ERLANG/OTP FRAMEWORK FOR COMPLEX MANAGEMENT APPLICATIONS DEVELOPMENT

Carlos Abalde, Víctor M. Gulías, Laura M. Castro, Carlos Varela and J. Santiago Jorge
MADS Group, Department of Computer Science, University of A Coruña, A Coruña, Spain
cabalde@udc.es, gulias@udc.es, lcastro@udc.es, carlosvarela@udc.es, sjorge@udc.es

Keywords: Distributed systems, functional languages, design patterns, cluster computing, client/server architecture.

Abstract: This paper presents our ideas about how to evolve a first and early architecture, which has been used in the development of a large scale management client/server application, to a more sophisticated one, which could be reused in other developments, significantly reducing the development life cycles of future versions. We also expose some problems that arised in the process and suggest some solutions.

1 INTRODUCTION

In this paper, we will elaborate some ideas explaining how to evolve the first and early architecture used in the development of a large scale management client/server application, using the distributed functional language Erlang (Armstrong et al., 1996), to a more sophisticated one. The application (*ARMISTICE* (Armistice, 2002; Cabrero et al., 2003; Gulías et al., 2005; Gulías et al., 2006), *Advanced Risk Management Information System: Tracking Insurances, Claims and Exposures*) is a risk management information system (RMIS) designed for a large regional company. The aim of the proposed architecture is to help to reduce the development life cycles, and to be reused in similar projects.

The paper is structured as follows. Section 2 gives an overview of the proposed architecture. Next, some discussion is provided on section 3, explaining how the framework would improve the development, taking our study case as example. Finally, section 4 shows some conclusions and future work to be done.

2 FRAMEWORK OVERVIEW

Our proposal is a layered client/server architecture, based on two well-known architectural patterns: Layers and Model-View-Controller (Gamma et al., 1999).

The *user side* is a lightweight client which only makes remote procedure calls (RPCs), and has no associated logic (i.e., there are no business objects in the user side). The *server side* supports the model and the business logic, and is structured in four tiers:

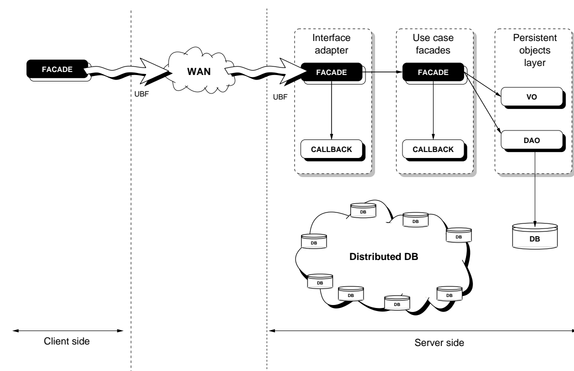

Figure 1: Framework architecture overview.

- **Interface adapter**: Receives messages from clients, deserialises their input and parameters, and gives them the appropriate format so that the server can process the enquiry, and viceversa.
- **Use case facades**: Facades of the system, which represent the access points to each subsystem. Every facade has methods implementing different use cases, using the persistent objects layer.

- **Persistent objects layer**: Here, domain transfer objects (TOs) and data access objects (DAOs) (Marinescu, 2002) are defined. Each TO models a domain entity, and the DAO is the module that interacts with the persistence layer.
- **Persistence**: It represents the permanent storage over a relational database.

Source code for implementing these layers can be automatically generated from a description of:

- The client-server communication format.
- The use cases.
- The domain transfer objects and relationships.

2.1 Client-Server Communication

The transport layer needs to be lightweight and easily integrable with different client applications. There are several possibilities to solve this problem. For example, interface definition languages like XDR (XDR, 2003) or ASN.1 (ASN.1, 2003) or complex frameworks like CORBA (Corba, 2003), most of them supported by Erlang/OTP.

The UBF scheme (Universal Binary Format (Armstrong, 2002)) has the expressive power of the XML set of standards, but it is considerably simpler. Thus, in our framework, we can see the server as a service whose interface is defined in XML as follows,

```
<method name="add_customer" cache="0">
  <input>UBF description</input>
  <output>UBF description</output>
</method>
```

Then, our compiler gets each facade interface definition and builds the following skeletons:

- `customer.erl`, which is a black box module (from the programmer's point of view).

```
add_customer(Session, UBFParams) ->
  % unpack UBFParameters
  % callback implementation module
  R = customer_cb:add_customer(Session,
                               Param1,
                               ...,
                               ParamN),
  % pack R in a UBF value
```

- `customer_cb.erl`, which is a callback module skeleton that must be filled in by the programmer.

```
add_customer(Session, P1,..., PN) ->
  % call domain facade
  case customer_facade:add_customer(Session,
                                    P1,...,
                                    PN) of
      {error, Reason} -> % adapt result
      R               -> % adapt result;
  end.
```

- `Customer.java`, which is an user interface facade to access the service and perform internal tasks.

