buffer* together with clear semantics what to do with them. These Buffers can and have been implemented by the different JVM vendors in native code by leveraging the buffer handling improvements in the underlying operating system. Since Buffers couldn't be nicely mixed with the existing Streams/Writer API's, there had to be, among other things, a new feature, the *Channels,* which are used to

handle those Buffers. One can think of Channels as some kind of sophisticated file-descriptors which can deal with the Buffers mentioned above. There are various Channels like *SocketChannel*, *FileChannel*, and the like. Looking at the classes and interfaces in java.nio.*, there are many more classes and interfaces introduced with Java NIO, but Buffers and Channels continue to constitute the backbone of Java NIO.

Non-blocking I/O

The second most important design principle, a better feature introduced with Java NIO, is something called Non-blocking *I/O*. While developed and therefore firstly available under Unix as the poll system call, Non-blocking I/O was completely new (and needed) for Java. It had a great effect in sustaining Java's success on the server side. Non-blocking I/O made it possible, that one thread could handle different I/O request. This means, in practice, that the various calls don't block until they are finished, but return immediately, by freeing the thread, to do other useful work. The most prominent example for this is the way the *accept* call works in conjunction with a new *Selector* class.

Example #1 from the previous chapter demonstrates this. The call to the *accept* method on the *ServerSocket* in line number 27 is where the program waits for something event to happen. This, in our case, is a connection request. The underlying snippet shows the call stack while the program is waiting for some event to occur:

```
Thread t@64259: (state = IN_NATIVE)
 - java.net.PlainSocketImpl.socketAccept(java.net.SocketImpl)
 - java.net.PlainSocketImpl.accept(java.net.SocketImpl)
 - java.net.ServerSocket.implAccept(java.net.Socket)
 - java.net.ServerSocket.accept()
 - com.sun.isve.SimpleSocketServer.main(java.lang.String[])
```

To analyse this condition, a high-traffic server has a high number of different requests such as new incoming connections, data-transfer, shut down connection going on, etc., all in parallel. So, in real life systems, the code doesn't have to wait that long for incoming connections, but has some hundreds of them happening at the very same moment. But that's not all. While dealing with those connection requests, a lot of clients generate read requests, and have completed calculations on the server, causing write requests that need to be pushed through the network back to the clients.

In the traditional model, all one could do is to fire up an new thread to deal with a new connection, to handle the request, and to wait till one is done with the activities. All code targeted at reading, writing, and monitoring the connection needs to be placed and handled by one thread. This creates a one to one linkage of one connection to one thread. This is fine if one doesn't have too many connections.

Java NIO solved this and a couple of other things by adapting the select()/ poll() mechanism to Java. The basic working principle is quite simple. First, one has to enable Non-blocking I/O by calling configureBlocking(false) on the *ServerChannel*. Then the code should register one of these operations (see *SelectionKey*):

1. **OP_READ**
2. **OP_WRITE**
3. **OP_CONNECT**
4. **OP_ACCEPT**

Together with a Channel with what is called a '*selector*' (*java.nio.channels.Selector*), one loops over a call to *select* on this Selector (line number 101). This *select* call returns when there is any kind of activity on this given Selector; whether it is a read, write or whatever, select will return if something happens. A look at the accept method in line 126 – 132 shows that there is something import happening - the client *SocketChannel* that has been just acquired from the accept call to the same selector is being registered. Selectors are just handles to gather the events that one is interested in, even is they stem from different channels.

With this very flexible semantic, one is able to handle different kind of requests anyway that one wants. Allowing one thread to handle all output, and another to handle all input, assigning priorities to the channels, because one would want to handle connections for admins with different priority than from some other channels processing low priority data, are all possible. With Java NIO one gets a flexible mechanism to tweak one's communication code to one's needs. One might wish to change one's requirements over time, or one might want to create a system, which is able to adjust itself to a changed environment automatically, NIO accommodates these changing work scenarios.

Networking API's, languages and programming-models for the JVM

NioServer example

The existing NioServer from the NIO Tutorial written by James Greenfield, has been simplified to present some NIO-style programming. The "The Rox Java NIO Tutorial" is highly recommended for those who wish to take a closer look at NIO (see references).

So, instead of blocking in the accept call on a ServerSocket (see example #1), with NIO, the blocking is now on the *select* call on a Selector. The big difference is that this *select* call returns if anything happens, not only incoming connections, but also on the Channel. This is basically multiplexing various activities over this Selector. In the example #2, the server dispatches different requests to different handler methods (here: read(), write(), accept()) which might process the request directly, fire up new threads or fetch handler threads from a pool. The JDK5 ThreadPool and ExecutorService is used to handle the requests.

One might say that the Java NIO is full of advantages, but there a price tag attached to it. The Example #2 runs approximately five times longer than the SimpleSocketServer from Example #1.

A word of caution is warranted here. All programs in this paper are trivial, synthetic and absolutely unrealistic examples. This is because these programs are showcase the basic principles, rather than depicting real life implementations. Production code looks quite different.

Besides this, the NIO example given here, does much more than the SimpleSocketServer. In real life, one is more likely to find programs similar to the NIO example, than that of the SimpleSocketServer given in #1. Both of them are quite unrealistic, since they don't do any processing on the incoming request, but only throw back a single page to the sender. Under normal circumstances server code has to fetch data from databases, do some processing, generate rather complicated HTML pages and so on. The amount of socket handling code becomes quite negligible.

