

Abstract

The goal of this project has been to go more deeply in the development of information systems using distributed functional technology and in the use of the object-oriented paradigm on this environment. A study about the use of object-oriented analysis and design techniques on non-object-oriented environments was done. The goal was to combine that techniques with the distributed functional programming, which provides many improvements in the application development process (robustness, fault tolerance, scalability, short development cycles, easier maintenance). We faced up to the lack of a mapping between the analysis, design and implementation, defining a representation for objects in ERLANG [1]. Thus, we achieved not to drop the quality of a development supported by a good analysis and design, and, at the same time, have the power of the ERLANG/OTP environment. Our work is explained with the development of a real application: a risk management system for a large company, where other interesting technologies are also involved.

Erlang/OTP

ERLANG is a concurrent and distributed programming language developed by Ericsson and it is specially suited for the development of telecommunication systems. The language has no constructs inducing side effects to an implicit store with the exception of communications among threads.



Values in ERLANG range from numbers and atoms (symbolic constants, lower-case in Erlang syntax) to complex data structures (lists, tuples) and functional values, which are treated as first-class citizens. A function is defined by a set of equations, each stating a different set of constraints based primarily on the structure of the arguments (pattern-matching). Iterative control flow is carried out by using function recursion.

Example

```
sort([Pivot|T]) ->
  sort([ X || X <- T, X < Pivot]) ++
  [Pivot] ++
  sort([ X || X <- T, X >= Pivot]);
sort([]) ->
[].
```

Features

- Dynamic typing.
- List comprehensions.
- Asynchronous message passing.
- Dynamic code change.
- Soft real time support.
- Multiplatform: x86, Alpha, SPARC, ARM, S/390 (zSeries), IA64, SH3, MIPS...
- Powerful collection of libraries which constitutes the Open Telecom Platform (OTP). OTP includes a distributed database, graphical interfaces, an ASN.1 compiler, a CORBA ORB, COM and Java interfaces among others.

These features make ERLANG one of the notable success exceptions of functional programming languages in real world applications [2].

Concurrent Erlang What makes ERLANG different from other functional languages is its support for concurrency and distribution. ERLANG's primitives for concurrency resembles formal calculi such as Milner's CCS [3] or Hoare's CSP [4].

Distributed Erlang The natural extension of ERLANG to a distributed framework is having processes running in more than one ERLANG virtual machine (nodes, in ERLANG terminology) possibly running at different physical nodes. The nice feature is that all the concurrency constructors are semantically equivalent in this distributed framework (even though communications among remote processes are less efficient, of course).



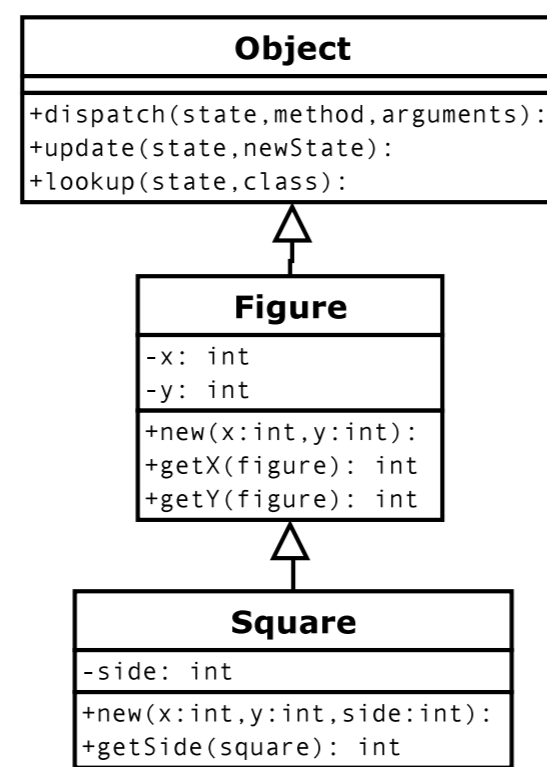
FIGURE 1: Development cluster.

Applications Nowadays ERLANG is used by many enterprises such as Alteon (now part of Nortel Networks), Cellpoint, Corelatus or One2One (now part of T-Mobile). And, of course, many Ericsson products like the AXD301 and ANx are developed at least partially in ERLANG. Additionally, the LFCIA Lab has developed many applications based in this technology in the last years, such as the VoDKA [5] video on demand streaming server or a distributed e-commerce [6] and payment gateway [7] system.

