Saburo, a tool for I/O and concurrency management in servers

Gautier Loyauté, Rémi Forax and Gilles Roussel

Institut Gaspard-Monge, Université de Marne-la-Vallée
5, Boulevard Descartes Champs-sur-Marne - 77454 Marne-la-Vallée Cedex 2 - France
{loyaute, forax, roussel}@univ-mlv.fr

Abstract

*This paper presents a Java framework based on **separation of concerns** and **code generation** concepts that facilitates development of concurrency and I/O in servers. In this approach, the application is modeled by a graph whose vertices correspond to units of treatment connected by channels. It allows to build all kind of servers: multi-threaded, **S**ingle-**P**rocess **E**vent-**D**riven, **S**taged **E**vent **D**riven **A**rchitecture, etc. without modification of the functional part. This architecture also permits to extend very easily an application, adding vertices and edges to the graph. The aim of our development tool is to improve programmer productivity and portability, decreasing development time, and reducing bugs or deadlock problems.*

1 Introduction

Development of concurrency and I/O in servers or middlewares becomes more and more complex because of the increasing demands for **effectiveness** (minimization of latency and maximization of bandwidth [10]), **dynamic variability** (the ability for elements to evolve or change at runtime [9]), **dynamicity** (elements can be added, modified or removed during the execution of an application [17]) and **scalability** [9, 15, 17]. This complexity induces an increasing number of errors such as random behaviors, falls of performance or deadlocks. This is why the development of new tools (languages, concepts, libraries, etc.) that helps the implementation of such softwares is important [10, 13].

In this paper, we present Saburo, a Java framework, based on the concepts of **separation of concerns** and **code generation**, conceived to develop concurrency and I/O parts of servers and middlewares. Our approach tries to answer the performance problems previously described in [10] and offers several other advantages.

Using Saburo to develop servers implies separation between the functional aspects and the concurrency model to be used, abstracting technical aspects from the service offered. By adopting this approach, it is very easy to switch from one model 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 [5]. Because there is no consensus on the best concurrent model [14, 1, 17], this method should allow to easily select the model more adapted to underlying architecture (cluster, multi-core, etc.).

Several works suggesting the use of concurrency by **cooperation** (the exchange of information between processes in order to handle a particular task), rather than by **competition** (only one process is selected to entirely handle a request), reveal an implementation less intuitive and more complex [7, 3], and, for the other model, difficulties to manage synchronization [7]. This is why, in Saburo we propose to develop an application in linear way (without synchronization) and abstracts the concurrency model to be used. To reach this goal, we chose to represent the application as a directed graph, in which each vertex corresponds to an atomic unit of treatment (for example, reading on a socket) and the edges reflect the channels between them (function calls, local queues, networks). This modeling also allows to extend very quickly and easily an application adding vertices and edges to the graph.

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

All these features should simplify the specification of the applications, decreasing development time and reducing bugs and security problems.

In the rest of this article, we present the existing concurrency models [9] and, then, some related works. Lastly, we detail Saburo, our component model and its use in the development of a simple “Echo” server.

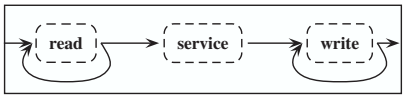
1-4244-0054-6/06/$20.00 ©2006 IEEE

2 Concurrency models

To minimize latency and to maximize throughput, a server must interlace the handling of several requests. Indeed, it is then possible to overlap the disc, network and processor(s) activities induced by the different requests. The strategy used by the server is determined by its **concurrency model**. We present here those taken into account by Saburo.

Sequential

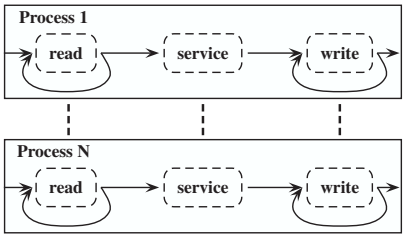
The sequential architecture uses a single process to handle several requests and carries out all the steps required by a request before accepting a new one. It respects the policy *first in, first out* (the network is used as a request buffer).



This model does not exhibit real concurrency. It is very easy to develop and may be used for servers sporadically accessed.

Multi-process

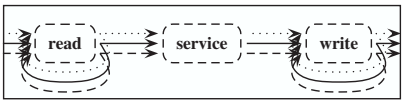
The multi-process architecture assigns a process to each request. Each process carries out all steps necessary to treat the request before accepting another one. Since several processes may be used, several requests are served at the "same time" on the server.