But if one takes a careful look at the code given in Example #2, one could spot the real problem. The keyword synchronized used in the example would trigger a wary programmer in the right direction. Working with various threads on shared data, sockets, connections and buffers are freely passed around. This is considered a benefit, but again, there's a price attached to it. One has to synchronize the access to the shared data. Synchronizing shared data access uses low-level mechanisms like the *synchronized* keyword, and methods such as the *wait* and *notify* on the Object class. This can lead to either deadlocks or race conditions. Means to handle these issues are addressed in the next chapter.

```
 3 import java.io.IOException;
 4 import java.net.InetSocketAddress;
 5 import java.nio.ByteBuffer;
 6 import java.nio.channels.SelectionKey;
 7 import java.nio.channels.Selector;
 8 import java.nio.channels.ServerSocketChannel;
 9 import java.nio.channels.SocketChannel;
10 import java.nio.channels.spi.SelectorProvider;
11 import java.util.ArrayList;
12 import java.util.HashMap;
13 import java.util.Iterator;
14 import java.util.LinkedList;
15 import java.util.List;
16 import java.util.Map;
17 import java.util.concurrent.ExecutorService;
18 import java.util.concurrent.Executors;
19
20 class ChangeRequest {
21
22     public static final int REGISTER = 1;
23     public static final int CHANGEOPS = 2;
24     public SocketChannel socket;
25     public int type;
26     public int ops;
27
28     public ChangeRequest(SocketChannel socket, int type, int ops) {
29         this.socket = socket;
30         this.type = type;
31         this.ops = ops;
32     }
33 }
34
35 class Worker implements Runnable {
36
37     NioServer server;
```

```
 38      SocketChannel channel;
 39      ByteBuffer data;
 40
 41      public Worker(NioServer server, SocketChannel channel, ByteBuffer data) {
 42          this.server = server;
 43          this.channel = channel;
 44          this.data = data;
 45      }
 46
 47      public void run() {
 48          server.send(channel, ByteBuffer.wrap(
 49                  Utility.getPage("NioServer", 512).getBytes()));
 50      }
 51  }
 52
 53  public class NioServer implements Runnable {
 54      private ServerSocketChannel serverChannel;
 55      private Selector selector;
 56      private ByteBuffer readBuffer = ByteBuffer.allocate(8192);
 57      private final ExecutorService pool;
 58
 59      final private List pendingChanges = new LinkedList();
 60      final private Map<SocketChannel,List> pendingData = new
                 HashMap<SocketChannel,List>();
 61
 62      NioServer(int port) throws IOException {
 63          this.selector = initSelector(port);
 64          pool = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
 65      }
 66
 67      public void send(SocketChannel channel, ByteBuffer data) {
 68          synchronized (this.pendingChanges) {
 69              this.pendingChanges.add(
 70                      new ChangeRequest(channel, ChangeRequest.CHANGEOPS,
                             SelectionKey.OP_WRITE));
 71
 72              synchronized (this.pendingData) {
 73                  List queue = (List) this.pendingData.get(channel);
 74                  if (queue == null) {
 75                      queue = new ArrayList();
 76                      this.pendingData.put(channel, queue);
 77                  }
 78                  queue.add(data);
 79              }
 80          }
 81
 82          this.selector.wakeup();
 83      }
 84
 85      public void run() {
 86          while (true) {
 87              try {
 88                  synchronized (this.pendingChanges) {
 89                      Iterator changes = this.pendingChanges.iterator();
 90                      while (changes.hasNext()) {
 91                          ChangeRequest change = (ChangeRequest) changes.next();
 92                          switch (change.type) {
 93                              case ChangeRequest.CHANGEOPS:
 94                                  SelectionKey key = change.socket.keyFor(this.selector);
 95                                  key.interestOps(change.ops);
 96                          }
 97                      }
 98                      this.pendingChanges.clear();
 99                  }
100
101                  this.selector.select();
102
103                  Iterator selectedKeys = this.selector.selectedKeys().iterator();
104                  while (selectedKeys.hasNext()) {
105                      SelectionKey key = (SelectionKey) selectedKeys.next();
106                      selectedKeys.remove();
107
```

```
108                     if (!key.isValid()) {
109                         continue;
110                     }
111 
112                     if (key.isAcceptable()) {
113                         this.accept(key);
114                     } else if (key.isReadable()) {
115                         this.read(key);
116                     } else if (key.isWritable()) {
117                         this.write(key);
118                     }
119                 }
120             } catch (Exception e) {
121                 e.printStackTrace();
122             }
123         }
124     }
125 
126     private void accept(SelectionKey key) throws IOException {
127         ServerSocketChannel serverSocketChannel = (ServerSocketChannel) key.channel();
128 
129         SocketChannel socketChannel = serverSocketChannel.accept();
130         socketChannel.configureBlocking(false);
131         socketChannel.register(this.selector, SelectionKey.OP_READ);
132     }
133 
134     private void read(SelectionKey key) throws IOException {
135         SocketChannel socketChannel = (SocketChannel) key.channel();
136 
137         readBuffer.clear();
138 
139         int numRead;
140         try {
141             numRead = socketChannel.read(readBuffer);
142         } catch (IOException e) {
143             key.cancel();
144             socketChannel.close();
145             return;
146         }
147 
148         if (numRead == -1) {
149             key.channel().close();
150             key.cancel();
151             return;
152         }
153 
154         pool.execute(new Worker(this, socketChannel, readBuffer));
155     }
156 
157     private void write(SelectionKey key) throws IOException {
158         SocketChannel socketChannel = (SocketChannel) key.channel();
159 
160         synchronized (this.pendingData) {
161             List queue = (List) this.pendingData.get(socketChannel);
162 
163             while (!queue.isEmpty()) {
164                 ByteBuffer buf = (ByteBuffer) queue.get(0);
165                 socketChannel.write(buf);
166                 if (buf.remaining() > 0) {
167                     break;
168                 }
169                 queue.remove(0);
170             }
171 
172             if (queue.isEmpty()) {
173                 socketChannel.close();
174             }
175         }
176     }
177 
178     private Selector initSelector(int port) throws IOException {
179         Selector socketSelector = SelectorProvider.provider().openSelector();
```

```
180          InetSocketAddress isa = new InetSocketAddress(port);
181
182          serverChannel = ServerSocketChannel.open();
183          serverChannel.configureBlocking(false);
184
185          serverChannel.socket().bind(isa);
186          serverChannel.register(socketSelector, SelectionKey.OP_ACCEPT);
187
188          return socketSelector;
189      }
190
191      public static void main(String[] args) {
192          try {
193              System.out.println("This is NioServer running on port " + Utility.PORT);
194              new Thread(new NioServer(Utility.PORT), "NioServer").start();
195          } catch (IOException e) {
196              e.printStackTrace();
197          }
198      }
199 }
```

References:

1. http://java.sun.com/j2se/1.4.2/docs/guide/nio/
2. http://java.sun.com/javase/6/docs/api/java/nio/package-summary.html
3. http://today.java.net/cs/user/print/a/350
4. http://rox-xmlrpc.sourceforge.net/niotut/
5. http://www.javanio.info/

NIO Frameworks, enter: Grizzly

Problems with plain NIO

While explaining the various advantages of NIO in the previous chapter, it is quite evident that there are at least two downsides: there's more code involved and that one has to do low-level synchronization. This lead to the creation of NIO frameworks, which aim to relieve the programmer from much of the boiler-plate code and way more important, to hide the synchronization in the framework. It's much better to do this synchronization and other activities at a centralized location in a library, than to burden each programmer to create and to sustain this time and again.

