Distributed programming in Erlang

Claes Wikström[*]
Computer Science Laboratory
Ellemtel Telecommunications Systems Laboratory[†]
Box 1505
S - 125 25 Älvsjö Sweden

Abstract: The construction of computer systems consisting of more than one computer is becoming more common. The complexity of such systems is higher than single computer systems. This paper addresses the question of how to simplify the construction of large concurrent distributed systems. To do this we have augmented the functional concurrent programming language ERLANG with constructs for distributed programming. Distributed programs written in ERLANG typically combine techniques for symbolic functional programming with techniques for distributed programming. In contrast to traditional imperative languages ERLANG does not need interface description languages to specify the format of interprocessor messages in a heterogeneous network. This considerably simplifies distributed programming. Distributed ERLANG is currently being employed in a number of large software projects within the Ericsson group.

Keywords: distributed programming, concurrency, symbolic programming, programming languages, fault tolerance, real-time systems

1 Introduction

ERLANG is a functional concurrent programming language designed for the construction of large concurrent programs. ERLANG is typeless in the same sense as traditional logic languages, uses pattern matching for variable binding and function selection, has explicit mechanisms to create concurrent processes and advanced facilities for error detection and recovery. Non distributed ERLANG is described in [1]. An efficient implementation of non distributed ERLANG is described in [8].

The construction of large distributed fault-tolerant concurrent real-time system is an extremely complex task. The programs that are used to control a telephony network typically consist of several millions lines of source code. Such systems consists of several cooperating CPUs. We claim that the languages we use to program such systems must not only support concurrency, but also support distributed processing. This paper describes how mechanisms supporting distributed programming have been added to the ERLANG language. We show how a small set of operations can be used in a variety of ways to implement different well known distributed programming mechanisms.

We also include measurements of the speed of the system which indicates the quality of the implementation.

Distributed ERLANG is *not* an experimental system. It is currently being used in a number of product projects within Ericsson. One interesting prototype project that has been reported is the implementation of an Intelligent Networks Development System [10]. Distributed

[*]email: klacke@erix.ericsson.se
[†]A company jointly owned by Ericsson and Telia AB

ERLANG is implemented as a very loosely coupled system of computers communicating with TCP/IP connections. There are no shared resources and each node is completely autonomous.

By now there are a number of languages which support distributed programming. In distributed imperative languages such as Orca [2] and Concurrent C [7] all inter process communication *interfaces* have to be specified at compile time. We will show the advantage of *not* having to specify interfaces at compile time. It is considerably easier to implement distributed applications in ERLANG than in these languages due to the high level of abstraction in the ERLANG code.

Extensions to concurrent logic languages that support the programming of loosely coupled systems can be found in [6] and [5]. These system have many similarities with distributed ERLANG. From a distributed programming perspective, the main difference is that the unit of concurrency in Concurrent Logic Languages is the goal and the clause. In a loosely coupled system it is not appropriate to have the goals of a clause evaluated in parallel on different nodes due to the overhead that is associated with the network. To circumvent this, the language DRL [5] defines an additional unit of concurrency called the *grain*, where a grain is a set of predicates. In ERLANG the unit of concurrency is the same as the unit of distribution, namely a process. A process is identified by its name, thus providing a more homogeneous view to the programmer.

2 Non distributed Erlang

Here we give a brief introduction to non distributed ERLANG. An object is either a constant (atom, float, process identifier or an integer), 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 a syntax for conditional expressions which will not be 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. 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.

ERLANG has explicit mechanisms to create new processes. The built in function (BIF) `spawn(Mod,Fun,[Arg1,Arg2,Arg3...])` is used to create a new process. The process will initially run the function `Fun`, defined in module `Mod`, and given the arguments supplied. The BIF returns a process identifier (pid) which is a builtin 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. For a message to be accepted, the optional guard statement must also evaluate to `true`. Guards will not be further described here. 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 have a registered name, other processes may send messages to the process using the syntax: `Name ! Message`. The name must be an atom. We can also obtain the pid of a registered process with the BIF `whereis(Name)`.

