A framework for development of concurrency and I/O in servers
Gautier Loyauté

To cite this version:
Gautier Loyauté. A framework for development of concurrency and I/O in servers. 1st European Conference on Systems (EuroSys 2006), Apr 2006, Belgium. 1pp., 2006. hal-00620056
Development of concurrency and I/O in servers andmiddlewares becomes more and more complex:
- minimization of latency;
- maximization of bandwidth;
- no consensus on the best concurrency model;
- select the model best adapted to the hardware.

Applications are modeled by a directed graph, in which each stage (or vertex) corresponds to anatomic unit of treatment and edges correspond tochannels (method calls, local queues or sockets) between them.

We describe here the implementation of a simple “Echo” server which uses three stages. The directed graph models the interconnection of its stages:

![Directed graph diagram]

Specifications and code generation are 100% Java! This ensures the portability of the applications developed using our framework.

**Development process**

This table summarizes the development steps of our framework:

<table>
<thead>
<tr>
<th>Event</th>
<th>Input / Output interfaces</th>
<th>Functional code of a stage</th>
<th>Technical code of a stage</th>
<th>Concurrency selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input/Output interfaces</td>
<td>specified in Java by user</td>
<td>generated from interface</td>
<td>specified in Java by user</td>
<td>generated from concurrency</td>
</tr>
</tbody>
</table>

**Stage description**

The developer should implement the `handle(...)` method which corresponds to the instructions carried out by a stage. Its parameters are the input and/or output events and the context.

**Example:**

```java
public interface AcceptStage {
    public void handle(StageContext ctx, InputAcceptEvent in) {
        SaburoServerSocket client = server.accept();
        InputReadEvent in = new InputReadEvent(client);
        out.setReadByteBuffer(buffer);
        ctx.dispatchToSuccessor(out);
    }
}
```

The context is the way to reach successor(s) in the graph.

**Context:**

If there is only one process, the context is a function call.

**Communication generation**

The interfaces previously defined of the input and/or output events which allow the communication between stages are automatically generated.

**Example:**

```java
public interface ReadStage {
    public void handle(StageContext ctx, InputReadEvent in, OutputHeadEvent out) {
        SaburoSocket client = in.getServerSocket();
        ReadStage stage = new ReadStage();
        out = stage.handle(ctx);
    }
}
```

The implementation of the context is also automatically generated according to the concurrency model.

**Concurrency generation**

The last step consists in the automatic generation of the concurrency model.

**Example:** Iterative architecture

```java
public class IterativeModel {
    public void service() throws Exception {
        new Thread(new Runnable() {
            public void run() {
                acceptSelector.doSelect();
                while (true) {
                    acceptSelector.doSelect();
                    writeSelector.doSelect();
                    while (true) {
                        acceptSelector.doSelect();
                    }
                }
            }
        }).start();
    }
}
```

**Example:** Staged Event-Driven Architecture

```java
public class SedaModel {
    public void service() throws Exception {
        new Thread(new Runnable() {
            public void run() {
                while (true) {
                    writeSelector.doSelect();
                }
            }
        }).start();
    }
}
```

The bytecode is generated automatically using ASM and all the code generators can be used at runtime, even if they are usually used at compile time.

**Event description**

The developer has to define the interface for input and/or output events for each stage. These events allow the communication between stages.

**Example:**

For the initial stage, only an output interface is defined:

```java
public interface OutputAcceptEvent {
    public void setAcceptSaburoSocket(SaburoSocket s);
}
```

For a final stage only an input interface is defined:

```java
public interface InputWriteEvent {
    public SaburoSocket getAcceptSaburoSocket();
    public void getReadByteBuffer();
}
```

For any other stage input and output interfaces should be defined:

```java
public interface InputReadEvent {
    public SaburoSocket getAcceptSaburoSocket();
}
```

```java
public interface OutputHeadEvent {
    public void setReadByteBuffer(ByteBuffer b);
}
```

**Stage connections**

The connection of the stages has to be specified inJava by the developer.

**Example:**

```java
StageManagerImpl manager = new StageManagerImpl();
manager.connect(AcceptStage.class, ReadStage.class);
manager.connect(ReadStage.class, WriteStage.class);
```

**Concurrency selection**

The concurrency model has to be selected in Java by the developer.

**Example:**

```java
ModelExecutorImpl executor = new ModelExecutorImpl();
executor.run(configurator, stageManager, SEDA);
```

Currently, these two steps are hand-coded but could be generated automatically via an Eclipse plugin.