JAVA NIO FRAMEWORK
Introducing a High-performance I/O Framework for Java

Ronny Standtke and Ulrich Ultes-Nitsche
Telecommunications, Networks & Security Research Group, University of Fribourg
Boulevard de Pérolles 90, CH-1700 Fribourg, Switzerland

Keywords: Java, NIO, high-performance, security, framework.

Abstract: A new input/output (NIO) library that provides block-oriented I/O was introduced with Java v1.4. Because of its complexity, creating network applications with the Java NIO library has been very difficult and build-in support for high-performance, distributed and parallel systems was missing. Parallel architectures are now becoming the standard in computing and Java network application programmers need a framework to build upon. In this paper, we introduce the Java NIO Framework, an extensible programming library that hides most of the NIO library details and at the same time provides support for secure and high-performance network applications. The Java NIO Framework is already used by well-known organizations, e.g. the U.S. National Institute of Standards and Technology, and is running successfully in a distributed computing framework that has more than 1000 nodes.

1 INTRODUCTION

Multicore and manycore processing units are now becoming the standard in computing architectures. The expected growth in number of cores per unit is a challenge for software engineers in almost all fields.

Java network application programmers have two basic choices:

- use the classical I/O API (Streams, blocking I/O)
- use the NIO API (ByteBuffers, non-blocking I/O)

The classical I/O API is very easy to use, even for network connections secured with SSL. Because it uses blocking I/O, the *one thread per socket* multiplexing strategy must be used to serve several network connections. Even though this API scales with the number of available cores, the runtime performance of network applications using this API is very poor.

The NIO API provides mechanisms for non-blocking I/O. With non-blocking I/O it becomes possible to use *readiness selection* as the multiplexing strategy for serving several network connections. On the other hand, programming with the NIO API is very difficult. A whole book has been written about it (Hitchens, 2002). If attention is paid to all the necessary NIO details, a programmer must write a lot of so-called boilerplate code, even for the most simple network application. The complexity is increased many times if NIO is combined with SSL for secure network connections or support for high-performance, parallel systems. Many of these challenges and their proposed solutions are described in (Pitt, 2005).

The Java NIO Framework presented here is an extensible programming library that solves many problems that Java network application programmers face when using the original NIO library:

- In contrast to the original NIO library the Java NIO Framework has a very simple API, hiding all unnecessary details.
- Support for securing network connections with SSL is an integral part of the library instead of providing a separate, add-on engine.
- The I/O processing performance of network applications using the Java NIO Framework automatically scales with the number of available cores.

We started the work on the Java NIO Framework after Ron Hitchen's presentation "How to Build a Scalable Multiplexed Server With NIO" at the JavaOne Conference 2006 (Hitchens, 2006). Although there have been other frameworks available, e.g. (Lee, 2006; Arcand, 2006; Roth, 2006; Shetty, 2006), none of them had the envisioned scalability and ease of use. The Java NIO Framework has been published in August 2007 and is available at `http://nioframework.sourceforge.net`. It is Free Software released under the GNU Lesser General Public License version 3.

Standtke R. and Ultes-Nitsche U. (2008).
JAVA NIO FRAMEWORK - Introducing a High-performance I/O Framework for Java.
In *Proceedings of the Third International Conference on Software and Data Technologies - PL/DPS/KE*, pages 206-211
DOI: 10.5220/0001901502060211
Copyright © SciTePress

2 MULTIPLEXING STRATEGIES

Two multiplexing strategies were mentioned in the introduction, *one thread per socket* and *readiness selection*. In the next sections both strategies are briefly introduced and analyzed.

2.1 One Thread Per Socket

Threads are a mechanism to split a process into several simultaneously running tasks. Threads differ from normal processes by sharing memory and other resources. Therefore they are often called lightweight processes. Switching between threads is typically faster than switching between processes.

When a server uses the *one thread per socket* multiplexing strategy it creates one thread for every client connection. When executing blocking I/O operations the thread is also blocked until the operation completes its execution (e.g. when reading data from a socket the thread blocks until new data is available to read from the socket).

This strategy is very simple to implement because every thread just continues its operation after returning from a blocking operation and all internal states of the thread are automatically restored. A programmer can implement the thread (more or less) as if the server handles only one client connection.

The drawback of this multiplexing strategy is that it does not scale well. Each blocked thread acts as a socket monitor and the thread scheduler is the notification mechanism. Neither of them was designed for such a purpose.