The BIF `self()` can be used to acquire our own pid. This BIF is 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 run-time 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 transformed it into a normal messages using the BIF `process_flag(Flag, Value)`. 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(Mod, Fun, Args),
   link(Pid),
   receive
      {'EXIT', Pid, _} ->
          keep_alive(Mod,Fun,Args)
   end.
```

The function never returns.

Another error detection mechanisms is provided by the primitive `catch`. The expression `catch Expression` always returns normally, even if `Expression` contains illegal statements. For example the expression `1 - foo` terminates the process evaluating the expression, whereas the expression `catch 1 - foo` evaluates normally to the tuple `{'EXIT', badarith}`.

ERLANG does not provide higher order functions, but the BIF `apply(Mod,Fun,Arglist)` is available for meta programming. The BIF evaluates the function `Fun`, defined in module `Mod` and supplies the arguments given in `Arglist` to the function.

3 Distributed Erlang

We have augmented the ERLANG language with constructs for distributed programming. This is basically done with a small set of new BIFs. A number of typical models for distributed computing have been implemented with the aid of the new BIFs and they can all be found in ERLANG standard libraries. This includes for example: Remote Procedure Call (RPC) [3], different broadcast algorithms, a global name server, asynchronous RPC, ISIS like named process groups with multicasting [4], multi node dictionaries, a load distribution package, a parallel evaluator, shared data structures etc. In this section we will give (shortened) implementations of some of these.

The new BIFs are as follows:

- `spawn(Node, Mod, Fun, Arglist)`. Creates a new process executing at the remote node `Node` where a *node* is a running ERLANG system.
- `node_link(Node)`. Sets up a unidirectional link to the node `Node`. If `Node` is non existent or fails, an exit signal is sent to the process executing the BIF.
- `node_unlink(Node)`. Removes a link to node `Node`.
- `{Name, Node} ! Mess`. Sends, asynchronously, the message `Mess` to the process registered as `Name` on `Node`.
- `node()`. Returns our own node identifier, which is a globally unique atom.
- `nodes()`. Returns a list of nodes we are currently connected to.
- `disconnect_node(Node)`. Removes the connection to `Node`.
- `alive(Name, Port, Setup)`. Makes a non distributed system become a networked node. This involves starting the right network drivers and announcing the name of the system to the network.

There are also a number of other BIFs, for example `link/1`, that take pids as their arguments. These BIFs have been modified to transparently work whether the pid is local or remote.

An ERLANG system becomes a networked node by evaluating the BIF `alive/3`. Once this is done, the system can communicate with other ERLANG nodes. A connection to an other node is setup the first time the name of the other node is used in a distribution BIF. Once a connection is set up, it remains active until explicitly removed by a call to the BIF `disconnect_node/1` or until the remote node fails. Hence distributed ERLANG is a very loosely coupled system where nodes may come and go dynamically. Each node is a completely autonomous entity and nodes interact by creating new processes and sending messages.

3.1 A Generic Concurrent RPC

One of the most well known programming techniques for distributed systems is the RPC. It is a technique to evaluate a function on a remote node. The code to implement a generic RPC service in ERLANG consists of a server process which must run on all nodes, and a client function which is used to send the RPC request to the right server for evaluation:

```
-module(rpc).
-export([start/0,loop/0,reply/4,call/4]).

call(Node,M,F,A) ->
    node_link(Node),
    {rpc,Node} ! {self(),{apply,M,F,A}},
    receive
        {rpc,What} ->
            node_unlink(Node),
            What;
        {'EXIT',Node,Reason} ->
            exit(Reason)
    end.
```

And the server code to perform an actual RPC. The server creates a new process for each request.

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

loop() ->
    receive
        {Cli,{apply,M,F,A}}  ->
            spawn(rpc,reply,[Cli,M,F,A]),
            loop()
    end.

reply(Client,M,F,A) ->
    Client ! {rpc,catch apply(M,F,A)}.
```

The following code fragment appends two lists at the node `Node`.

```
  L1 = [a,ee,rr,tt],
  L2 = [5,6,7],
  rpc:call(Node,lists,append,[L1,L2]),
```

As another example we may wish to find the Pid of the registered process `init` at `Node`:

```
  P = rpc:call(Node,erlang,whereis,[init])
```

Since all BIFs are defined to reside in the module `erlang`, this is the way to call a BIF as an RPC. Note that the server `rpc` spawns a new process for each request. This means that the same `rpc` server can concurrently execute several RPCs. This type of server is often referred to as a concurrent server, as opposed to an iterative server which processes requests one at a time in the order they arrive. One major advantage of the concurrent server is that if an individual request hangs, the server continues to operate correctly. Note the use of `node_link/1`. The call is encapsulated within a matching pair of `node_link/1` and `node_unlink/1`. We are guaranteed that a call to `rpc:call/4` either:

- Terminates normally, returning the computed result or `{'EXIT', Reason}` if a computational error occurred.
- Exits the process which made the call if the remote node fails.