A bridge between two worlds

When we needed to deal with the way of representing objects in ERLANG, two approaches could be undertaken: the explicit and the implicit one.

Explicit structures Objects may not change and each object maps to a data structure (a record with several fields) that is implemented in an ERLANG module. The object state is an ordered list of structures (states) mapping to each level of its hierarchy. The creation of a class object with generic methods provides, for example, polymorphism.



```
-module(figure).
-record(figure, {x, y}).

new(X, Y) -> {ok, [#figure{x=X, y=Y}]}.
getX(Figure) -> Figure#figure.x.
getY(Figure) -> Figure#figure.y.

-module(square).
-record(square, {side}).

new(X, Y, Side) ->
{ok, [SuperClassState] = figure:new(X, Y),
 [ok, [#square{side=Side} | SuperClassState]].
getSide(Square) ->
Square#square.side.

-module(test).
...
getX(Object) ->
[ok, Figure] = object:lookup(Object, figure),
Figure#figure.x.
...
new(Square) = square:new(5, 8, 3),
object:dispatch(Square, getX),
...
loop(NSquare) ->
receive
{dispatch, Method, Arguments, Caller} ->
{ok, Result} = execute(Square, Method, Arguments),
{ok, NSquare} = update(Result, Square,
Caller ! {ok, Result},
loop(NSquare);
...

```

FIGURE 2: First approach: explicit structures.

Implicit structures Objects may change and each object maps to a process. Hence, the object state is the process state. The process waits for messages or requests and dispatches them, updating its internal state when necessary.

```
-module(square).
new(X, Y, Side) ->
{ok, Square} = square:new(X, Y, Side),
spawn(MODULE, loop, [Square]).

loop(Square) ->
receive
{dispatch, Method, Arguments, Caller} ->
{ok, Result} = execute(Square, Method, Arguments),
{ok, NSquare} = update(Result, Square,
Caller ! {ok, Result},
loop(NSquare);
...

```

FIGURE 3: Second approach: implicit structures.

Performance We carried out some performance tests to check the execution time penalty of a method in an ancestor against the execution in the class itself. This penalty was about 40%. We also checked the overload associated with the use of the object class. The resulting time was about 16 times worst than the direct approach.

	1	10	100	1,000	10,000
Ancestor	10	22	148	1.384	14.269
Subclass	5	13	99	958	10.515

TABLE 1: Inherit methods invocation.