2.2 Access Points and Use Cases

Each access point is described by means of it use cases. Use case descriptions include names, input parameter list, and transactional information.

As in the previous section, two Erlang modules are built from each access point description:

```
<access-point name="customer">
  <use-case name="add_customer" trans="yes">
    <param name="CustomerName"/>
    <param name="CustomerDescription"/>
  </use-case>
</access-point>
```

The Erlang generated black box module for this access point is:

```
-module(customer_facade).
-export([add_customer/3]).

add_customer(Session, CName, CDescription) ->
  ConnRef = db_facade:connect(),
  session:set_connection(Session, ConnRef),
  R = customer_facade_cb:add_customer(Session,
                         CName, CDescription),
  db_facade:endtrans(ConnRef),
  R.
```

And the callback module skeleton is:

```
-module(customer_facade_cb).
-export([add_customer/3]).

add_customer(Session, CName, CDescription) ->
  %% write your code here
  not_implemented.
```

The parameter `Session` represents the distributed database over the deployment cluster, in which client session state data is stored, providing fault-tolerance features. Database connections are organised within a supervision tree, so that automatic rollbacks can be performed if necessary.

2.3 Persistent Objects Layer

Business logic associated with a complex RMIS usually has to deal with a big number of domain entities with complex relationships amongst them. The actual coding of these objects and relationships storage is a very annoying and repeatable task.

The entities are described with their name, a list of their attributes and their relationships.

```
<entity name="customer">
  <primary-key>
    <attribute name="id" type="integer"/>
  </primary-key>
```

```
  <attribute name="name" type="string"/>
  <relation entity="account" multiplicity="*"/>
</entity>
```

Here some ideas from powerful persistence and object-relational mapping frameworks like Hibernate (Java) (Hibernate, 2006) and Active Object (Ruby on Rails) (Ruby, 2006) are borrowed.

From this definition the obtained source code will be composed by a TO module and a DAO module skeleton. An example of TO source code could be:

```
-module(customer_to).
-include("customer_to.hrl").
-export([new/2, get_name/1, set_name/2]).

new(Id, Name) ->
  #customer_to{id = Id, name = Name}]}.

get_name(CustomerTO) ->
  State#customer_to.name.

set_name(CustomerTO, Name) ->
  CustomerTO#customer_to{name = Name}.
```

And the DAO would look like:

```
-module(customer_dao).
-include("customer_to.hrl").
-export([create/2, find/2]).

create(ConnRef, CustomerTO) ->
  Id   = customer_to:get_id(CustomerTO),
  Name = customer_to:get_name(CustomerTO),
  Acc  = customer_to:get_account(CustomerTO),
  Query = "INSERT INTO customer "
          "VALUES ( " ++ Id   ++ ", '"
                      ++ Name ++ "', "
                      ++ Acc  ++ ")",
  case db_facade:process(ConnRef, Query) of
    ...
  end.
```

As we can see, the DAO assumes the existence of a database table named after the entity and columns after the attributes. Relationships are managed depending on the multiplicity of the participants.

3 ARMISTICE: CASE STUDY

ARMISTICE is a risk management information system for a large company. It is a three-tier client/server vertical application which is able to:

- Model contracted policies.
- Select the most suitable warranty to cover resources damaged in an accident.
- Manage the claims for accidents.
- Manage another several accident-related tasks (payments, invoices, repairs...).

ARMISTICE logic is structured in three components. The *risk subsystem* models a set of basic concepts: risk situations (e.g. a shop), risk situations attributes (e.g. merchandise value), risk groups (which classify risk situations), dangers (e.g. a fire),... and defines the notion of *exposure* based upon dangers and risk attributes. The *policies subsystem* uses those concepts to define *insurance policies* based upon exposures and applicability constraints. Finally, the *claims subsystem* uses the insurance policy to track an *accident* affecting one or more of the covered exposures, providing useful guidance to forecast limits and franchises and a way to manage the related tasks.

3.1 Actual Armistice Architecture

As we said before, ARMISTICE is a three-tier client/server application. Client and a server communicate with each other via XML-RPC, a remote procedure call protocol which uses HTTP as the transport and XML as the encoding.

The client, developed in Java, is a thin standalone client, because the complex interactivity advised against a web or an Erlang-based solution.

Implementing ARMISTICE server using the functional programming language Erlang has shorten the development cycle, allowed its deployment over a low-cost cluster, its scalability, and its configuration in a supervision tree. Combined with these, storage of its internal state in a distributed database gives it fault-tolerance and reliability properties.