Another very import aspect, which these NIO frameworks address, is that the Non-blocking I/O in NIO not only happens when working with Selectors, but also when reading and writing data to and from *SocketChannel*'s. In the traditional stream programming style, one reads from a stream until one has everything that one needs. In contrast, with NIO one calls read() on a SocketChannel and one gets only what is ready to be read from the JVM's perspective. The data probably got stuck somewhere in the OS or in the device driver, or more likely, hasn't arrived yet. So the bottom line is that the code has to deal with partial reads almost all the time.

Since NIO frameworks are, by definition, made to help to create high-performance, good quality server software such as application servers, the assembling of buffers into something useful and the encoding and decoding of protocols is one of the many add on benefits that one gets to use for free when using a framework.

Although the Grizzly NIO framework is what is addressed and used in creating the test program as has been done in the previous examples, its worth mentioning that there are quite a few good NIO frameworks that one could avail. The reference section in this chapter provides an overview of such frameworks.

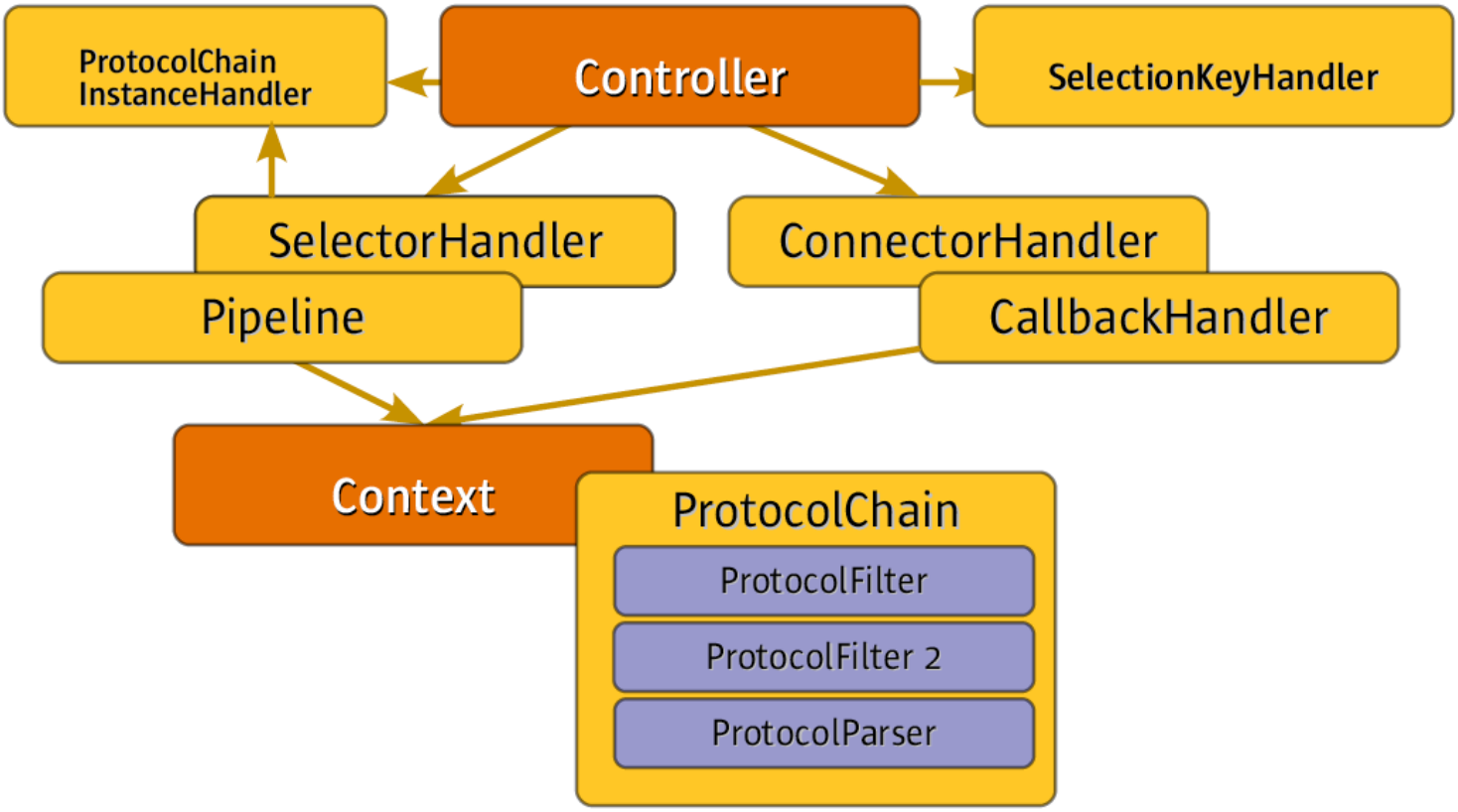
The Grizzly NIO framework

The Grizzly NIO Framework was originally created by Jean-Francois Arcand of Sun Microsystems in 2005. This framework was used as the foundation for the GlassFish application server. It has now matured into a core component of a few other TCP/IP based server projects such as Comet, Jetty, GlassFish, Jersey and Sailfin. Grizzly is supported by an active user and developer community. It is distributed under a Open Source licenses and comes with some very good example programs bundled with the software. For a good, up-to-date overview about Grizzly, the JavaOne 2008 presentation in the reference section could be made use of.

Grizzly makes writing TCP/IP applications easier by gathering all boiler-plate and thread-handling code in the framework. In addition, support for implementing protocols and simplifying packet assembly is provided by the framework. One could easily add and remove features by adding the so called '*filters*' to a ProtocolChain. Some examples of these filters are logging, TLS, etc. The neat thing about Grizzly is that it comes with a lot of good default implementations for the basic building blocks. In other words, one doesn't have to write much code oneself, if one is fine with the defaults. But one could change pretty much everything if one has to. One could code one's own SelectorHandler, ConnectorHandler or ProtocolChain if one would like to.