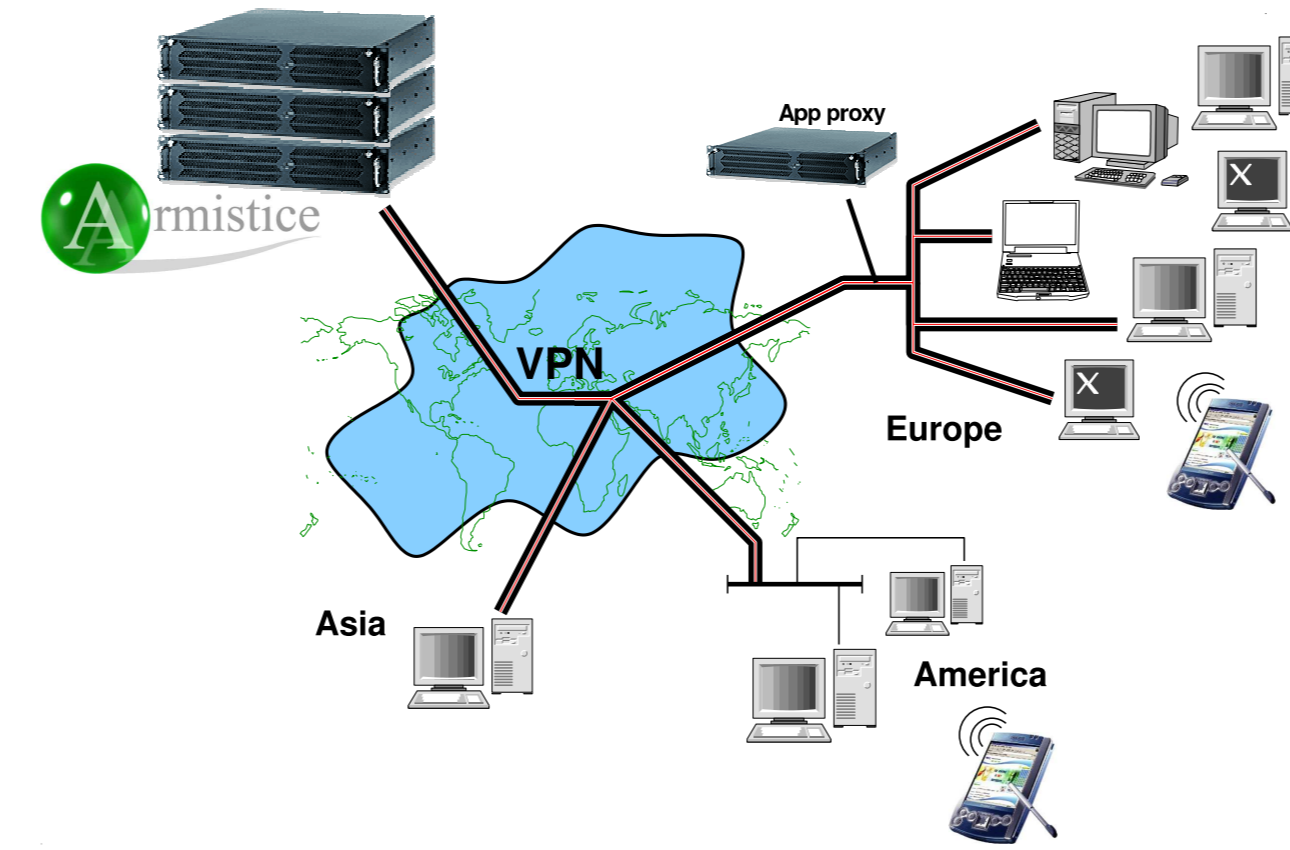
	1	10	100	1,000	10,000
Dispatch	67	256	2.215	23.879	227.589
Direct	7	16	130	1.230	13.920

TABLE 2: Method invocation with/without delegation.

ARMISTICE

ARMISTICE (*Advanced Risk Management Information System: Tracking Insurances, Claims and Exposures*) is an information system devoted to the advanced management of risks in a multinational enterprise. Among the tasks it can perform we can point out:

- Modeling of policies (language with formulas and constraints)
- Warranty selection to cover a risk situation against a damage (decision support expert system)
- Accident management
- Tracking of related activities (payments, invoices, repairs...)



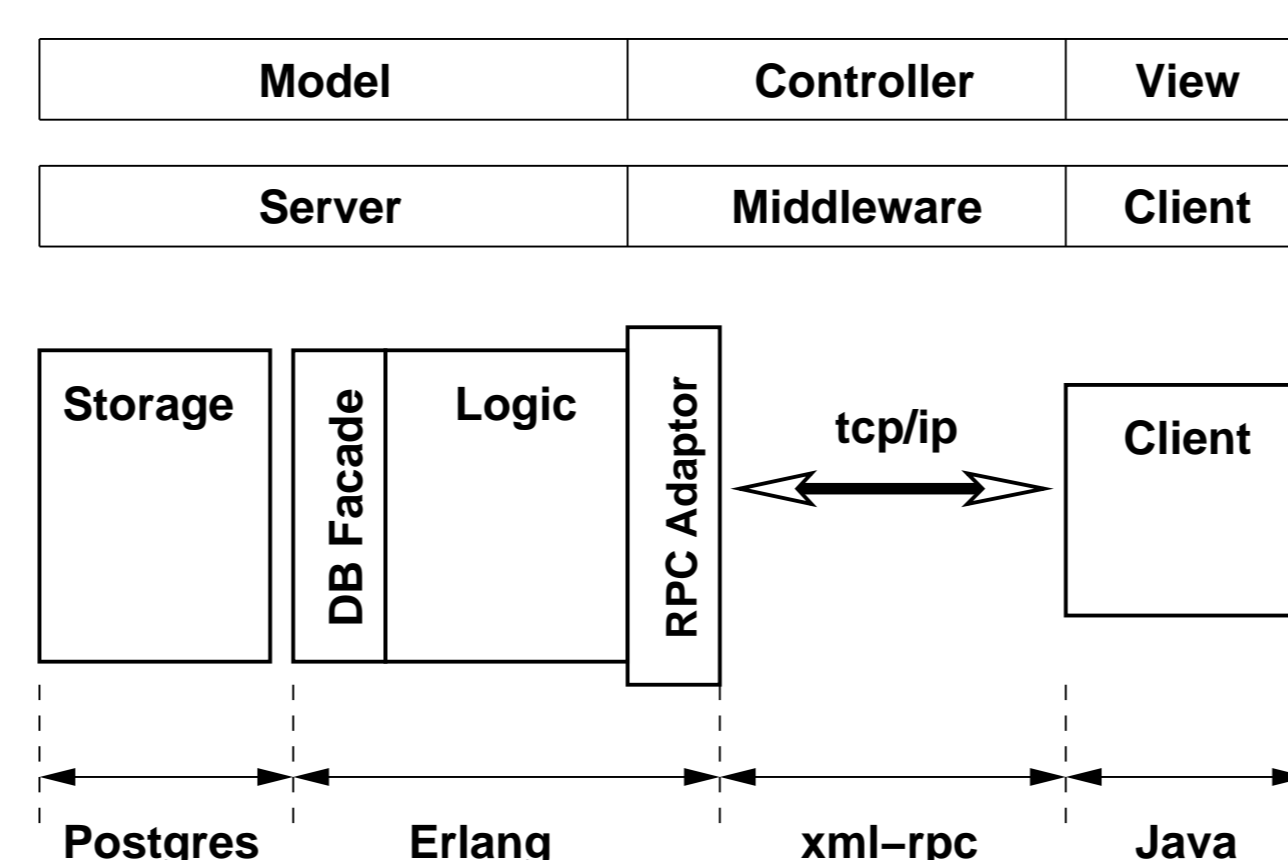
Main problems:

- Lack of specific domain knowledge (complex logic application) => **Fast prototyping**
- Communication problems with the expert => **Modeling language (UML)**

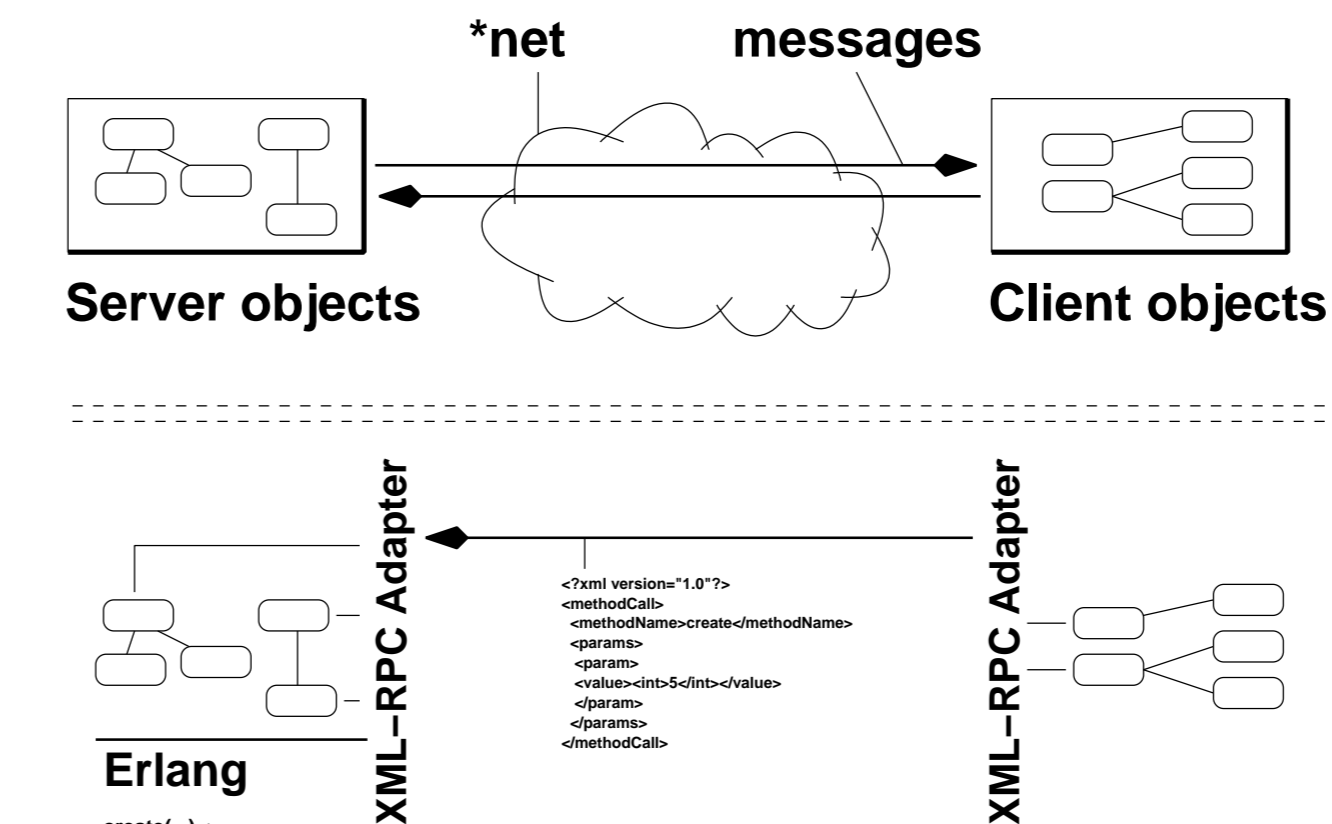
Application domain In the analysis stage three large subsystems were identified:

- **Risks Subsystem:** handles metainformation about business objects and defines the basic concepts used by the other two subsystems, such as risk situations (shops, warehouses) and dangers (fires, earthquakes, floods). It is the most abstract and dynamic subsystem.
- **Policies Subsystem:** closer to the domain, its main goal is the modelling of policies through a language of formulas and logic expressions, designed to represent conditionals and warranties.
- **Accident Subsystem:** used daily by users all over the world, it is devoted to the management of accident and all related tasks.

Architecture overview The goal is a scalable, reliable and flexible system. Using architectural design patterns like Model-View-Controller or Layered Design, the result is a three layer client/server architecture:

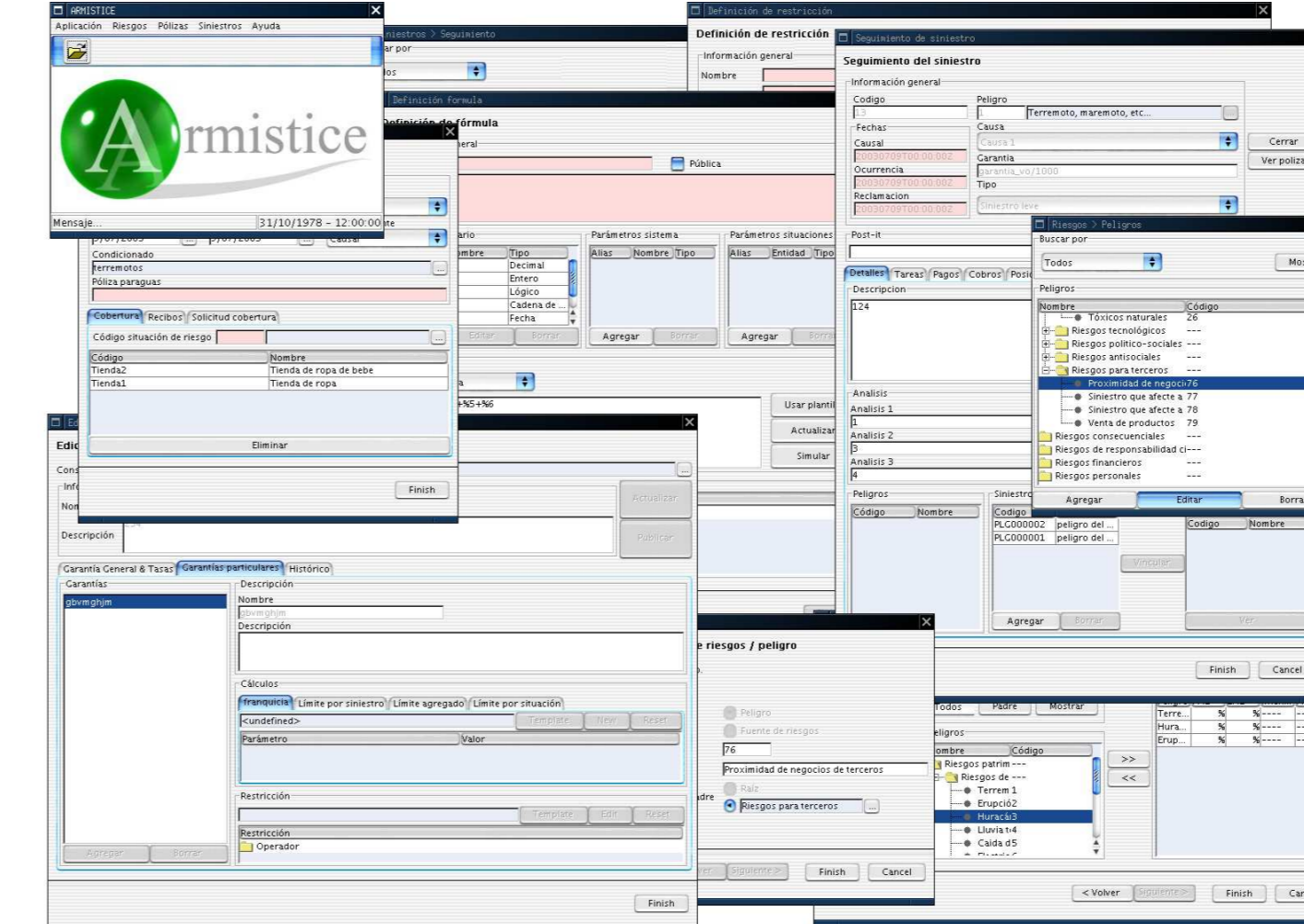


Development framework Client and server communicate with each other using XML-RPC, a remote procedure invocation protocol transported over HTTP and encoded using XML. XML-RPC fits very well with the simplicity of our clients, so it was preferred to options like SOAP.