Last but not least, the use of abstraction in the form of both functional and design patterns helped to reduce the programming effort to evolve the prototypes during the iterative development process.

3.2 Discussion

A powerful mixture of technologies has been put on practice with ARMISTICE. But, at the same time, there are a lot of work which could be automatically performed, and the ideas presented in section 2 are inspired by this reason.

At the moment of writing this article, implementation of ARMISTICE's TOs and DAOs is done completely by hand, as well as are client/server communication interfaces, etc. These tasks are very monotonous and repeatable, so our XML-based description framework will make developer's life much easier, as they will be able to concentrate on the design and implementation of really important parts (business logic) and also saving programming errors.

4 CONCLUSIONS AND FUTURE WORK

This paper has presented an innovative framework based on the evolution of our first experiences developing a real application on a distributed functional language. The proposed architecture could be reused in other traditional management software development and it is a step forward in features like maintainability, scalability and availability with regard to the early architecture.

The implementation of management software using a functional language may seem an exotic approach. But the adaptation of O.O. concepts to the functional environment is a key factor for success. Another advantage of using Erlang/OTP as the underlying platform for the system development (in addition to those common in declarative languages), is that techniques coming from the formal methods area can be applied in order to improve the system design and performance (Arts and Benac Earle, 2002), and to ensure (at least partially) the system correctness (Arts and Sánchez Penas, 2002). These techniques can be used more naturally with high level declarative languages, and their results are specially appreciated in complex management software.

Also, the design of a full application container (similar to the available in J2EE (J2EE, 2003)), its design and development could be a future work line, focusing on simpler applications where persistence matters can be hidden. Particularly interesting would be the development of a persistence layer following the same philosophy of the Ruby on Rails (RoR) Active Record, for the Erlang platform. The two main design principles of RoR, “Don’t repeat yourself” and “Convention over configuration” have proved themselves as highly productive in this architecture, taking advantage of the use of simple conventions and metaprogramming, which could be applied in the functional world.

Finally, as far as lightweight-style user interface development is concerned, we strongly believe that the integration in the framework of any kind of interface definition language like XUL (XUL, 2003) will alleviate this task, and will improve interface design and maintainability.

ACKNOWLEDGEMENTS

This work has been partially supported by Spanish MEyC TIN2005-08986 (FARMHANDS: *Recursos Funcionales para la Construccin de Sistemas Distribuidos Complejos de Alta Disponibilidad*).

REFERENCES

Armistice (2002). Armistice, Advanced Risk Management Information System: Tracking Insurances, Claims and Exposures. http://www.madsgroup.org/armistice.

Armstrong, J. (2002). Getting Erlang to talk to the outside world.

Armstrong, J., Virding, R., Wikström, C., and Williams, M. (1996). *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall.

Arts, T. and Benac Earle, C. (2002). Verifying Erlang code: a resource locker case-study. In *Int. Symposium on Formal Methods Europe*, volume 2391 of *LNCS*, pages 183–202. Springer-Verlag.

Arts, T. and Sánchez Penas, J. J. (2002). Global scheduler properties derived from local restrictions. In *Proceedings of ACM Sigplan Erlang Workshop*. ACM.

ASN.1 (2003). Introduction to ASN.1. http://asn1.elibel.tm.fr/en/introduction/index.htm.

Cabrero, D., Abalde, C., Varela, C., and Castro, L. (2003). Armistice: An experience developing management software with Erlang. *Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, pages 23–28.

Corba (2003). *Object Management Group*. http://www.omg.org.

Gamma, E. et al. (1999). *Design Patterns. Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley.

Gulías, V. M., Abalde, C., Castro, L., and Varela, C. (2005). A new risk management approach deployed over a client/server distributed functional architecture. In *Proceedings of 18th International Conference on Systems Engineering (ICSEng’05). August 16 - 18, 2005*, pages 370–375. IEEE Computer Society.

Gulías, V. M., Abalde, C., Castro, L., and Varela, C. (2006). Formalisation of a functional risk management system. In *Proceedings of 8th International Conference on Enterprise Information Systems (ICEIS’06). May 23 - 27, 2006*, pages 516–519. INSTICC Press.

Hibernate (2006). Hibernate homepage. http://www.hibernate.org/.

J2EE (2003). Enterprise Javabeans 2.0 Specification. http://java.sun.com/products/ejb/2.0.html.

Marinescu, F. (2002). *EJB Design Patterns. Advanced Patterns, Processes, and Idioms*. John Wiley & Sons, Inc.

Ruby (2006). Ruby on rails project homepage. http://www.rubyonrails.org.

XDR (2003). RFC 1832 - XDR: External Data Representation Standard. http://www.faqs.org/rfcs/rfc1832.html.

XUL (2003). XML User Interface Language. http://xul.sourceforge.net.