A remaining problem of this strategy is that a design with massive parallel threads naturally is prone to typical threading problems, e.g. deadlocks, lifelocks and starvation.

2.2 Readiness Selection

Readiness selection is a multiplexing strategy that enables a server to handle many client connections simultaneously with a single thread. An overview of readiness selection is given in (James O. Coplien, 1995) when presenting the reactor design pattern.

The reactor design pattern proposes the software architecture presented in Figure 1.

- The class *Handle* identifies resources that are managed by an operating system, e.g. sockets.
- The class *Demultiplexer* blocks awaiting events to occur on a set of Handles. It returns when it is possible to initiate an operation on a Handle without blocking. The method `select()` returns which Handles can have operations invoked on them synchronously without blocking the application process.

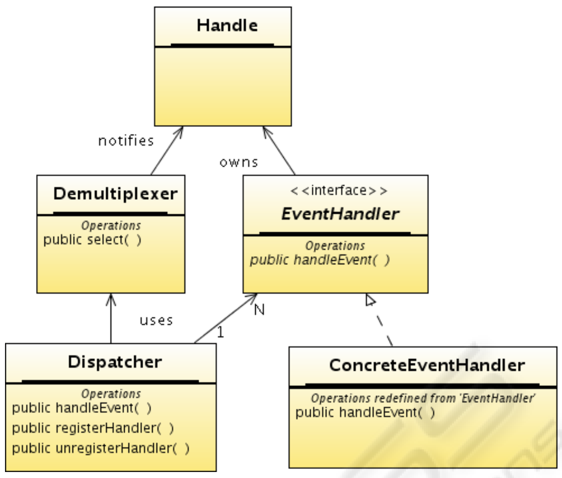

Figure 1: Reactor design pattern.

- The class *Dispatcher* defines an interface for registering, removing, and dispatching EventHandlers. Ultimately, the Demultiplexer is responsible for waiting until new events occur. When it detects new events, it informs the Dispatcher to call back application-specific event handlers.
- The interface *EventHandler* specifies a hook method that abstractly represents the dispatching operation for service-specific events.
- The class *ConcreteEventHandler* implements the hook method as well as the methods to process these events in an application-specific manner. Applications register ConcreteEventHandlers with the Dispatcher to process certain types of events. When these events arrive, the Dispatcher calls back the hook method of the appropriate ConcreteEventHandler.

Readiness selection scales much better but it is not as easy to implement as the *one thread per socket* strategy.

3 NIO FRAMEWORK DESIGN

Because the NIO Framework should be scalable to handle thousands of network connections simultaneously, the decision was made to use *readiness selection* as the multiplexing strategy, which is much more appropriate for high-performance I/O than the *one thread per socket* strategy.

3.1 Mapping the Reactor Design Pattern

If the reactor design pattern presented above had been used for the NIO Framework without modification, every application-specific ConcreteEventHandler would still have to take care of many NIO specific details. These include buffers, queues, incomplete write operations, encryption of data streams and much more. To provide a simple API to Java network application programmers, the NIO Framework was complemented with several additional helper classes and interfaces that will be introduced in the following sections.

The concepts and techniques used to design and implement a safe and scalable framework that effectively exploits multiple processors are presented in (Peierls et al., 2005).

A simplified model of the NIO Framework core is shown in Figure 2.

The blue UML elements (Runnable, Thread, Selector, SelectionKey and Executor) are part of the Java Development Kit (JDK). The interface Runnable and the class Thread were part of JDK from the very beginning, Selector and SelectionKey have been added to the JDK with the NIO package in JDK v1.4 and the interface Executor was added with the concurrency package in JDK v1.5. The yellow UML elements (ChannelHandler, HandlerAdapter and Dispatcher) are the essential core classes of the NIO Framework.

The Dispatcher is a Thread that runs in an endless loop, processes registrations of ChannelHandlers with a channel (a nexus for I/O operations that represents an open connection to an entity such as a network socket) and uses an Executor to offload the execution of selected HandlerAdapters. The Executor interface hides the mechanics of how each task will be executed, including details of thread use, scheduling, etc. This abstraction is necessary because the NIO Framework may be used on a wide range of systems, from low-cost embedded devices up to high-performance multi-core servers.

The class Selector determines which registered channels are ready.

The class SelectionKey associates a channel with a Selector, tells the Selector which events to monitor for the channel and holds a reference to an arbitrary object, called “attachment”. In the current architecture the attachment is a HandlerAdapter.