Moreover, the management of the overlapping between disc activity, processor, and network access is carried out, in a transparent way for the programmer, by the underlying operating system. It manages the concurrent accesses to these resources. In this model, the processes have their own memory space that usually avoids the use of the synchronization primitives. It also implies a containment of the processes and thus greater safety, but it induces difficulties to share information. Moreover, process creations induce time overhead for each request.

Multi-threaded

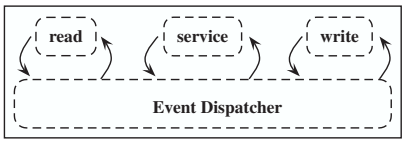
Like the multi-process architecture, in the multi-threaded architecture a thread is in charged of one request before accepting a new one.



Contrarily to processes, threads share their address spaces and it is very easy pass information through global variables. However, synchronization mechanisms are required to control their accesses. These synchronizations involve time overhead, greater difficulty of programming [7] and safety problems. Moreover, the cost of thread creation, although less important than process one, remains. It induces an important fall of performances when the number of threads reaches a particular threshold [17].

Single-Process Event-Driven

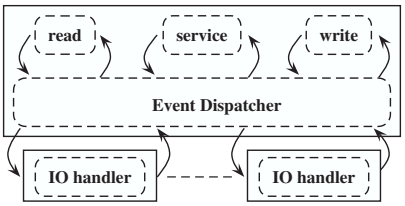
The SPED architecture uses only one process to handle the requests. Contrary to the sequential architecture, the server use non blocking calls to carry out asynchronous I/O. A selection mechanism allows to know what operations can be performed without blocking.



In fact, this architecture corresponds to a finite state machine which performs the instructions associated with the state in which the request is. State changes when the I/O in progress finishes. Thanks to non blocking and asynchronous I/O there is an interlacing between multiple requests. In addition to its difficulty of implementation, the main problem of this architecture is the lack of reliable support for asynchronous disc operations in most operating systems [15].

Asymmetric Multi-Process Event-Driven

The AMPED model answers the problem of disc I/O which are not always asynchronous by combining the *SPED* architecture with threads in charge of carrying out the blocking I/O.



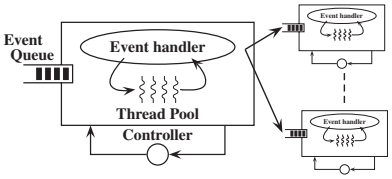
The main process performs all instructions except disc I/O and delegates them to dedicated processes. Using this architecture, one preserves the effectiveness of SPED while avoiding the problem of blocking discs I/O. The number of processes involved in I/O being modest with respect to the number of potentially managed requests, the

scheduling, cache error and context switching overhead remain negligible.

Staged Event Driven Architecture

SEDA performs the steps necessary to handle a request as a finite state machine.

A state corresponds to a component, called **stage**, that represents a fundamental unit of treatment. The transitions are queues or networks. A stage is composed of an input events queue, a thread pool and an event handler.



The handler represents the functional part of the stage, i.e. the instructions performed given an input event. It handles each event found in the input queue and produces new events as output that are pushed into the queues of its successors. Moreover, each stage is managed by a controller in charge of adapting the resources according to statically specified policies. Like AMPED, SEDA uses a thread pool to simulate non blocking disc operations.

3 Related Work

Several frameworks and libraries may be used for rapid implementation of servers and middleware, but none of them uses a graph model, separation of concerns and code generation, all together.

Tuxedo [2] is a commercial library that supports four different communication methods (request / response, message queuing, publish / subscribre and conversational). These functions can be used in Java or C++. The library supports event-driven and multi-threaded applications on several platforms but it is difficult to finely control the program.

Twisted [12] is a Python framework that uses an event-driven model. Unlike many other tools mentioned here, it allows access to the underlying platform if a developer wishes more control over his application.

SEDA [17] is a Java framework for developing event-driven servers. In this framework an application is represented as a directed graph where various stages are connected by queues. The concurrent model is fixed.

Serveez [11] is a library written in C that provides many functionalities necessary to quickly write servers. One of its main goals is portability. The concept of portability for the highly concurrent applications is very important, but suggests the use of languages such as Java and Python.

JAWS [10, 9] is a C++ framework allowing to write concurrent applications. It provides design patterns and recurring solutions to increase flexibility, reusability and modularity of applications. It is used in many projects such as embedded systems or real time ones, illustrating its robustness and effectiveness.

