Implementing Distributed Real-time Control Systems in a Functional Programming Language

Claes Wikström
klacke@erix.ericsson.se
Computer Science Laboratory
Ericsson Telecommunications AB
Box 1505
S - 125 25 Älvsjö Sweden

Abstract

The design and implementation of large distributed fault-tolerant systems in the telecommunications industry is becoming increasingly complex. Traditionally, telecommunications systems have been equipped with fault-tolerant hardware. Recent advances with cheap high speed off-the-shelf CPUs make an approach with fault-tolerance implemented in software more attractive. This typically involves different techniques to replicate data over several computers.

The actual implementation of such systems has however turned out to be a costly business. Our experiences indicate that traditional software engineering techniques for distributed systems where typically various interface description languages are used, lead to overly complex systems. The functional programming language Erlang *has been designed to ease the design of distributed fault-tolerant applications with massive concurrency.* Erlang *is currently being employed in a number of applications within the Swedish telecommunications company Ericsson, and the experiences so far are very promising.*

In this overview paper we will describe the ideas behind the support for distributed fault-tolerant programming in the language together with some implementation aspects. The main topic of the paper is the idea of dynamic interfaces. Symbolic programming languages often have the ability to dynamically identify the type of a data object at runtime. Using this capability it is possible to implement a general purpose encoder/decoder which maps data objects from the language to and from a stream of bytes which can be sent onto a network. We will show what implications this has on distributed programming techniques.

1 Introduction

The programming language Erlang has been designed in an attempt to ease the construction of large concurrent distributed software systems. Distributed telecommunications systems are traditionally implemented in imperative languages such as C, C++ or special purpose telecoms languages such as Chill [2] or PLEX [11]. The major reason for this is of course performance. It has been believed that symbolic languages do not execute fast enough for this type of applications. However, in real life telecommunications applications, the amount of time spent executing sequential code is small compared to the time spent in message passing, context switching and decoding network messages. Therefore it is feasible to implement such systems in a symbolic programming language. This is also desirable in order to decrease the complexity of the system. There is not doubt that it is easier to write and maintain an Erlang program than e.g. a C++ program. Since the software in for example a telephony switching system is extremely complex it is of paramount importance to decrease the complexity of the software. The Ericsson AXE system for example consists of over 6 million lines of PLEX source code.

By now there exists a large number of different programming languages which have been augmented with facilities for distributed programming. The system which most closely resembles distributed Erlang is an extension of Standard ML, called Facile [4].

In this paper we will describe what we believe are the key characteristics that make Erlang suitable as a distributed systems programming language. The single most important aspect is something we refer to as *dynamic interfaces*. This is a technique whereby the structure of network messages is interpreted and it partly removes the need for interface description languages such as ASN.1 [3] and CORBA IDL [10].

The structure of the paper is as follows: In this section we provide a brief introduction to Erlang. In section (2) we show an RPC service can be implemented in Erlang. In section (3) we provide an argument for dynamic interfaces and in section (4) we describe some of the implementation aspects. Finally in section (5) we discuss aspects of data management in typical telecommunications applications.

The language ERLANG is fully described in [1]. Here we give a brief introduction to the language. It is a functional language with explicit constructs to create light weight processes anywhere in a network of ERLANG nodes. Message passing between processes is asynchronous and message reception is based on pattern matching.

ERLANG is dynamically typed and an object is either a constant (atom, float, process identifier or an integer) a compound term or a variable. Variables begin with an upper case letter and '_' denotes the anonymous variable. ERLANG does not allow destructive assignment and variables are bound by the pattern matching operator '=' or by explicit pattern matching, for example in a function clause head. All ERLANG terms are ground. ERLANG does not have logical variables.

Compound terms are tuples and lists. `{a,b,c}` is a tuple of size 3 having atoms as its elements. A list is either the empty list `[]` or a pair `[H|T]`. The construction `[a,b,c]` is syntactic sugar for the pair `[a|[b|[c|[]]]]`.

A function can consist of several clauses, and clause selection is done by means of pattern matching. So for example to compute the length of a list we have:

```
len([]) -> 0;
len([H|T]) -> 1 + len(T).
```

ERLANG also has syntax for conditional expressions as well as classical functional programming constructions such as list comprehensions and lambda expressions. This will not be furher described here.

