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

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

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:

![Graph Diagram]

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

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:
```java
public class AcceptStage {
    public void handle(int stageContext ctx, InputStream in, OutputStream out) {
        SaburoServerSocket server = server.getAcceptSaburoSocket();
        in.setAcceptSaburoSocket();
        out.setAcceptSaburoSocket();
        ctx.dispatchSuccessor(out);
    }
}
```

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.

Context:

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

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

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() {
                while (true) {
                    acceptStageWrapper.handle();
                }
            }
        }).start();
    }
}
```

Example: Staged Event-Driven Architecture
```java
public class StagedModel {
    public void service() throws Exception {
        new Thread(new Runnable() {
            public void run() {
                while (true) {
                    writeSelector.doSelect();
                }
            }
        }).start();
    }
}
```

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 accept(SaburoSocket server); }
```

Concurrent selection

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

Example:
```java
new ModelExecutorImpl() {
    executor.run(configurator, stageManager, SEDA);
}
```

For a final stage only an input interface is defined:
```java
public void handle(int stageContext ctx, InputStream in, OutputStream out) {
    SaburoSocket client = server.getAcceptSaburoSocket();
    in.setAcceptSaburoSocket();
    out.setAcceptSaburoSocket();
    ctx.dispatchSuccessor(out);
}
```

Concurrent selection

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.

The implementation is based on the Java NIO API which provides blocking and non blocking I/O. To avoid the complexity of this API, we provide encapsulation classes which simplify implementation.

This ensures the portability of the applications developed using our framework.

Stage connections

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

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