MSPL is a specification language [6] used to easily describe Internet protocols. A compiler is used to generate the low-level implantation of the communication for both the client and server side from a declarative description of the protocol. One of the drawbacks of this approach is that it does not give access to the underlying implementation language.

NEST [18] is a specification language approach close to Lex and Yacc allowing to generate the server code automatically from a grammatical description of the protocol. One of the principal characteristics of this approach is the possibility of generating three types of servers (multi-process, multi-threads and event-driven).

4 Description of Saburo

In Saburo, applications are modeled by a directed graph, in which each vertex, or stage (in reference to SEDA), corresponds to an atomic unit of treatment and edges correspond to the channels between them. The vertices consist of a sequence of instructions ending with a blocking call (I/O, synchronization). The edges may correspond to method calls, local queues or sockets. For communication, the stages define input and output interfaces which are respectively used for the reception and the emission of events. More exactly, there are three types of vertices, the **initial** ones define only an output interface, the **final** ones require only an input interface and the **unspecified** ones define at the same time an input and output interface. The communication between the vertices can be direct, which avoids bottlenecks at the level of the application and simplifies communication implementation, or centralized. It corresponds to the two modes of communication in the event-driven systems, explicit (the vertices are directly connected) or implicit (we use a centralized vertex to connect the others).

The graph can be seen as a finite state machine. The construction of this graph relies on the enumeration of all synchronization and I/O operations that the application must carry out to provide a service. This concept of **blocking graph** was already used in [3] for thread scheduling at runtime, but it has never been used to modularize a concurrent application to facilitate its development, as we do in Saburo.

This modeling, thanks to its local aspect, allows to (re)configure very quickly, and possibly dynamically, an application by simple addition of vertex or edge in the graph.

5 Implementation

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:

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.

For the *accept* stage, which is the initial vertex, an interface is only defined for output events as:

```
public interface OutputAcceptEvent extends Event {
 public void sendAcceptSaburoSocket(SaburoSocket s);
}
```

The *read* stage, which is an unspecified vertex, must define an input and output interface.

```
public interface InputReadEvent extends Event {
 public SaburoSocket receiveAcceptSaburoSocket();
}

public interface OutputReadEvent extends Event {
 public void sendReadSaburoByteBuffer(SaburoByteBuffer b);
}
```

Lastly, the *write* stage, which is a final vertex, only defines an input interface, as follows:

```
public interface InputWriteEvent extends Event {
 public SaburoSocket receiveAcceptSaburoSocket();
 public SaburoByteBuffer receiveReadSaburoByteBuffer();
}
```

Description of stages

Once the various interfaces of the events defined, the developer must implement the stages. Each stage contains a *handle()* method which corresponds to intructions carried out by the stage. Its parameters are input and output events. This is used to connect the stages and allows to check the reachability of all the vertices of the graph.

The *accept* stage is implemented by:

```
public class AcceptStage extends AbstractSourceStage {
 public void handle(OutputAcceptEvent o) {
  SaburoSocket s = server.accept();
  o.sendAcceptSaburoSocket(s);
  dispatchToSuccessor(o);
 }
}
```

The *read* stage may be:

```
public class ReadStage extends AbstractQueueStage {
 public void handle(InputReadEvent i,
                    OutputReadEvent o) {
  SaburoSocket c = i.receiveAcceptSaburoSocket();
  SaburoByteBuffer b = c.read();
  o.sendSaburoByteBuffer(b);
  dispatchToSuccessor(o);
 }
}
```

Lastly, the *write* stage can be implemented by:

```
public class WriteStage extends AbstractSinkStage {
 public void handle(InputWriteEvent i) {
   SaburoSocket c = i.receiveAcceptSaburoSocket();
   c.write(i.receiveReadSaburoByteBuffer());
 }
}
```

The Saburo's implementation is based on the NIO API [16] which provides blocking and non blocking I/O. Because the use of this API is complex, we provide encapsulation classes which simplifies implementation in our context. Particularly, complete writing of data in non-blocking mode is managed by this layer.

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, as follows:

```
public abstract class AbstractEchoService
  extends AbstractService {
 public void connect()  {
   AcceptStage accept = new AcceptStage();
   ReadStage read = new ReadStage();
   WriteStage write = new WriteStage();
   // Connection of stages
   accept.setSuccessor(read);
   read.setSuccessor(write);
   // Add stages in service
   addStage(accept);
   addStage(read);
   addStage(write);
 }
}
```

Currently, this step is hand-coded but should be generated automatically via an Eclipse plugin.