Functions are defined within a module (which is usually a file) and a compiled module is an item which can be loaded in the system. We denote a function with `N` arguments as `Fun/N` so if the above function `len/1` was defined in module `foo`, we use the syntax: `foo:len([1,2,3])` to call the function from outside the module.

Processes are explicitly created with the built in function (BIF) `spawn(Node, Mod, Fun, ArgList)`. The process will execute at the node `Node`, evaluating the function `Fun` defined in the module `Mod`. The BIF returns a process identifier (Pid) which is a built in data type. This Pid can be used to send a message to the process using the syntax

```
Pid ! Message,
```

where `Message` can be any ERLANG term. The recipient can receive messages with the syntax:

```
receive
   Pattern1 [when Guard1] ->
      Action1;
   Pattern2 [when Guard2] ->
      Action2
    ....
   [after Time
      Action3]
end,
```

When a process executes a `receive` expression, the process is suspended until a message arrives which matches one of the patterns `Pattern1`, `Pattern2`,.... or until `Time` milliseconds has elapsed. This is the only means to express timeliness in the language. For a message to be accepted, the optional guard statement must also evaluate to `true`. When a message arrives which does not match any of the patterns in the `receive` expression, the message is buffered in a mailbox. Messages are asynchronous.

Processes can be given a name by use of the BIF `register(Name,Pid)`. Once a process has a registered name, other processes may send messages to the process using the syntax: `Name ! Message` to send a message to a locally registered process or the syntax `{Name, Node} ! Message` to send a message to a remotely registered process. The name must be an atom. We can also obtain the Pid of a registered process with the BIF `whereis(Name)`.

The BIF `self()` returns the process identity of the calling process and `node()` returns the name of the local node. These BIFs are typically used when we interact with a server. A message to a server usually contains our own Pid, so the server knows where to send the reply.

The BIF `exit(Reason)` can be used to terminate the current process. Termination is abnormal unless `Reason` is bound to the atom `normal`.

Since many applications require error recovery in the event of an unplanned failure, ERLANG provides explicit error detection capabilities. ERLANG processes can be linked together by means of the BIF `link(Pid)`. Links are bi-directional. On abnormal termination of a process (a runtime error) an exit signal is sent to all processes currently linked to the terminating process. The default action for a process which receives an exit signal is to terminate and continue to propagate the exit signal. However a process can choose to trap the exit signal and transform it into a normal messages by evaluating the BIF call `process_flag(trap_exit, true)`. The following function creates a new process, monitors this process and restarts the process if it terminates.

```
keep_alive(Mod, Fun, Args) ->
   process_flag(trap_exit, true),
   Pid = spawn(node(), Mod, Fun, Args),
   link(Pid),
   receive
      {'EXIT', Pid, Reason} ->
          keep_alive(Mod,Fun,Args)
   end.
```

The function never returns.

Another error detection mechanisms is the exception handling provided by the primitive `catch`. The expression `catch Expression` always returns normally, even if `Expression` contains illegal statements. For example the expression `math:log(0)` terminates the process evaluating the expression, whereas the expression `catch math:log(0)` evaluates normally to the tuple `{'EXIT', badarith}`.

ERLANG provides higher order functions as well as the BIF `apply(Mod,Fun,Arglist)` to facilitate for meta programming. The BIF evaluates the function `Fun`, defined in module `Mod` and supplies the arguments given in `Arglist` to the function.

We also provide a BIF to monitor the wellbeing of a remote node. If an ERLANG process evaluates the BIF `monitor_node(Node, true)`, the process will receive a `{nodedown, Node}` message from the runtime system if the node should terminate (or be non existent).

Finally we want to emphasize the fact that all BIFs that work with Pids, have exactly the same semantics regardless of whether the Pid refers to a local or remote process.

2 Evaluating expressions remotely

We usually want to evaluate an expression at a remote node for either of two reasons.

- **Speed**. We want to evenly spread parts of a computation over several nodes in order to make the program go faster.
- **Location of resources**. The remote node has some resources we wish to manipulate.