The task of the XML-RPC adaptor is to receive request from clients, adapt them to the server, send them to it, translate the response to the client expected format and send it back to the client.

Client The client was developed using the programming language JAVA as a standalone and thin (almost with no logic) client. Due to the high number of dynamic elements and interaction possibilities among UI components, a web based client or an ERLANG client solution were dismissed from consideration.



Server The use of the functional language ERLANG was a key factor in the development of the application business logic. It contributed to shorten the development cycle, to make the deployment in a low-cost cluster, to make the system scalable. It also contributed to configure the system in a supervision tree, which combined with the storage of the internal system state in a distributed database, resulted in a system with the desired high disponibility and fault tolerance features. Finally, the application of formal methods to optimize and verify [8] key points in the system is an interesting future line of work.

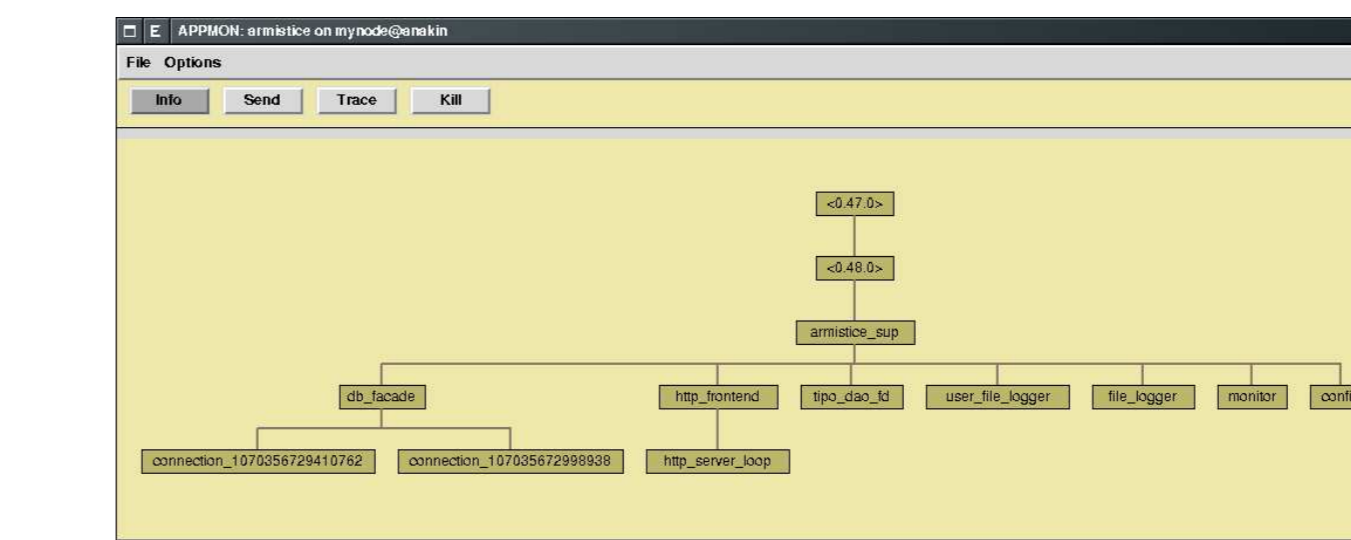


FIGURE 4: ARMISTICE supervision tree.