Generation of the technical code

Interfaces of input and output events defined previously are then all implemented by the *DefaultEvent* class. Then, the application uses a single event object through all stages and can reuse it efficiently. The bytecode of the *DefaultEvent* class is generated automatically form the list of interfaces by the *event generator* using ASM [4].

The following source code has been obtained decompiling the Java bytecode generated by event generator from the interfaces previously defined.

```
public class DefaultEvent implements OutputAcceptEvent
     InputReadEvent,OutputReadEvent,InputWriteEvent{
 private SaburoSocket acceptSaburoSocket;
 public SaburoSocket receiveAcceptSaburoSocket() {
  return acceptSaburoSocket;
 }

 public void sendAcceptSaburoSocket(SaburoSocket s){
  this.acceptSaburoSocket = s;
 }
```

```
  private SaburoByteBuffer readSaburoByteBuffer;
  public SaburoByteBuffer receiveReadSaburoByteBuffer(){
   return readSaburoByteBuffer;
  }

  public void sendReadSaburoByteBuffer(SaburoByteBuffer b){
   this.readSaburoByteBuffer = b;
  }
}
```

The stages with successors use the *dispatchToSuccessor()* method to send their output events, while the stages with predecessors define a *push()* method allowing them to receive input events.

The implementation of the *push()* method is generated automatically according to the concurrency model. Indeed, if only one process is used to handle a request (competition), the method is only a function call. On the other hand, if several processes are used (cooperation), it is necessary to introduce queues. Finally for distributed applications, this method will establish the connections between peers.

Given the functional code above, we illustrate the implementation of a *multi-threads* server and of a *single-process event-driven* server. The differences between the various concurrency models are visible in these two examples. This last step consists in generating the configuration of the I/O mode for each stage and the technical part of the server using one of the *server generator*.

The *multi-thread* server accepts new client and instantiates processes for each of them; the I/O are in blocking mode. The *accept* and *read* stages are not connected explicitly because they are performed in the same thread (indicated by the *pushToSameThread(false)* function).

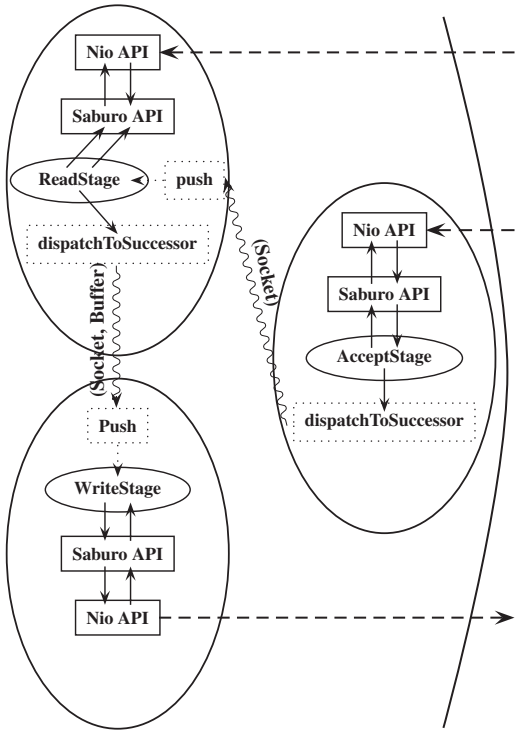
```
public class MultiThreadEchoService
  extends AbstractEchoService {
 public void service() throws ServiceException {
   accept.pushToSameThread(false);
   // Configure IO mode
   accept.configureIOBlocking(true);
   read.configureIOBlocking(true);
   write.configureIOBlocking(true);
   while(true) {
     final Event acceptOutput = new DefaultEvent();
     final Event readOutput = new DefaultEvent();
     accept.handle(acceptOutput);
     new Thread(new Runnable() {
          public void run() {
            read.push(acceptOutput, readOutput);
          }}).start();
   }
 }
}
```

In the *single-process event-driven* server, all the stages are connected and the main process performs the selection of an I/O event at each iteration; the I/O are in non blocking mode.

```
public class SpedService
  extends AbstractEchoService {
 public void service() throws ServiceException {
   // Configure IO mode
   accept.configureIOBlocking(false);
   read.configureIOBlocking(false);
   write.configureIOBlocking(false);
   while(true)
     doSelect();
 }
}
```

In order to only focus the developer on the functional part, these last two classes have to be generated automatically. 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 figure below summarizes the "Echo" server. It details the connections between stages (zigzag), inputs / outputs events for each stage (label on zigzag) and the different parts of the implementation : generated (dot lines), hand-coded (ovals) and libraries (frames).