In this paper we are primarily concerned with the latter of the two reasons above. In traditional computing environments Remote Procedure Calls are often used to accomplish remote evaluation. ERLANG has asynchronous message passing as its only means of process communication and we will show how to build an RPC mechanism in ERLANG which is based on message passing. Assume we run the following server code on all nodes.

```
-module(rpc)

start() ->
   register(rpc,
            spawn(node(), rpc, loop, [])).

loop() ->
   receive
      {Client, {apply, M, F, Args}}  ->
         spawn(node(), rpc,
               reply, [Client, M, F, Args]),
         loop()
   end.

reply(Client, Mod, Fun, Args) ->
   Client ! {rpc, node(),
             catch apply(Mod, Fun, Args)}.
```

It can be assumed that all nodes always evaluate `rpc:start()` when they start. This will start a server which has the name `rpc`. This server sits in a recursive loop and waits for messages that match the pattern `{Client, {apply, Mod, Fun, Args}}`. Whenever such a message arrives, the server creates an additional process which evaluates the expression `reply(Client, Mod, Fun, Args)`. This new process evaluates the RPC and sends the value of the RPC back to the caller. The `Client` variable is assumed to be the PID of the caller.

This type of "evaluation" server can be used to build a variety of RPC-like services. A very useful client function to this server is a multi node version of RPC which is called `multi:call(Nodes, M, F, A)`. This function takes a list of nodes, a module name, a function name and a list of arguments as parameters and evaluates the function in mudule `M` at all the nodes in the `Nodes` list. It can be implemented as:

```
-module(multi).

call(Nodes, M, F, A) ->
   send(Nodes, M, F, A),
   collect(Nodes, [], []).

send([], _, _, _) ->
   ok;
send([N|Tail], M, F, A) ->
   monitor_node(N, true),
   {rpc, N} ! {self(), {apply, M, F, A}},
   send(Tail, M, F, A).

collect([], Replies, Bad) ->
   {Replies, Bad};
collect([Node|Tail], R, Bad) ->
   receive
      {rpc, Node, What} ->
         monitor_node(Node, false),
         collect(Tail, [What|R], Bad);
      {nodedown, Node} ->
         collect(Tail, R, [Node|Bad])
   end.
```

The function returns a tuple `{Replies, Errors}`. It is a highly useful function, since it allows us to remotely call any function at any node without that function being aware of the fact that it is being called in an RPC. For example if we have a function to poll some hardware device called `hw:poll(DeviceName)`. If we want to call this function on a list of nodes (maybe including our own node) and we wish to collect the results we can do this by simply evaluating:

```
multi:call(Nodes, hw, poll, [DeviceName]).
```

A traditional RPC is of cource the trivial case where the length of the `Nodes` list is one.

3 Interfaces

Traditionally, distributed systems design is a mostly top-down activity since the interfaces between different processes are in the focus. All interfaces have to be specified in advance. Usually a stub code compiler is used to compile the interface specifications into routines to encode and decode the different network messages, the so called marshalling process.

This is in contrast to the code from the previous section. The `multi:call/4` function makes it possible to call *any* function at any node. It is not necessary to make any special preparation at compile time to make a function callable from a remote node. This fact has a major impact on the design and implementation of a distributed system since the implementor of a function never has to be concerned with whether a function should be callable by RPC or not.

A typical ERLANG system has two different kinds of interfaces.

- **Functions.** A module can export a number of functions.
- **Message based.** Many processes and servers export a message interface only.

What is important in ERLANG is that neither of these two types of interfaces have to be prepared specially in order to be used in a distributed environment. A server can receive any message regardless of the location of the sender of the message and functions can be called freely, locally or remote. This include polymorphic functions. Say for example that we have a polymorphic function `pmap(Data, Fun)` which takes two arguments, a data structure and a function and we want to apply the function on all sub objects in the data structure and return a new data structure of the same type, but with the function applied to all sub objects. We could have:

```
pmap({list, Data}, Fun) ->
    {list, lists:map(Fun, Data)};
pmap({tree, Data}, Fun) ->
    {tree, tmap(Fun, Data)};
pmap({Other, _} , Fun)  ->
    {error, Other}.
```

This is a general purpose function which can take any list or any tree as argument. The actual contents of the lists or the trees is unspecified. What is important is that any list and any tree can be marshalled at runtime dynamically. We refer to this as *dynamic interfaces* and it is closely related to dynamic typing.