The EventHandler from the reactor design pattern is split up into several components. The first component is the class HandlerAdapter. It manages all the operations on a channel (connect, read, write, close) and its queues, interacts with the Selector and SelectionKey classes and, most importantly, hides and encapsulates most NIO details from higher level classes and interfaces.

The second EventHandler component in the NIO Framework is the interface ChannelHandler. It defines the methods that any application-specific channel handler class has to implement so that it can be used in the NIO framework. These include:

```
public void channelRegistered(
        HandlerAdapter handlerAdapter)
```

This method gets called when a channel was registered at the Dispatcher. It is mostly used on server type applications to send a welcome message to clients that just connected.

```
public InputQueue getInputQueue()
```

This method returns the InputQueue that will be used by the HandlerAdapter, if there is data to be read from the channel.

```
public OutputQueue getOutputQueue()
```

This method returns the OutputQueue that will be used by the HandlerAdapter, if there is data to be written to the channel.

```
public void handleInput()
```

The HandlerAdapter calls this method, if the InputQueue has new data to be read from the channel.

```
public void inputClosed()
```

This method gets called by the HandlerAdapter, if no more data can be read from the InputQueue.

```
public void channelException(
        Exception exception)
```

The HandlerAdapter calls this method, if an exception occurred while reading from or writing to the channel.

Not shown in Figure 2 are all the application-specific channel handlers that implement the interface ChannelHandler. They represent the ConcreteEventHandler of the reactor design pattern. Because the details of the method `handleInput()` may vary with every specific handler they are outside the scope of the NIO Framework.

Table 1 shows the mappings from the reactor design pattern to the NIO Framework.

3.2 Parallelization

Some parts of the Java NIO Framework are parallelized by default, other parts can be customized to be parallelized.

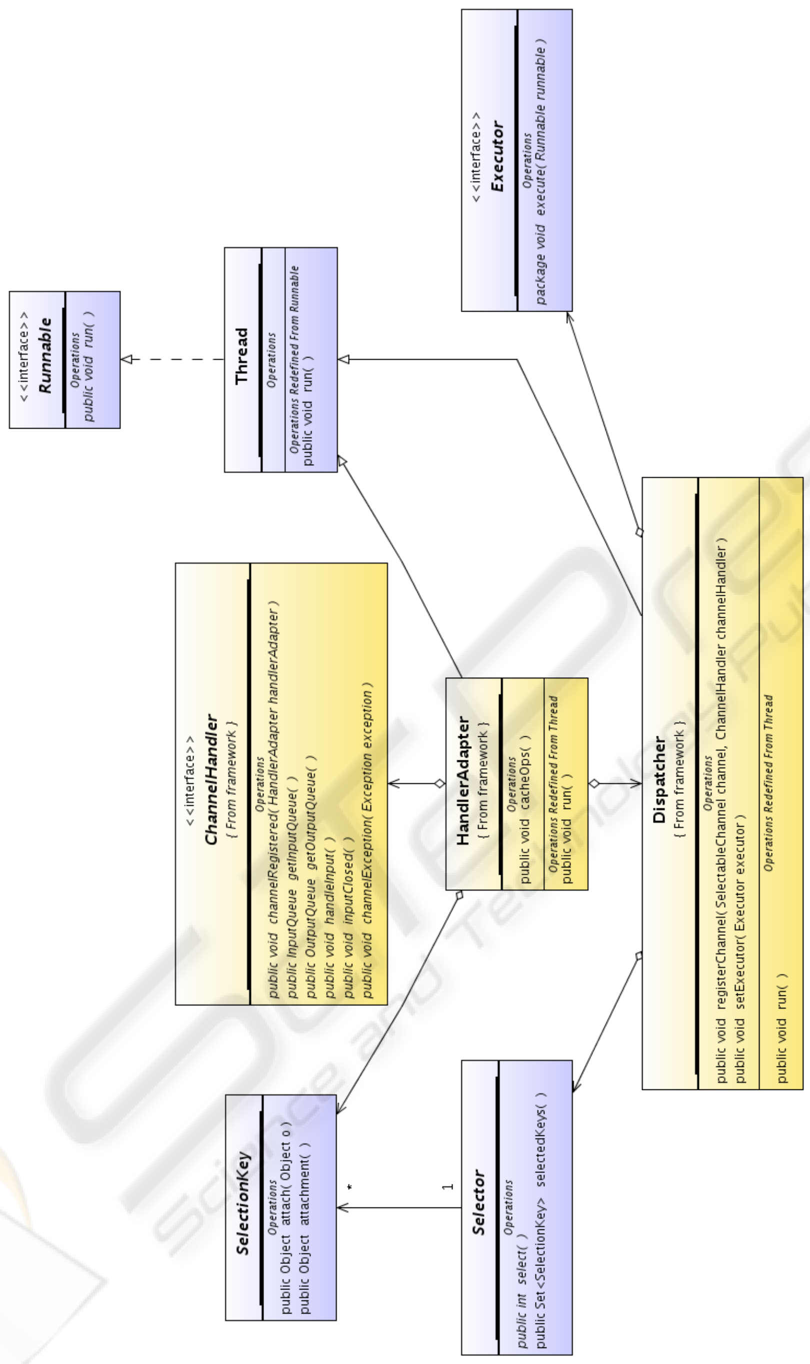
<<interface>>
Runnable
Operations
public void run( )
Thread
Operations
Operations Redefined From Runnable
public void run( )
<<interface>>
Executor
Operations
package void execute(Runnable runnable)
<<interface>>
ChannelHandler
{ From framework }
Operations
public void channelRegistered(HandlerAdapter handlerAdapter)
public InputQueue getInputQueue( )
public OutputQueue getOutputQueue( )
public void handleInput( )
public void inputClosed( )
public void channelException(Exception exception)
HandlerAdapter
{ From framework }
Operations
public void cacheOps( )
Operations Redefined From Thread
public void run( )
Dispatcher
{ From framework }
Operations
public void registerChannel(SelectableChannel channel, ChannelHandler channelHandler)
public void setExecutor(Executor executor)
Operations Redefined From Thread
public void run( )
SelectionKey
Operations
public Object attach(Object o)
public Object attachment( )
Selector
Operations
public int select( )
public Set<SelectionKey> selectedKeys( )
1
*