Networking API's, languages and programming-models for the JVM

Grizzly's overall architecture is as follows:



The 3rd incarnation of the test program, developed using Grizzly, is not dissected for comprehension. The central instance in a Grizzly based program is the *Controller* (`com.sun.grizzly.Controller`). Once the Controller is started with the *start* method, Grizzly starts its operations, which in turn creates a number of threads and answers to request. One can adjust the number of read-threads used by Grizzly (line 26) and one needs to set the TCP/IP port (line 30). Creating the ProtocolChainInstanceHandler is the only kind of boiler-plate code in this example; the real custom code is the class TimeBlockWriterFilter. These ProtocolFilter's can be chained to each other, and they do their actual work in the execute-method. The execute method is supplied with a *com.sun.grizzly.Context* which is the linkage between the framework and the particular filter. As one could observe, there are several elements that one gets to use for free when one uses the Grizzly framework, elements that one would have to develop oneself otherwise. Classes like the ProtocolParser help a lot, if one needs to create one's own protocols.

```
 3 import java.io.IOException;
 4 import java.nio.ByteBuffer;
 5 import java.nio.channels.SelectableChannel;
 6 import java.util.logging.Level;
 7 import java.util.logging.Logger;
 8
 9 import com.sun.grizzly.Context;
10 import com.sun.grizzly.Controller;
11 import com.sun.grizzly.DefaultProtocolChain;
12 import com.sun.grizzly.TCPSelectorHandler;
13 import com.sun.grizzly.ProtocolChain;
14 import com.sun.grizzly.ProtocolChainInstanceHandler;
15 import com.sun.grizzly.ProtocolFilter;
16 import com.sun.grizzly.util.OutputWriter;
17 import java.net.Socket;
18 import java.nio.channels.SocketChannel;
19
20 public class GrizzlyServer {
21
22     public static void main(String[] args) {
23         Controller controller = new Controller();
24
25         if (args.length > 0) {
26             controller.setReadThreadsCount(Integer.parseInt(args[0]));
27         }
28
29         TCPSelectorHandler tcpSelectorHandler = new TCPSelectorHandler();
30         tcpSelectorHandler.setPort(Utility.PORT);
31
32         controller.addSelectorHandler(tcpSelectorHandler);
33
34         ProtocolChainInstanceHandler pciHandler =
35             new ProtocolChainInstanceHandler() {
36
37                 final private ProtocolChain protocolChain = new DefaultProtocolChain();
38
39                 public ProtocolChain poll() {
40                     return protocolChain;
41                 }
42
43                 public boolean offer(ProtocolChain instance) {
44                     return true;
45                 }
46             };
47
48         controller.setProtocolChainInstanceHandler(pciHandler);
49
50         ProtocolChain protocolChain = pciHandler.poll();
51         protocolChain.addFilter(new TimeBlockWriterFilter());
52
53         try {
54             System.out.println("This is GrizzlyServer running on port " + Utility.PORT);
55             controller.start();
56         } catch (IOException ex) {
57             Logger.getLogger(GrizzlyServer.class.getName()).log(Level.SEVERE, null, ex);
58         }
59
60     }
61 }
62 class TimeBlockWriterFilter implements ProtocolFilter {
63
64     public boolean execute(Context ctx) throws IOException {
65         if (ctx.getProtocol() == Controller.Protocol.TCP) {
66             SelectableChannel channel = ctx.getSelectionKey().channel();
67             ByteBuffer buffer =  ByteBuffer.wrap(
68                     (Utility.getPage("GrizzlyServer", 512)).getBytes());
69
70             synchronized (channel) {
71                 Socket socket = ((SocketChannel) channel).socket();
72                 if (channel != null && !socket.isOutputShutdown()) {
```

```
73                      OutputWriter.flushChannel(channel, buffer);
74                      socket.close();
75                  }
76              }
77
78              buffer.clear();
79          }
80
81          return false;
82      }
83
84      public boolean postExecute(Context ctx) throws IOException {
85          return false;
86      }
87  }
```

References:

http://weblogs.java.net/blog/jfarcand/

http://blogs.sun.com/oleksiys/

https://grizzly.dev.java.net/

http://blogs.sun.com/oleksiys/resource/Grizzly-JavaOne2008.pdf

http://mina.apache.org/

http://xsocket.sourceforge.net/

http://www.jboss.org/netty/index.html

http://technfun.wordpress.com/2008/04/21/critique-of-java-nio-frameworks/

The Actor Model

Network API evolution

In the preceding sections the history of Java network programming was addressed, starting with the early, basic interfaces, moving on to the release of NIO, which was quite a success, and the inevitable rise of frameworks for NIO. The Example program #1's purpose was to demonstrate the general program-flow of traditional Java servers. But this style has it's limitations by assigning one thread to each connection. The invention of NIO was a huge step forward for Java network programming, but involved a lot of very tricky coding. Especially synchronization became more of an issue. So, it was quite a logical move to wrap these tasks and recurring requirements in tested libraries, and to use mature design patterns like the Reactor pattern. Regardless of the approach that one adopts, such as hiding the complexity (for example NIO coding) in a framework, one still faces the fundamental challenge of getting Java threading right. In an non-trivial Java server using one of these frameworks, one still has to develop a significant amount of threading code, which could be quite error prone.

Threading API evolution

The reason why multi-threaded programming is still so hard to get right is because it still needs to be merely done by reasoning. And this, in turn, stems from the Java's underlying synchronization, which is based on shared data guarded by locks. To get to the heart of the problem, if one doesn't happen to work on some experimental system using transactional memory or similar issues, shared data and locks still remain as the foundation for all computer systems. But this doesn't necessarily mean, that this should be the only API the programmers get to avail. In the Java world the next, the introduction of the Java concurrency utilities by Doug Lea, with are part of the JDK since version 5 (see java.util.concurrent), was long overdue. This was a huge step forward and the usage of these carefully implemented classes by default is strongly recommended. The Java concurrency utilities tend to be fast, reliable, and efficient implementations of many useful standard patterns used in concurrent programming should cover most of the development needs. Engineering one's own object/wait/synchronized code is strongly deprecated, and one should abstain from doing so wherever feasible. Grizzly for example uses these classes for thread management. With these approaches, the possibilities for effective usage have been exhausted, which some perceive as an accident waiting to happen, due to the extensive usage of shared data.

Actors: shared-nothing versus shared data

The disadvantages of the shared data model have been know since ages, and as early as 1973, a group of computer science researchers around Carl Hewitt proposed the Actor Model as a solution in their publication. The Actor Model is based on a shared-nothing principle. One could think of an actor as a independent thread equipped with a mailbox where it receives messages. All communication is done by actors sending messages to one other. This is fundamentally different from threads owning data which call into each others methods. If one lets others call into objects directly, one has to guard the fields carefully. In the actor model, by contrast, one sends immutable messages to the actor, and the actor implementation takes care about parallelism, message queuing, delivery and ordering. An actor has an *act* method as compared with a *run* method of a thread. The developer using the actor library can rely on the fact that the code is synchronized by definition. Coding the *act* method of an actor is like coding in a sequential fashion in the old days. The model gives the programmers a kind of window, where reflections on parallelism and locking are obviated. This makes actor-based systems way more reliable in the long run. Actor based languages and systems are especially used in large scale packet switching systems and the like. With the successes of multi-core and multi-CPU systems, even in the consumer space, the need for efficient and manageable solutions to facilitate the programmer to exploit the power of modern computer systems became a major issue. This issue could only be addressed by inspecting the entire computing stack - operating systems, languages and libraries.