4 External representation

At the core of heterogeneous distributed computing lies the problem of marshalling and unmarshalling data structures in an efficient, machine and language independent way. Many symbolic languages have the ability to identify data objects at runtime. Data objects are internally tagged with their type and this information is typically used by the garbage collector. For this class of languages the marshalling process can efficiently be performed at runtime without the aid of a type description language. Interface description languages have been necessary in order to generate stub code to perform the marshalling process. This leads to static interfaces that cannot be changed in runtime without recompiling.

In ERLANG all objects are dynamically typed. The main reason for this is for the garbage collector to be able to perform its task. The ability to identify the type of an object at runtime gives us the possibility to define a generic format for external data representation into which any term can be encoded and from which any term can be decoded. Hence there is a well defined mapping f such that for every ERLANG term T there is an external representation X of T such that:

$$f(T) = X$$

and

$$f^{-1}(X) = T$$

So when a process sends a message and the runtime system discovers that the recipient resides on a remote node, the function f is applied to the message in order to pack the message in a buffer before it is sent. The runtime system on the receiving side applies the function f^{-1} to reconstruct the original term.

In the real ERLANG system, the encoding and decoding is performed by built in highly optimized non-recursive C-code, however the following code fragment gives a clue to the characteristics of the function f.

```
encode(X) when integer(X) ->
    [$I, (X bsr 24) band 255,
     (X bsr 16) band 255,
     (X bsr 8) band 255, X band 255];
encode(X) when atom(X) ->
    List = atom_to_list(X),
    Len = length(List),
    [$A, (Len bsr 8) band 255,
     Len band 255 | List];
```

Here we made use of built in operator `bsr` which is bit shift right, built in operator `band` which is bitwise 'and' and the expression `$Char` which evaluates to the corresponding ASCII value of `Char`. The function must be further defined for all possible data types including the compound types, tuples and lists, where we have a recursive definition.

The corresponding (partially defined) decode function would look like:

```
decode([$I, Hi, Mid1, Mid2, Low | Tail]) ->
    {(Hi bsl 24) bor (Mid1 bsl 16) bor
     (Mid2 bsl 8) bor Low, Tail};
decode([$A, Hi, Lo | Tail]) ->
    Len = (Hi bsl 8) bor Lo,
    {Head, Rest} = strip(Len, Tail, []),
    {list_to_atom(Head), Rest};

strip(0, Rest, Ack) ->
    {lists:reverse(Ack), Rest};
strip(I, [H|Rest], Ack) ->
    strip(I-1, Rest, [H|Ack]).
```

An example session with the ERLANG shell reveals the input-output relation of the `encode` function.

```
> test:encode(256).
[73,0,0,1,0]
> test:encode(abc).
[65,0,3,97,98,99]
> test:decode([73,0,0,1,0]).
{256,[]}
> test:decode([65,0,3,97,98,99]).
{abc,[]}
```

The encoding of Pids is especially interesting. A Pid always carries information which identifies the node where the Pid originates. This makes Pids location transparent. The application programmer can use a Pid without knowing if the Pid refers to a local or a remote process.

We can identify at least the following methods of representing data objects externally.

- Ignore the issue, i.e. send objects as they are laid out in memory. This approach is efficient, but has serious consequences on interoperability between different computers.
- ASCII representation, i.e. send strings only. This is portable, easy to debug and understand. The disadvantage is inefficiency.
- Use an interface description language and a stub code compiler. This is the traditional solution. The major disadvantage is that the application programmer must decide at compile time, which data structures to send on the network. It is also difficult to work with dynamic data structures, such as trees or linear hash lists using this static technique. This is the approach of for example CORBA [10].
- Use an interpreter to perform the marshalling. This interpreter encodes and decodes data in runtime according to its type. The major advantage is that any data structure can be encoded dynamically. No interface description language is necessary. Theoretically it is possible for a stub code compiler to generate code which is faster than an interpreter. However, measurements reported in [15] show that the RPC implementation described in this paper executes significantly faster than i.e. SunOs RPC [13].