The table below summarizes the various development steps and the way they are obtained (specification or bytecode generation using ASM [4]).

Input / Output interfaces	specified in Java
Events	generated from interfaces
Functionnal code of a stage	specified in Java
Technical code of a stage	generated from concurrency
Stages's connection	specified in Java
Concurrency	generated from concurrency

All code generators presented here can be used dynamically, even if they are usually used statically.

6 Conclusion and perspectives

In this paper, we present Saburo, a tool that simplifies the development of servers or middlewares. It allows to switch, with minimum overhead, between different concurrent models (*SPED*, *AMPED*, *SEDA*, etc.). Its development model induces a weak interlacing between the functional code and the concurrent model introducing an high level specification of the application (the technical code is generated automatically). It should improve programmer productivity since he only focuses on the functional aspect of the software. Saburo also allows to extend very quickly applications by simple addition of vertices or edges in a graph.

Although this work is still under progress, Saburo presents many advantages compared to other existing tools. One of the perspectives of this work consists in applying model checking tools, such as SPIN [8], to automatically detect, thanks to the modeling of the application as a finite state machine, deadlock, unreachable states, or temporal properties of the application.

References

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative Task Management without Manual Stack Management. In *Proceedings of the USENIX Annual Technical Conference*, pages 289 – 302, Monterey, CA, USA, June 2002.

[2] BEA. Tuxedo. http://www.bea.com. White papers.

[3] R. V. Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio : Scalable Threads for Internet Services. In *Nineteenth ACM Symposium on Operating Systems Principles*, pages 268 – 281, Bolton Landing, NY, USA, Oct. 2003. ACM Press.

[4] E. Bruneton, R. Lenglet, and T. Coupaye. ASM : A Code Manipulation Tool for the Construction of Adaptable Systems. In *Adaptable and Extensible Component Systems*, Grenoble, France, Nov. 2002. ACM Press.

[5] K. Czarnecki and U. W. Eisenecker. *Generative Programming, Methods, Tools and Applications*. Addison Wesley Professional, June 2000.

[6] M. A. L. Douglas and P. K. Chan. A Protocol Language Approach to Generating Client-Server Software. Technical report, Florida Institute of Technology, Melbourne, Florida, USA, 2000.

[7] D. Gupta and R. Jaiswal. Threads vs. Events. Report for CSE221, University of California, San Diego, USA, Dec. 2003.

[8] G. J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison Wesley Professional, Sept. 2003.

[9] J. Hu and D. C. Schmidt. *Domain-Specific Application Frameworks : Frameworks Experience by Industry*, chapter JAWS : A Framework for High-Performance Web Servers, pages 339 – 376. John Wiley and Sons Ltd, 1999.

[10] J. C. Hu and D. C. Schmidt. Developing Flexible and High-Performance Web Servers with Frameworks and Patterns. *ACM Computing Surveys*, 32(1):39 – 45, Mar. 2000.

[11] S. Jahn. Serveez documentation. `http://www.gnu.org/software/serveez/manual/index.html`.

[12] G. Lefkowitz and I. Shtull-Trauring. Network Programming for the Rest of Us. In *Proceedings of the USENIX Annual Technical Conference*, pages 77 – 89, San Antonio, USA, June 2003.

[13] R. Lenglet. *Composition Flexible et Efficace de Transformations de Programmes*. PhD thesis, Institut National Polytechnique de Grenoble, Nov. 2004.

[14] J. Ousterhout. Why Threads Are a Bad Idea (for most purposes). In *Proceedings of the USENIX Annual Technical Conference*, San Diego, CA, USA, Jan. 1996. Invited talk.

[15] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash : An Efficient and Portable Web Server. In *Proceedings of the USENIX Annual Technical Conference*, pages 199 – 212, Monterey, CA, USA, June 1999.

[16] Sun Microsystems. New I/O APIs, 2002. http://java.sun.com/j2se/1.4.2/docs/guide/nio/.

[17] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA : An Architecture for Well-Conditioned, Scalable Internet Services. In *Eighteenth Symposium on Operating Systems Principles*, pages 230 – 243, Chateau Lake Louise, Canada, Oct. 2001. ACM Press.

[18] K. Wilson and J. Aycok. NEST : NEtwork Server Tool. Technical Report TR-2004-746-11, The University of Calgary, Apr. 2004.