References:

1. http://en.wikipedia.org/wiki/Actor_model
2. http://dli.iiit.ac.in/ijcai/IJCAI-73/PDF/027B.pdf
3. http://lamp.epfl.ch/~phaller/doc/haller07coord.pdf

Scala

In this section, a new, promising language running on top of the Java Virtual Machine, Scala, is presented. Two more incarnations of the test program demonstrate the language features that make Scala a robust development tool. The following citation from Scala's creators vouch for its suitability in reducing code size, among other benefits:

"Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages, enabling Java and other programmers to be more productive. Code sizes are typically reduced by a factor of two to three when compared to an equivalent Java application."

Scala was invented by Martin Odersky, Professor for Computer Science in Lausanne, the original author of Sun's javac compiler. Although this is not supposed to be a complete introduction of Scala, highlighting some main features of the language is warranted:

Functional and Object-Oriented programming combined

In Scala everything is an object, even code. That paves the way for features like closures, currying, high-order functions and DSL's. There are no primitive data-types like in Java anymore. Under the hood, some mapping to these primitive types (which still exist on the byte-code level) is done for performance reasons. But nothing of this is exposed to the language. There is polymorphism and even better: mixin's with the name traits. At the first glance this looks like multiple inheritance, but here it's done right. There is very sophisticated support for functional programming, but the hybrid nature of the language doesn't force programming in a purely functional style.

Statically typed

Scala is strictly statically typed, but providing the programmer the comfort of type inference. In practice this makes Scala sources look like dynamical languages like Groovy, but without all the disadvantages of dynamic languages.

Running inside the Java Virtual Machine

Scala gets compiled to JVM byte-code. It runs inside the Java Virtual Machine, it benefits from all improvements in JIT compiler technology and runs on all platforms supported by Java. One could instantly use Java classes, and therefore, leverage all the Open Source pearls out in the world. No JNI, marshalling, or wrapping is needed to use existing Java libraries.

Modern language constructs: closures, type interference, pattern matching, powerful literals etc.

Scala offers quite an extensive set of new and powerful features to be addressed here. There is pattern matching (equivalent to Java's case-statement on steroids), closures, currying, traits (mix ins), a flexible import and module system, intelligent control structures, easily facilitates the creation of DSL's, powerful literals, and direct support for XML just to name a few. The book, "Programming in Scala", mentioned in the reference section offers excellent reading about this programming language.

Excellent Actor library nicely integrated in the language

Scala has an intuitive, and a nice integration in the language, with an excellent actor library. This was the single most important reason, to choose Scala for network programming and to demonstrate actor style programming in real live. Besides this, the fact that it runs inside the JVM made it possible to have a common ground for a small benchmark.

References:

1. http://www.scala-lang.org/
2. http://www.nabble.com/Scala-Programming-Language-f20934.html
3. http://www.planetscala.com/
4. http://www.amazon.com/Programming-Scala-Comprehensive-Step-step/dp/0981531601

A simple Scala server using the Actor Model

Now let's take a look, how the Scala language and actors play together. Without much further ado, here's our page generating test program version four using Scala and actors:

```
 3 import scala.actors.Actor
 4 import scala.actors.Actor._
 5 import _root_.java.net.Socket
 6 import _root_.java.net.ServerSocket
 7 import _root_.java.io.OutputStreamWriter
 8 import _root_.java.io.InputStreamReader
 9 import _root_.java.io.LineNumberReader
10
11 case class Idle(worker: ServerWorker)
12 case class Connection(socket: Socket, id: Int)
13
14 class ServerWorker(val id: Int, val dispatcher: Dispatcher) extends Actor
15 {
16     def act()
17     {
18         loop
19         {
20             react
21             {
22                 case Connection(socket, id) =>
23                     handleConnection(socket)
24                     socket.close()
25                     dispatcher ! Idle(this)
26             }
27         }
28     }
29
30     override def hashCode(): Int = id
31
32     override def equals(other: Any): Boolean =
33         other match
34         {
35             case that: ServerWorker => this.id == that.id
36             case _                  => false
37         }
38
39     def handleConnection(socket: Socket) =
40     {
41         val os = socket.getOutputStream
42         val writer = new OutputStreamWriter(os)
43
44         val is = socket.getInputStream
45         val reader = new LineNumberReader(new InputStreamReader(is))
46
47         reader.readLine()
48         writer.write(Utility.getPage("ScalaServer", 512))
49         writer.flush()
50     }
51 }
52
53 class Dispatcher() extends Actor
54 {
55     import scala.collection.mutable.{Map, ListBuffer}
56     import _root_.java.util.Random
57
58     val idleWorkers = new ListBuffer[ServerWorker]
59     val busyWorkers = Map[Int, ServerWorker]()
60     val rng = new Random()
61
62     for (i <- 1 to Runtime.getRuntime().availableProcessors() * 4 + 1)
63     {
64         val w = new ServerWorker(i, this)
65         w.start()
66         idleWorkers += w
67     }
```

```
 68
 69     def act()
 70     {
 71         loop
 72         {
 73             react
 74             {
 75                 case Idle(worker) =>
 76                     busyWorkers -= worker.id
 77                     idleWorkers += worker
 78
 79                 case conn: Connection =>
 80                     val worker =
 81                         if (idleWorkers.length == 0)
 82                             busyWorkers.get(rng.nextInt(busyWorkers.size)).get
 83                         else
 84                         {
 85                             val w = idleWorkers.remove(0)
 86                             busyWorkers += w.id -> w
 87                             w
 88                         }
 89
 90                     worker ! conn
 91             }
 92         }
 93
 94     }
 95 }
 96
 97 class Server()
 98 {
 99     val name: String = "Main"
100
101     def run() =
102     {
103         val socket = new ServerSocket(Utility.PORT)
104         val dispatcher = new Dispatcher()
105         var i = 0
106
107         dispatcher.start()
108
109         while (true)
110         {
111             val clientConn = socket.accept()
112             i += 1
113             dispatcher ! Connection(clientConn, i)
114         }
115     }
116 }
117
118 object ScalaWebserver extends Application
119 {
120     println("This is ScalaWebserver running on port " + Utility.PORT);
121     new Server().run()
122 }
```

Over all, the code looks much cleaner, and therefore should be easier to maintain. Because Scala sports a new development syntax, and paradigm in the Java world, explanation on some parts of the Example program #4 is necessary.