To set up a traditional RPC-based application we have to perform the following steps:

1. Define *all* interfaces in a special interface description language.
2. Run the RPC compiler to produce the stub code.
3. Link the client program, the client stubs and a runtime system into a loadable client module.
4. Link the server program, the server stubs and a runtime system into a loadable server module.

The ERLANG programmer must not concern himself with these details and we believe that we get a more powerful programming environment if we let a runtime system dynamically perform the marshalling process than if we use a static stub code compiler.

5 Data management

Many telecommunications systems typically have a small number of main CPUs controlling a large number of small CPUs and hardware devices. The main CPUs often keep very large amounts of replicated data in RAM.

One of the major advantages of using a garbage collecting language to implement non-stop systems is that the runtime system ensures that there is never any memory leakage. The garbage collector can also arrange for memory to always be allocated in such a way that memory fragmentation is prevented. Both of these aspects are extremely important in non-stop systems. It is however hard to devise a memory management strategy which satisfies all of the following properties:

1. All garbage is collected fast.
2. The collector must never stop the system for longer than a fixed time.
3. Prevents internal as well as external memory fragmentation.
4. Utilizes memory well. Many low end telecommunications systems are cost sensitive, and memory is still fairly expensive.
5. Handles normal execution garbage as well as the "very large amounts of replicated data in RAM". This data often amounts to several hundreds of megabytes. It is characterized by its size and its stability. The data is often read, and seldom written.

These requirements have been addressed in ERLANG by combining the normal memory system with built in memory resident lookup dictionaries. The dictionaries are implemented as linear hash lists [6, 8].

The hash lists are not garbage collected in the same manner as the rest of the system. This makes it feasible to implement systems that keep very large amounts of data in RAM. When some external stimuli arrives, the appropriate data objects associated with the stimuli can be located in an efficient manner. The data dictionary is updated destructively. This in contrast with the semantics of normal ERLANG objects which have write once semantics. Much research has been conducted during the past years regarding the issue of destructive update of data structures in functional languages. Most notable are monads in Haskell [14] and uniqueness typing in Clean [12]. We have however taken a more pragmatic approach in ERLANG where we impurely update the data dictionary.

Using these dictionaries it also becomes easier to implement certain types of programs. It is for example notoriously difficult to efficiently implement a program which counts the number of occurrences of each word in a large file if it is not possible to destructively update data structures. So given a function `get_word/1` that returns the next word in a file we have:

```
wcount(F) ->
   wcount(file:open(F, read), dict:new()).
wcount(Fd, Dict) ->
   case get_word(Fd) of
      {ok, Word} ->
         Old = dict:lookup(Dict, Word),
         dict:insert(Dict, {Word, add(Old)}),
         wcount(Fd, Dict);
      eof ->
         file:close(Fd),
         Dict
    end.

add(undefined) -> 1;
add({Word, Counter}) -> Counter + 1.
```

The `wcount/1` function returns the dictionary `Dict` which contains `{Word, Occurrences}` tuples for a given file.

These fast dictionaries provide the basis for efficient data management in a distributed system.

It is by now clear that the key to managing replicated data in a consistent and maintainable way is to use atomic transactions [5]. ERLANG has turned out to be ideal for the implementation of distributed transaction managers (TMs). A number of general purpose TMs have been developed.

Many projects also develop tailor made TMs which utilize semantic knowledge about the actual application. The typical telecommunications application which receives some external stimuli and acts accordingly, usually has soft real time constraints. For example if a subscriber picks up the receiver, a dial tone must be presented to the subscriber within a fixed maximum time. This means that it is not always possible to base the entire telecommunications application on standard TM locking techniques. It is sometimes more important to get some data in time, regardless of consistency of the replicas of the object.

Another important aspect of data management (as well as fault tolerance) is the ability to efficiently implement persistent data structures. When we wish to write data to disc, we essentially face the same problems as when we write data to the network. The traditional solution is again to rely on interface description languages. ERLANG instead provides the programmer with BIFs to transform any term into an opaque data object which holds the encoding of the term into the external term format. This is the same format which is used to send a term onto the network.