Implementation example

```
%% relevant: Danger x Constraint -> true | false | string
relevant(P, [Literal, String]) -> String;
relevant(P, [danger, P]) -> true;
relevant(P, [danger, Q]) -> false;
relevant(P, [negation, C]) -> do_not(relevant(C));
relevant(P, [any, Cs]) -> lists:foldl(fun(A, B) -> do_or(A, B) end,
false,
Cs);
relevant(P, [all, Cs]) -> lists:foldl(fun(A, B) -> do_and(A, B) end,
true,
Cs);
relevant(P, C) || C <- Cs ||;
relevant(P, C) || C <- Cs ||;

do_not(Op) when atom(Op) -> not Op;
do_not(Op) when list(Op) -> 'NOT' ++ Op.
do_or(OpA, OpB) when atom(OpA), atom(OpB) -> OpA or OpB;
do_or(true, OpB) when list(OpB) -> true;
do_or(false, OpB) when list(OpB) -> OpB;
do_or(OpA, OpB) when list(OpA), list(OpB) -> OpA ++ 'OR' ++ OpB;
do_or(OpA, OpB) -> do_or(OpB, OpA).
do_and(OpA, OpB) when atom(OpA), atom(OpB) -> OpA and OpB;
do_and(true, OpB) when list(OpB) -> OpB;
do_and(false, OpB) when list(OpB) -> false;
do_and(OpA, OpB) when list(OpA), list(OpB) -> OpA ++ 'AND' ++ OpB;
do_and(OpA, OpB) -> do_or(OpB, OpA).
```

The combined use of abstractions (functional patterns), compositional (both concurrent and functional composition) and design patterns [9] adapted to the development environment (Value Object, Data Access Object, Session Facade, Business Delegate, etc.) were key factors to reduce

the development efforts. Despite our initial concerns, the gap between classical software engineering and the functional programming paradigm has been successfully fulfilled.

Technologies As well as ERLANG in the server, JAVA (SWING) in the client, and XML-RPC to communicate them, in the development of ARMISTICE other technologies have been used, such as WEBDAV to perform documental management operations or POSTGRESQL to support persistent storage (through ODBC).



Performance Some of the performance tests were oriented to measure the time overhead introduced by XML-RPC on the system. It results