The case classes defined in line 11 and 12 are immutable classes to be used as messages. One of the many benefits of case classes is, that the compiler generates appropriate factory methods to create them, the corresponding fields for the data defined in the constructor and the getters and setters. Therefore they are ready to use when needed (see line number 25, there's no need to do something like new Idle(...)).

This program starts its life in the body of the object ScalaWebserver, because it extends Application. The program consists of one Server running an endless loop (line 109) waiting on connection attempts dropping off the *accept* call (line 111). Then, the server sends a message to one dispatcher, whose sole purpose is to dispatch these new connections to a configurable number of actors with the name ServerWorker to process the these connections.

The ServerWorker overrides the hashCode and equals methods just to be a nice citizen in the List and Map it is stored into.

The servicing of the connection itself works very much the same like the SimpleSocketServer talked about in the first chapter.

But to get there is to take a different approach. This might not look that beneficial in this elementary test program, but is definitely so in large scale systems. Actors send and receive messages, so a closer look on achieving this in Scala is demonstrated.

Sending messages

The sending of messages is written as simple as this:

```
receiver ! message
```

In our case, a connection is passed from the Server to the dispatcher in line 113, and the dispatcher sends a connection to a worker in line 90. Once the workers job it's done, it signals this by sending an Idle message back to the dispatcher in line 25. The dispatchers main tasks is to keep track of a free, and busy list of workers and to dispatch new connections to either idle workers, or if this fails, randomly to the already busy workers. Since messages are queued and delivered in-order by the actor library, no messages, meaning connections, are lost or dropped.

Receiving messages

Receiving messages in actor style coding is done by pattern matching inside an structure like this:

```
loop {
    react {
        case Connection(socket, id) => doSomething()
        }
    }
```

All of this has do be done in the actor's *act* method, which is the actors equivalent to a thread's *run* method. The creation of all the ServerWorker actors isdone only once when constructing the dispatcher (line 62-67), but after that everything happens in the loop/react section of the actor.

One might might wonder, how this loop and react cascading works? This syntax can be used in the Scala language, since code is treated like data, and could therefore be passed around. If one knows closures, from other languages like Groovy, one should have already worked with this concept. One turns out chunks of code into data objects. It's a bit like the function pointers in the C programming language, but a lot safer. Scala just adds the syntactic beautification to allow methods, in our case here loop and react, to get their only parameter inside of curly braces, rather than ordinary braces. So, it's the loop method getting as parameter the code, which consists of a single react method getting a partial regex, a case without the match expression, which hopefully matches incoming messages.

Combining the Actor Model and NIO

Now it's about time to present the last example program (example #5), which is only a small modification of the previous example. It's basically the actor base server which now uses NIO instead of plain sockets and streams. To achieve this, the Socket.accept() call was exchanged with a Selector/Key handling mechanism (line 105 - 125). This code dispatches the *read* and *accept* requests on the socket to appropriate methods. In practice, socket-writes obviously need to be handled as well. The only other difference is, that its not the sockets but channels that are carried around. This offers the chance to write ByteBuffer's to the channel. But the overall design is the same: Server -> Dispatcher -> ServerWorker(n).

```
 3 import scala.actors.Actor
 4 import scala.actors.Actor._
 5
 6 import _root_.java.nio.channels.Selector
 7 import _root_.java.nio.channels.spi.SelectorProvider
 8 import _root_.java.nio.channels.SelectionKey
 9 import _root_.java.nio.channels.Selector
10 import _root_.java.nio.channels.ServerSocketChannel
11 import _root_.java.nio.channels.SocketChannel
12 import _root_.java.net.InetSocketAddress;
13 import _root_.java.nio.ByteBuffer
14
15 case class Idle(worker: ServerWorker)
16 case class Read(channel:SocketChannel, id: Int)
17
18 class ServerWorker(val id: Int, val dispatcher: Dispatcher) extends Actor
19 {
20     def act()
21     {
22         loop
23         {
24             react
25             {
26                 case Read(channel, id) =>
27                     channel.write(ByteBuffer.wrap((
                                Utility.getPage("ScalaNioServer", 512).getBytes())))
28                     channel.close()
29
30                     dispatcher ! Idle(this)
31             }
32         }
33     }
34
35     override def hashCode(): Int = id
36
37     override def equals(other: Any): Boolean =
38         other match
39         {
40             case that: ServerWorker => this.id == that.id
41             case _                  => false
42         }
43 }
44
45
46 class Dispatcher() extends Actor
47 {
48     import scala.collection.mutable.{Map, ListBuffer}
49     import _root_.java.util.Random
50
51     val idleWorkers = new ListBuffer[ServerWorker]
52     val busyWorkers = Map[Int, ServerWorker]()
53     val rng = new Random()
54
55     for (i <- 1 to Runtime.getRuntime().availableProcessors() * 4 + 1)
56     {
57         val w = new ServerWorker(i, this)
```

```
 58          w.start()
 59          idleWorkers += w
 60      }
 61
 62      def act()
 63      {
 64          loop
 65          {
 66              react
 67              {
 68                  case read: Read =>
 69                      getWorker() ! read
 70
 71                  case Idle(worker) =>
 72                      busyWorkers -= worker.id
 73                      idleWorkers += worker
 74              }
 75          }
 76
 77      }
 78
 79      def getWorker(): ServerWorker = {
 80          if (idleWorkers.length == 0)
 81              busyWorkers.get(rng.nextInt(busyWorkers.size)).get
 82          else
 83          {
 84              val w = idleWorkers.remove(0)
 85              busyWorkers += w.id -> w
 86              w
 87          }
 88      }
 89
 90  }
 91
 92  class SelectingRunnable()
 93  {
 94      val selector = initSelector(Utility.PORT);
 95
 96      def run() =
 97      {
 98          val dispatcher = new Dispatcher()
 99          var i = 0
100
101          dispatcher.start()
102
103          while (true)
104          {
105              selector.select()
106
107              val selectedKeysItr = selector.selectedKeys().iterator()
108
109              while (selectedKeysItr.hasNext()) {
110                  val key = selectedKeysItr.next().asInstanceOf[SelectionKey]
111                  selectedKeysItr.remove();
112
113                  if (!key.isValid()) {
114                      continue;
115                  }
116
117                  if (key.isAcceptable()) {
118                      accept(key)
119                  } else if (key.isReadable()) {
120                      i += 1
121                      dispatcher ! Read(key.channel().asInstanceOf[SocketChannel], i)
122                      key.cancel
123                  }
124              }
125          }
126      }
127
128      def accept(key:SelectionKey) = {
129          val serverSocketChannel = key.channel().asInstanceOf[ServerSocketChannel]
```

```
130          val socketChannel = serverSocketChannel.accept()
131
132          socketChannel.configureBlocking(false);
133          socketChannel.register(selector, SelectionKey.OP_READ);
134      }
135
136
137
138      def initSelector(port:int): Selector = {
139          val socketSelector = SelectorProvider.provider().openSelector()
140          val serverChannel = ServerSocketChannel.open()
141          val isa = new InetSocketAddress(port)
142
143          serverChannel.configureBlocking(false)
144          serverChannel.socket().bind(isa)
145          serverChannel.register(socketSelector, SelectionKey.OP_ACCEPT)
146
147          return socketSelector
148      }
149 }
150
151 object ScalaNioServer extends Application
152 {
153      println("This is ScalaNioServer running on port " + Utility.PORT);
154      new SelectingRunnable().run()
155 }
```