One very common component of telecommunications systems is a log. For example, each call in a telephony switch generates charging data which has to be logged to disc for later off line processing. So to efficiently implement a log in ERLANG we would have the following code:

```
-module(log).
start() ->
    register(log, spawn(log, go, [])).

go() ->
  Fd = file:open("LOG", read_write),
  file:position(F, eof),
  loop(F).

loop(Fd) ->
  receive
    {From, {log, Term}} ->
        B = term_to_binary(Term),
        file:write(F, B),
        From ! {log, logged},
        loop(F)
  end.
```

This code starts a server which sits in a loop awaiting requests to log terms to a file. The server calls on the BIF `term_to_binary/1` in order to transform the term into its corresponding representation in the external term format. It then appends the encoding to the LOG file. The client function which synchronously requests a term to be logged looks like:

```
log(Term) ->
  log ! {self(), {log, Term}},
  receive
    {log, logged} -> logged
  end.
```

This code is capable of logging all terms similar to the way we were able to send any term onto the

network. This is important since it means that the same log server is able to provide the logging service to all applications.

These also exists a BIF `binary_to_term/1` to perform the inverse operation, i.e. given an object representing a term in the external term format, it is possible to reproduce the original term.

Various implementations of atomic transaction managers and a highly efficient external term format interpreter in combination with the above mentioned lookup dictionary provide the basis for the implementations of fault tolerant systems in ERLANG. There is also an ongoing project to develop a full scale distributed telecommunications DBMS [9] by means of these components.

6 Conclusions

ERLANG is currently being used within Ericsson to implement large distributed soft real time control systems. This is an application area which has until now been reserved for imperative languages for reasons of speed. The combination of a highly optimized runtime systems with state of the art compilers [7] has however made distributed system implemented in ERLANG execute at least as fast as its imperatively programmed counterparts.

Hence the conclusion from the past years experience with ERLANG is that it is indeed possible to build commercial real time software with a symbolic programming language. This will pay off with decreased costs for development, market dependent adaptation of the software, as well as maintenance.

References

[1] Armstrong, J. L., Williams, M. C., Wikström, C. and Virding, S. R., *Concurrent Programming in Erlang, 2:nd ed.* Prentice Hall (1995)

[2] *CCITT High Level Language (CHILL)* Recommendation Z.200. ITU, Geneva, 1980-92.

[3] *CCITT recomendation X.208* Specification of Abstract Syntax Notation one. (ASN.1), Geneva, 1987.

[4] Giacalone, A., Mishra, P., and Prasad, S. *Facile: A Symmetric Integration of Concurrent and Functional Programming*, International Journal of Parallel Programming, Vol. 18, No. 2, 1989.

[5] Gray, J. and Reuter, A. *Transaction Processing: concepts and techniques.* Morgan Kaufmann publishers, 1993.

[6] Griswold, W. and Townsend, G. *The design and implementation of dynamic hashing for sets and tables in Icon*, Software-Practice and Experience, Vol 23(4), 351-367 (April 1993)

[7] Hausman, B. *Turbo Erlang: Approaching the Speed of C* Implementations of Logic programming Systems, ed. Tick, E. and Succi, G., Kluwer 1994.

[8] Larsson, P-Å Larsson. *Dynamic Hash tables* Communications of the ACM, **31**, (4), (1988)

[9] Nilsson, H., Törnquist, T. and Wikström, C. *A Distributed Real-Time Primary Memory DBMS with a Deductive Query Language* 12th International Conference on Logic Programming, 1995.

[10] The Object Management Group. The Common Object Request Broker: Architecture and Specification, 1992.

[11] *PLEX language description, Ericsson document LXT 101 680 Ue R1,* 1983, Ericsson Telecommunications AB.

[12] Smetsers, Sjaak, Barendsen, E., van Eekelen ,M. and Plasmeijer ,R. (1994), *Guaranteeing safe destructive updates through a type system with uniqueness information for graphs*, In Proc. of Graph Transformations in Computers Science, International Workshop, Dagstuhl Castle, Germany, Schneider and Ehrig Eds., Springer-Verlag, LNCS 776, pp. 358-379.

[13] *SunOs 4.0 reference manual Vol. 10* 1987 Sun Microsystems, Inc.

[14] Wadler, P. *Comprehending Monads* ACM Conference on Lisp and Functional Programming, Nice, June 1990.

[15] Wikström, C. *Distributed programming in Erlang* First International Symposium on Parallel Symbolic Computation, Linz Austria, 1994.