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

Gautier Loyauté

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:

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:

Input / Output interfaces	specified in Java by user
Events	generated from interfaces
Functionnal code of a stage	specified in Java by user
Technical code of a stage	generated from concurrency
Stage connection	specified by user
Concurrency selection	
Concurrency	generated from concurrency

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:

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

For a final stage only an input interface is defined:

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

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

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

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

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 {
  private final SaburoServerSocket server;

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

public class ReadStage {
  public void handle(StageContext ctx,
                     InputReadEvent in,
                     OutputReadEvent out) {
    SaburoSocket client = in.getAcceptSaburoSocket();
    ByteBuffer buffer = null;

    while((buffer = client.read()) != null) {
      buffer.flip();
      out.setReadByteBuffer(buffer);
      ctx.dispatchToSuccessor(out);
    }
  }
}

public class WriteStage {
  public void handle(InputWriteEvent in) {
    SaburoSocket client = in.getAcceptSaburoSocket();
    client.write(in.getReadByteBuffer());
  }
}
```

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 simplifies implementation.

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);
```

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

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.
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**.

Example: Iterative architecture

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

Example: **S**taged **E**vent-**D**riven **A**rchitecture

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

    new Thread(new Runnable() {
      public void run() {
        while(true) {
          readSelector.doSelect();
        }
      }
    }).start();

    while(true) {
      acceptSelector.doSelect();
    }
  }
}
```

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**.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITÉ de Marne-la-Vallée