The Benchmark

This paper has been written to sketch out a little bit of the history of Java networking API's, introduce a modern language and a completely new programming paradigm applied to the Java Virtual Machine. An unrealistic micro-benchmark on these little test programs was performed to observe the performance. It is to be noted that this simple benchmark should not be used to draw any final conclusions about the languages, tools or the API's that were used towards the benchmark results. The main purpose of this paper is to showcase the various options for network programming, and to show how such coding looks like. It was never intended to be a fair benchmark. Without going into too much detail as to how a reasonable benchmark should look like, a short note on the most evident shortcomings of this code is presented:

- The server sends-out data by just knocking on the socket. Real-life applications parse incoming URL's and act differently upon them.
- The connection is closed right after the request. Real-live applications keep the connection open by either using HTTP 1.1 or setting the "Connection: Keep-Alive" header flag, when running with HTTP 1.0.
- No serious work is done on the server, but only some numbers are churned out. This causes a compute-result/handle-connection ratio which one could never see under normal circumstances.
- The generation of the result page, which is used as a kind of work simulation, takes the same time for each request. In real world, servers are facing quite different requests, which take varying time spans to be serviced. As a result, various requests overlap on the server and generate completely different load patterns.

So, to wrap this up, code to demonstrate different coding styles was created, and this can at best, be used to benchmark the socket handling of the OS, the JVM and the Scala and Java libraries. One should expect the numbers for real live applications to look quite differently from the ones given in the chart below. Nevertheless, information on the setup, and the results of this micro-benchmark are shared here, since there still might be some insights gained from it.

Benchmark setup

ApacheBench was used as the benchmark driver with a statement simple as this:

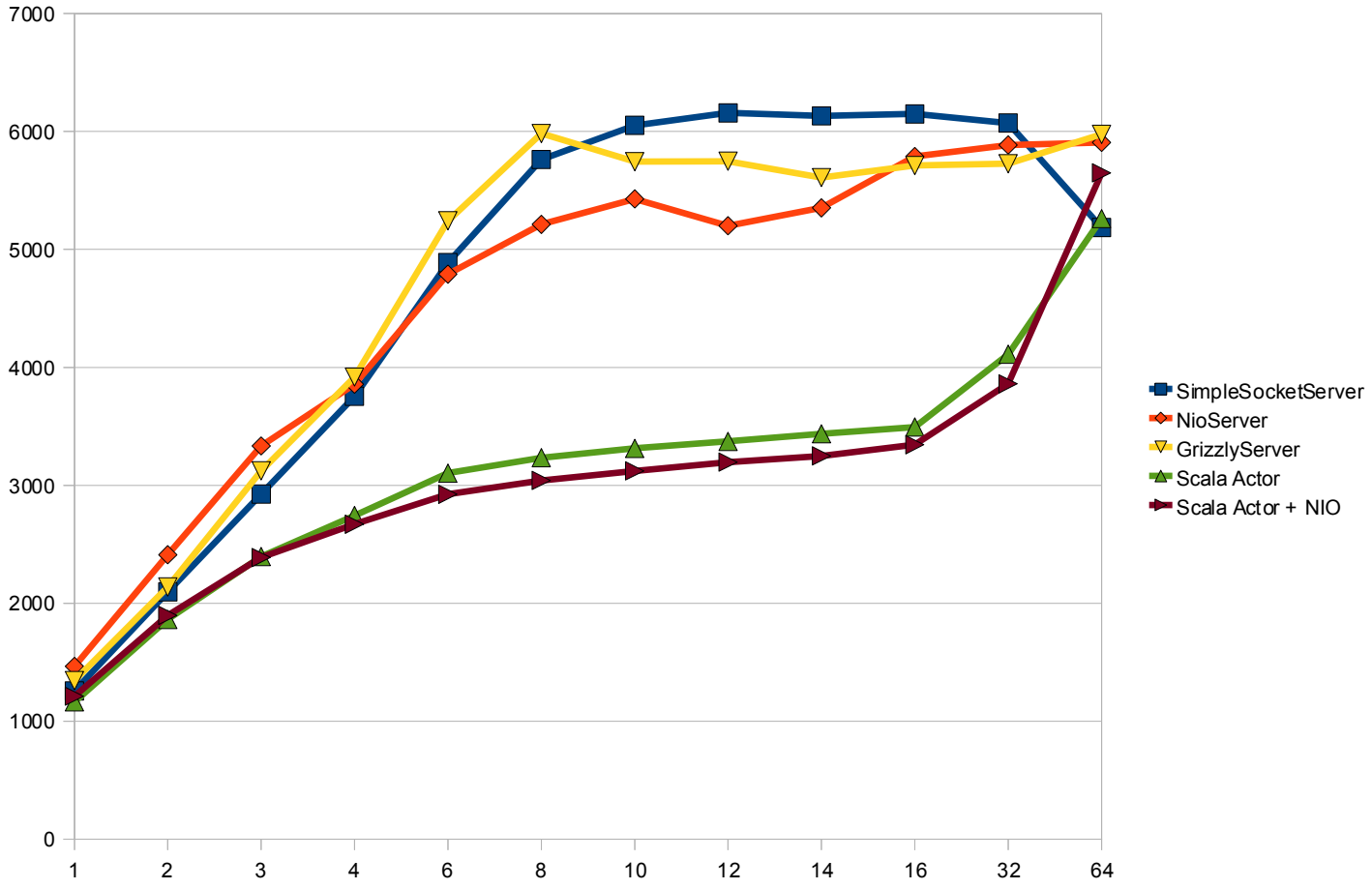
```
while true; do ab -c <n> -n 30000 'http://solarisx64:1405/gettimeblock' ; sleep 60; done
```

where n is the number of concurrent requests issued by ApacheBench. The numbers 1-4, 6, 8, 10, 12, 14, 16, 32, 64 as shown in the table below, were used.

