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

HAL Id: hal-00620056
https://hal-upec-upem.archives-ouvertes.fr/hal-00620056
Submitted on 7 Sep 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Development of concurrency and I/O in servers and middlewares 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 an atomic unit of treatment, and edges correspond to channels (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:

```
accept -> read -> write
```

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

### 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.

```
public interface OutputAcceptEvent { public void accept(SaburoServerSocket server); }
```

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

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

For a final stage only an input interface is defined:

```
public interface InputWriteEvent { public void setWriteByteBuffer(ByteBuffer buffer); }
```

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

```
public interface InputReadEvent { public SaburoServerSocket getAcceptSaburoSocket(); }
```

```
public interface OutputWriteEvent { public void setWriteByteBuffer(ByteBuffer buffer); }
```

```
public interface InputWriteEvent { public ByteBuffer getReadByteBuffer(); }
```

### Concurrency selection

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

```
public class NodeExecutorImpl { public ModelExecutorImpl executor = new ModelExecutorImpl(); public void run(ExecutorConfigurator configurator, stageManager, SEDA); }
```

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

### 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.

```
public class StageManagerImpl { public void handle(InputWriteEvent in, OutputReadEvent out) { client.write(in.getWriteByteBuffer()); ctx.dispatchToSuccessor(out); } }
```

### Communication generation

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

```
public interface Context { void start(); }
```

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

In the case of several processes, we introduce queues to implement the context. For distributed applications, the context establishes the connections between peers.

### Concurrency generation

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

```
public class Echo { public void service() throws Exception { while (true) { acceptStageWrapper.handle(); } } }
```

Example: Iterative architecture

```
public class IterativeModel { public void service() throws Exception { while (true) { acceptStageWrapper.handle(); } } }
```

Example: Staged Event-Driven Architecture

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

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

```
public class IterativeModel { public void service() throws Exception { while (true) { acceptStageWrapper.handle(); } } }
```

Example: For distributed applications, the context establishes the connections between peers.