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.

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

Gautier Loyauté
Université de Marne la Vallée
Laboratoire d’Informatique de l’Istitut Gaspard-Monge
UMR-CNRS 8049
F-77454 Marne la Vallée, Cedex 2, France
e-mail: loyaute@univ-mlv.fr

Example:
```
public interface AcceptStage {
    public void handle(StageContext ctx, OutputAcceptEvent out) {
        SaburoSocket client = server.accept();
        out.setAcceptSaburoSocket(client);
        ctx.dispatchToSuccessor(out);
    }
}
```

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.

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

Example:
```
public class AcceptStage {
    public void handle(StageContext ctx, OutputAcceptEvent out) {
        SaburoSocket client = server.accept();
        out.setAcceptSaburoSocket(client);
        ctx.dispatchToSuccessor(out);
    }
}
```

Stage connections
The connection of the stages has to be specified in Java by the developer.

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

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.

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 interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Events</td>
<td>specified in Java by user</td>
<td>generated from interface</td>
</tr>
<tr>
<td>Functional code of a stage</td>
<td>specified in Java by user</td>
<td>generated from interface</td>
</tr>
<tr>
<td>Technical code of a stage</td>
<td>generated from concurrency</td>
<td></td>
</tr>
<tr>
<td>Concurrency selection</td>
<td>specified by user</td>
<td>generated from concurrency</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.

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

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 getReadByteBuffer(ByteBuffer b);
}
```

For any other stage input and output interfaces should be defined:
```
public interface InputReadEvent {
    public void setAcceptSaburoSocket(SaburoSocket s);
    public void setReadByteBuffer(ByteBuffer b);
    public void getReadByteBuffer(ByteBuffer b);
}
```

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

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

Example: Iterative architecture
```
public class IterativeModel {
    public void service() throws Exception {
        new Thread(new Runnable()) {
            public void run() {
                while (true) {
                    acceptSelector.doSelect();
                }
            }
        }.start();
    }
}
```

Example: Event-Driven Architecture
```
public class EventDrivenModel {
    public void service() throws Exception {
        new Thread(new Runnable()) {
            public void run() {
                while (true) {
                    selector.select();
                }
            }
        }.start();
    }
}
```

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