Figure 2: NIO Framework Core.

3.2.1 Execution

The execution of all HandlerAdapters is off-loaded from the Dispatcher thread to an Executor. Because I/O operations are typically short-lived asynchronous tasks, the default Executor of the Java NIO Framework uses a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. Threads that have not been used for a while are terminated and removed from the pool. Therefore, if the Executor remains idle for long enough, it will not consume any resources.

Not every I/O operation meets the typical crite-

Table 1: Mappings from reactor design pattern to the Java NIO Framework.

Reactor Design Pattern	Java NIO Framework
Dispatcher	Dispatcher
Demultiplexer	Selector
Handle	SelectionKey
EventHandler	HandlerAdapter ChannelHandler Executor
ConcreteEventHandler	n.a.

ria, e.g. SSL operations are comparatively long-lived. If the actual requirements (e.g. a certain thread usage or scheduling) are not met by the default Java NIO Framework Executor, it can be customized with the method `setExecutor()` of the Dispatcher. Because this method is thread-safe, the Executor can even be hot-swapped at runtime.

3.2.2 Selection

There is only *one* Dispatcher running per default in the Java NIO Framework, waiting until new events occur on channels represented by SelectionKeys. If the Dispatcher would ever become the bottleneck of the framework it could simply be parallelized by starting several Dispatcher instances.

Load-balancing could be done by distributing channel registrations between the parallel Dispatcher instances. Some of the most simple scheduling algorithms that could be applied are round-robin distribution or random scheduling.

If connection lifetimes have a high degree of variation, both algorithms could lead to a very unequal distribution of channels to Dispatchers. To prevent this scenario, an *active channel* counter could be integrated into every Distributor and a *lowest-channel-counter-first* scheduling algorithm could be used.

If connections have a high degree of "activity" variation, i.e. on some channels there is always something to read or write and other channels are mostly idle, the scheduling algorithm should be based on a `select()`-counter in the Dispatcher.

3.2.3 Accepting

Another thread, the *Acceptor*, is running on server type applications. It is listening on a server socket for incoming connection requests from clients over the network. Every time a request comes in, the Acceptor creates a new channel and appropriate handler, and registers them both at the Dispatcher of the server type application (or Dispatchers, if selection was parallelized like mentioned in Section 3.2.2).