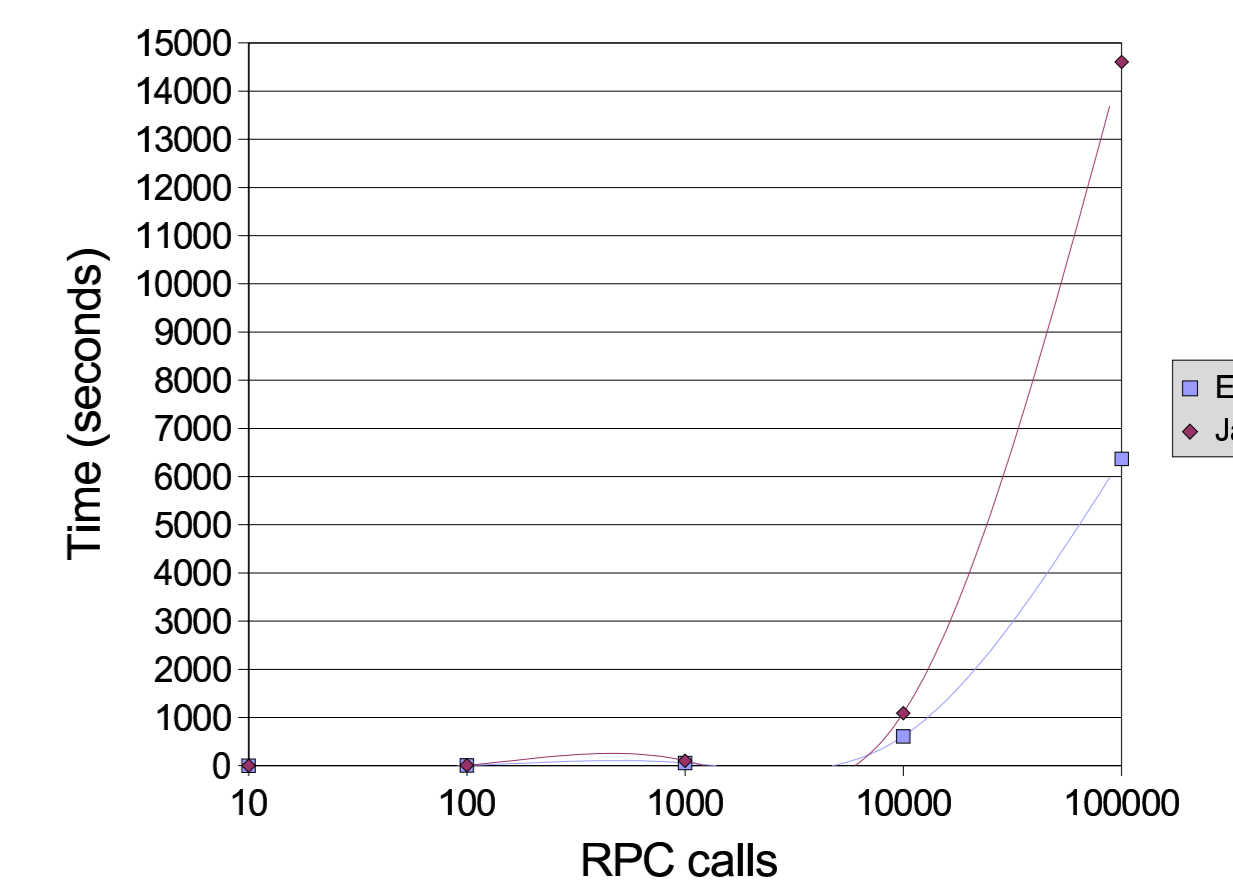


TABLE 3: Communication with/without XML-RPC.

Other tests allowed us to compare inserting times using ODBC from a program and from an ERLANG program. Here the factor is near 3:

TUPLES NUM.	Consecutive inserts num.			
	Todas	100	1,000	10,000
100	0,167	-	-	-
1,000	1,619	1,730	-	-
10,000	16,721	17,622	16,930	-
100,000	170,522	174,205	172,267	170,890
1,000,000	1,671,516	1,758,779	1,723,741	1,667,620

TABLE 4: Massive insertions from ERLANG.

TUPLES NUM.	Consecutive inserts num.			
	Todas	100	1,000	10,000
100	0,063	-	-	-
1,000	0,620	0,651	-	-
10,000	6,273	6,520	6,291	-
100,000	63,912	69,107	67,060	64,563
1,000,000	657,535	689,652	675,819	648,739

TABLE 5: Massive insertions from c.

References

- [1] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.
- [2] Philip Wadler. Functional programming: An angry half dozen. *SIGPLAN Notices*, 33(2):25–30, February 1998.
- [3] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1), September 1992.
- [4] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.
- [5] Víctor M. Gulías, Carlos Abalde, and Juan J. Sánchez. Lambda goes to hollywood. *Lecture Notes in Computer Science*, 2562, 2003.
- [6] José R. Gulías, Víctor M. Gulías, and Alberto Valderruten. Un sistema de comercio electrónico sobre un cluster de computadores. In *Proceedings of XXVIII Conferencia Latinoamericana de Informática*, Noviembre 2002.
- [7] Carlos Abalde, Víctor M. Gulías, José L. Freire, Juan J. Sánchez, and José M. García-Atizán. Development of a scalable, fault tolerant and low cost cluster-based e-payment system with a distributed functional kernel. In *Ninth International Conference on Computer Aided Systems Theory (Eurocast'03)*, LNCS. Springer-Verlag, February 2003.
- [8] Thomas Arts and Clara Benac Earle. Verifying Erlang code: a resource locker case-study. In *Int. Symposium on Formal Methods Europe*, volume 2391 of LNCS, pages 183–202. Springer-Verlag, July 2002.
- [9] F. Marinescu. *EJB Design Patterns. Advanced Patterns, Processes and Idioms*. John Wiley & Sons Inc., 2002.