The machine used for the benchmark was a Sun Fire X4600M2 equipped with 8 AMD Quadcore Opteron 8384 (2.7 GHz), with 256 GB RAM running Solaris 10. Tests were conducted on a JVM, patched to the latest level, i.e. JDK6 (build 1.6.0_14-b08), running with the Server VM and 2 GB of fixed heap. Scala version 2.7.5 was used for the experiments on Scala. Runs for each concurrency were conducted five times, dropping the first result, and computing the average of the remaining four runs.

This simplified table lists the results together:

	LOC	1	2	3	4	6	8	10	12	14	16	32	64
SimpleSocketServer	62	1257	2097	2926	3755	4888	5762	6052	6158	6132	6149	6071	5186
NioServer	197	1465	2411	3334	3858	4791	5213	5428	5201	5353	5788	5885	5907
GrizzlyServer	86	1345	2137	3122	3918	5243	5983	5745	5748	5610	5712	5727	5976
Scala Actor	122	1164	1863	2396	2744	3104	3234	3314	3373	3437	3494	4112	5262
Scala Actor + NIO	155	1211	1897	2388	2671	2925	3041	3122	3197	3250	3345	3860	5649



Observations

As mentioned before, given the simplicity of the code, one has to be careful to not draw quick conclusions from this chart. But what is very remarkable, and to some extend expected, is the behaviour of the code when facing heavy concurrent load. The interesting range in this diagram is from 16 to 64 connections. The decrease in throughput with the SimpleSocketServer is in contrast the rise in throughput by using Scala actors, especially, when the number of physical cores of the system crosses 32, proves the point that:

1. Simple socket programming with a connection/thread ratio of 1:1 has it's hard limit at the number of CPU's in the system. Since businesses generally shooting for having more customers (aka concurrent connections) than CPU's in there systems, the simple socket approach is just not suitable for such applications.
2. The actor model in general and the fork/join based implementation of the actor library in Scala shines under heavy load.
3. It's very unlikely to achieve better performance by doing raw NIO coding instead of using an NIO framework like Grizzly. Not to mention how easy it is on the other hand, to introduce subtle bugs.

Page generation

All five programs shared the same implementation for generating a single, simple HTTP response with an HTML page made of numbers derived from the current time. Here's the implementation:

```
 3 public class Utility {
 4
 5     public static final int PORT = 1405;
 6
 7     public static String checkUrlAndReply(String serverName, String request) {
 8         if (request.startsWith("GET /gettimeblock HTTP")) {
 9             return getPage(serverName, 512);
10         } else {
11             return "Invalid URL";
12         }
13     }
14
15     public static String getTimeBlock(int size) {
16         StringBuilder sb = new StringBuilder();
17         String nowNano;
18
19         int i = 1;
20
21         while (i <= size) {
22             nowNano = String.valueOf(System.nanoTime()).substring(8, 16);
23             sb.append(nowNano);
24             i++;
25         }
26
27         return sb.toString();
28     }
29
30     public static String getPage(String serverName, int size) {
31
32         StringBuilder sb = new StringBuilder();
33
34         sb.append("HTTP/1.1 200 OK\n");
35         sb.append("Date: Sat, 07 Mar 2009 00:00:00 GMT\n");
36         sb.append("Server: " + serverName + "\n");
37         sb.append("Accept-Ranges: bytes\n");
38         sb.append("Content-Length: $$$$\n");
39         sb.append("Connection: close\n");
40         sb.append("Content-Type: text/html\n");
41         sb.append("\n");
42
43         String header = sb.toString();
```

```
44          String body = "<html><body><h1>" + getTimeBlock(size) + "</h1></body></html>";
45
46          header = header.replace("$$$$", String.valueOf(body.length()));
47          String result = header + body;
48
49          return result;
50      }
51  }
```

References:

http://httpd.apache.org/

Conclusion

Having taken a look at five test programs, used four different programming models, such as plain sockets, NIO, Grizzly and actors, and two programming languages, one faces the question of facing the correct choice for the problem on hand. Unfortunately, there is no one simple answer for this concern. As always, it depends on the number of concurrent connections one expects, on the number of systems at hand, on the number of programmers on the team, their skill set, and experience, etc. It depends on so many factors that it's impossible to give an recommendation here which holds true once and forever.

Even so, there are many observations one could make by looking at these examples and the resulting charts. There are, in fact, some conclusions that one could draw. But some general considerations hold good before one delves into the question on the choice.

One has to bear in mind that many software engineers are in the lucky situation, not to be troubled by such considerations at all. They have to adhere to existing standards such as servlets or beans. Only a few programmers write system software such as webservers, application servers, or even frameworks. The majority of the people use these frameworks and servers.

To a certain extent, all mentioned approaches get the job done. It only depends, what the job is. If one has to do some simple socket communication sharing some data, and one doesn't expect many concurrent users, one might be better off using simple sockets. There's no need to over-engineer simple tasks. At the end, it's choosing the right tool for the job. The limitations of the simple socket server approach has been adequately addressed, so no further discussions on the same lines are necessary. The bottom line is the following - it's just not suitable for large scale, multi-user systems.

If the tasks is to write high-end networking software, one could not use, what is already publicly available, and what one would like. One has to, to stick with Java; a clear recommendation for some of the good features of NIO frameworks would be the best approach. One of the many reasons is that one would have to code much of the functionality of such a NIO framework oneself. These are way more efficiently done in a library shared, tested and maintained by a larger user base, rather than written from scratch in house. On top of that, there exists a lot of knowledge about usage-patterns and best-practices accumulated in these frameworks. If the requirements on one's system rise, one is very likely find ready-to-use classes in the framework.

Another factor is the sympathy or dislike for a certain programming language or coding style. Many people consider Scala a potential candidate to replace Java as the leading programming language on the JVM platform in the long run. The actor programming model already proved its fitness for large scale multi-threaded systems. Since actors are quite nicely integrated in Scala and the implementation is very efficient, Scala and actors are definitely worth a look. The brevity but expressiveness of the language combined with the intuitive messages sending syntax makes Scala programs usually shorter and way easier to debug. One can even think of combinations like for example using Scala as one's programming language of choice, instead of the actor based communications using a NIO framework such as Grizzly.