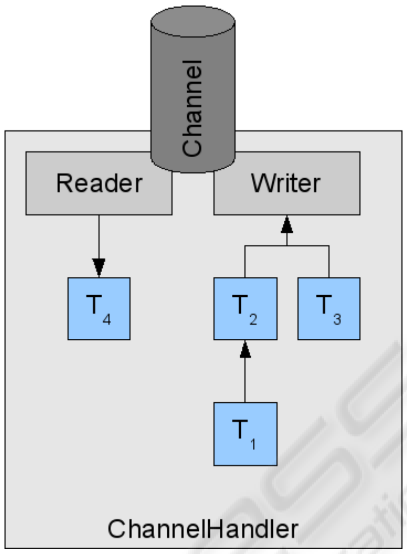

Figure 3: I/O Transformation example.

Currently the Java NIO Framework does not support parallelization of Acceptors.

3.3 I/O Transformations

When application data units (objects, messages, etc.) have to be transmitted over a TCP network connection, they have to be transformed into a serialized representation of bytes.

There are many ways to represent application data and there are also many ways to serialize data into a byte stream. Therefore, there are countless transformations between application space and network space imaginable.

The first approach to this problem in the NIO Framework was to provide an extensible hierarchy of classes, where every class dealt with a certain transformation (e.g. string serialization, SSL encryption). This architecture turned out to be very simple and efficient. The downside of this approach was that every combination of transformations required its own implementing class. Changing the order or composition of transformations was very difficult and much too inflexible for a generic framework.

The second and current approach to message transformation is to implement a set of transformer classes were each class offers just a certain transformation. An application programmer can put these transformer classes together into a hierarchy of almost arbitrary order. Almost no programming effort is required besides assembling the needed classes of the transformation hierarchy in the desired order.

A diagrammatic example of the I/O transformation architecture is shown in Figure 3:

The shapes T_x are the transformation classes. When writing to a channel, the ChannelHandler hands the application level data units to one of the input transformation classes T_1, T_2 or T_3 (depending on the type of input it just accepted). Every transformation class transforms the data and hands it over to its next transformer until it reaches the Writer, which writes the final byte stream to the channel and handles many channel specific problems, e.g. incomplete write operations.

When reading from a channel, the Reader handles the channel specific problems, e.g. connection closing and read buffer reallocations. After reading a byte stream from the Channel, the Reader passes the data to T_4, which transforms the data. The ChannelHandler can get the application level messages from T_4.

There are four basic I/O models for the transformation classes T_x. In ascending order of complexity they are:

- **1:1** (one type of input, one type of output)
- **1:N** (one type of input, different types of output)
- **N:1** (different types of input, one kind of output)
- **N:M** (different types of input, different types of output)

Every model is valid insofar as one can establish a fully functional transformation hierarchy with any of these I/O models. While the 1:1 model would be the most simple one, transformation classes of the N:M model would have the highest flexibility. The interesting thing to note here is that with respect to flexibility every transformation class of the more complex models can be replaced by chaining several transformation classes of the 1:1 model. While trying to implement prototypes for all models above it became clear that the most simple API was provided by using Java Generics and the 1:1 model. Another advantage of the 1:1 model is the encouragement of code reuse, because every transformation should be implemented in a separate class.

The elegance and simplicity comes at the small price of an almost immeasurable performance loss. Currently, Java Generics are implemented by type erasure: generic type information is present only at compile time, after which it is erased by the compiler. The compiler automatically inserts cast operations into the byte code at necessary places which may cause a tiny performance loss. Using the 1:1 model results in slightly longer transformation chains, more involved objects and more locking and unlocking when passing data through a transformation hierarchy.

4 CONCLUSIONS

We presented in this paper a framework for secure high-performance Java network applications that builds upon the NIO library. The framework combines the ease of use of classical I/O operations with the performance gain of NIO, hiding the inconvenient aspects of NIO from the developer. Developing the NIO framework was motivated by research on an anonymity server, a system that requires high-performance network operations over secure channels. First experiments with using the NIO framework within this project show a tremendous performance increase, making the system meet the requirements for anonymity servers in productive environments.

REFERENCES

Arcand, J.-F. (2006). Project Grizzly.

Hitchens, R. (2002). *Java NIO*. O'Reilly & Associates, Inc.

Hitchens, R. (2006). How to build a scalable multiplexed server with NIO. JavaOne Conference.

James O. Coplien, D. C. S. (1995). *Pattern Languages of Program Design*. Addison-Wesley.

Lee, T. (2006). Apache MINA project.

Peierls, T., Goetz, B., Bloch, J., Bowbeer, J., Lea, D., and Holmes, D. (2005). *Java Concurrency in Practice*. Addison-Wesley Professional.

Pitt, E. (2005). *Fundamental Networking in Java*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Roth, G. (2006). xSocket.

Shetty, A. (2006). QuickServer.