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

Gautier Loyauté
loyaute@univ-mlv.fr

student: yes

Development of concurrency and I/O in servers or middlewares becomes more and more complex because of the increasing demands for \textit{effectiveness}, \textit{dynamic variability}, \textit{dynamicity} and \textit{scalability} \cite{Welsh}. In this poster, we present Saburo, a Java framework, based on the concepts of \textit{separation of concerns} and \textit{code generation}, conceived to try to address these issues.

We propose to develop an application in linear way (without synchronization) and to distinguish the functional aspects and the concurrency model to be used, abstracting technical aspects from the service offered. To reach this goal, we chose to represent the application as a directed graph, in which each vertex, or stage (in reference to \textit{Staged Event-Driven Architecture} \cite{Welsh}), corresponds to an atomic unit of treatment and edges correspond to the channels between them (method calls, local queues, networks). The vertices are made up of a sequence of instructions ending with a blocking call (I/O, synchronization). For communication, the stages define input and output interfaces which are respectively used for the reception and the emission of events. The communication between the vertices can be direct (which avoids bottlenecks at the level of the application and simplifies communication implementation) or centralized.

By adopting this approach, it is very easy to switch from one concurrent model (single-process event-driven, multi-threaded, . . . ) to another without having to modify the functional part. Thus, only the technical code is changed, generated more exactly, automatically and transparently for the user. Because there is no consensus on the best concurrent model \cite{Welsh}, this method should allow to easily select the model more adapted to underlying architecture.

In Saburo, specifications and code generations are 100\% Java. It insures the portability of the applications developed using our framework.

To illustrate the three steps of the Saburo development process, we describe the implementation of a simple “Echo” server. This server uses three stages, the first one accepts new clients, the second one reads data received on the connection and, finally, the last one writes them back to the client. The directed graph below models the connection of these stages:

\begin{tikzpicture}
    \node[draw, shape=rectangle, rounded corners] (a) at (0,0) {accept};
    \node[draw, shape=rectangle, rounded corners] (b) at (2,0) {read};
    \node[draw, shape=rectangle, rounded corners] (c) at (4,0) {write};
    \draw[->] (a) -- (b);
    \draw[->] (b) -- (c);
\end{tikzpicture}

\textbf{Description of events}

According to the position of a stage in the graph, the developer has to define the interfaces for its input and/or output events.

\textbf{Description of stages}

Once the various interfaces of the events defined, the developer must implement the stages. Each stage contains a \texttt{handle()} method which corresponds to instructions carried out by the stage. Its parameters are input and output events.

\textbf{Connection of stages}

Then, the connection of the various stages has to be specified. It is implemented using an abstract class which is common to all servers whatever is the concurrency model.

\textbf{Generation of the technical code}

Interfaces of input and output events defined previously are then all implemented by the \texttt{DefaultEvent} class. The bytecode of the \texttt{DefaultEvent} class is generated automatically from the list of event interfaces using bytecode generator. To send their output events, the stages with successors use the \texttt{dispatchToSuccessor()} method which is generated automatically according to the concurrency model. The last step consists in generating the configuration of the I/O mode for each stage and the technical part of the server.

This separation between functional part, specified by the developer, and technical part, generated automatically, enforces the independence of the functional part from the underlying platform, and prepares future adaptations and evolutions.

The table below summarizes the various development steps and the way they are obtained.

\begin{table}[h]
\begin{tabular}{|c|c|}
\hline
\textbf{Input / Output interfaces} & specified in Java by user \\
\hline
\textbf{Events} & generated from interfaces \\
\hline
\textbf{Functionnal code of a stage} & specified in Java by user \\
\hline
\textbf{Technical code of a stage} & generated from concurrency \\
\hline
\textbf{Stages’s connection} & specified in Java by user \\
\hline
\textbf{Concurrency} & generated from concurrency \\
\hline
\end{tabular}
\end{table}

\cite{Welsh} M. Welsh. \textit{An Architecture for Highly Concurrent, Well-Conditioned Internet Services}. PhD thesis, Department of Computer Science, University of California at Berkeley, USA, August 2002.