3.2 Promises

RPCs are synchronous. This means that the caller is suspended until the computation terminates. The main disadvantage of RPCs is that we have to do an *idle wait* for the answer, although we may have something better to do than to wait. The use of promises has been suggested in [12]. Promises are an asynchronous variant of remote procedure calls and a promise is a place holder for a future return value from an RPC. This overcomes the disadvantages that stem from the synchronous nature of the RPC. The caller can perform the RPC, do something else and at a later time, try to claim the computed value, which may or may not be ready. This is implemented as follows:

```
-module(promise).
-export([call/4, yield/1,
         nb_yield/1, do_call/5]).

call(Node,M,F,A) ->
    spawn(promise,do_call,[self(),Node,M,F,A]).

do_call(ReplyTo,N,M,F,A) ->
    R = rpc:call(N,M,F,A),
    ReplyTo ! {self(),{promise_reply,R}}.

yield(Key) ->
    receive
        {Key,{promise_reply,R}} -> R
    end.
```

Each call is handled by a local process which performs the RPC. The process identifier of this process is returned as a key which can be used in a subsequent blocking yield operation. This example nicely demonstrates the power of the selective receive in ERLANG.

A non blocking version of `yield/1` can be written as:

```
nb_yield(Key) ->
    receive
        {Key,{promise_reply,R}} ->
            {value,R};
    after 0 ->
        timed_out
    end.
```

This function merely checks whether there is a message in the mailbox matching the expression `{Key,{promise_reply,R}}`, if there is not, it returns the atom `timed_out`.

3.3 Parallel evaluation

Now we have sufficient tools to write a parallel evaluator. The following function evaluates a list of `{Module,Fun,Args}` tuples in parallel on all nodes:

```
parallel_eval(ArgL) ->
    Nodes = nodes(),
    Keys = map_nodes(ArgL,Nodes,Nodes),
    lists:map(promise,yield,Keys).

map_nodes([],_,_) -> [];
map_nodes(Argl,[],Orig) ->  %% one more round
    map_nodes(Argl,Orig,Orig);
map_nodes([{M,F,A}|Tail],[Node|More], Orig) ->
    [promise:call(Node,M,F,A) |
     map_nodes(Tail,More,Orig)].
```

We use the asynchronous variant of RPC from module `promise`, and generate a list of promises, each acting as a place holder for the desired value, and then traverse the list yielding all the values with the standard `map/3` function. The second clause of `map_nodes/3` is entered when we have run out of nodes, that is, when there are more goals than available nodes.

It is interesting to note the behavior of the list `Keys` here. Once we have the list `Keys` we try to `yield` the elements one by one. If one of the calls to `yield` suspends this does not affect the evaluation of the remaining elements of the list. They all continue to evaluate and their results will be collected when the last suspended call to `yield/1` returns.

The algorithm here places the jobs on different nodes in a predefined order. This algorithm does not take account of different functions being more expensive than others. Some nodes may get a lot of work and other nodes get less. This is known as *static load distribution*. If we want to implement dynamic load distribution, we can use the BIF `statistics(run_queue)` which returns the number of processes that are scheduled to run. This has been showed in [9] to be a most effective way of predicting the future load of a node.

3.4 A pseudo server

Finally we will give the implementation for something we call a *pseudo server*. The standard way of providing a specific service in an ERLANG system is to hide the service in a server, have the server register with the name of the service, and have the clients access the server via its name. For example in the standard ERLANG libraries we have an interface to X windows. This is implemented as a server registered under the name `pxw`. All processes that wish to do graphics, do so by calling functions that send messages to the `pxw` server. If we run ERLANG on a host which does not have X windows (for example a telephone switching system) and we still want to run graphics applications on the host we can use the following function:

```
start(Node, Name) ->
    Id = spawn(pseudo,relay,[Node, Name]),
    register(Name,Id).

relay(Node, Name) ->
    P = rpc:call(Node,erlang,whereis,[Name]),
    link(P),
    loop(P).

loop(RealPid) ->
    receive
        Anything ->
            RealPid ! Anything
    end,
    loop(RealPid).
```

If we know that the node `Node` supports graphics programming through the server `pxw`, we can evaluate the expression `pseudo:start(Node, pxw)` to bring the service of `pxw` to the node which does not support this.

The pseudo server relays all messages to the real server which resides on `Node`. This allows the server to execute on a remote node without the clients being aware of this. The advantage of this is that code which is written for a single node system can be used to access the server whether it is running remotely or locally. The standard ERLANG file system is also accessed through a single registered server, hence this technique is also used to provide the service of files to nodes without a disc.

4 Implementation

ERLANG code is executed in an abstract machines representing a particular implementation of the language. One such abstract machine is described in [8]. In this section we describe the implementation of the distribution aspects of the language. In UNIX an ERLANG node executes as one UNIX process internally containing many ERLANG processes.

The implementation of distribution is split in two parts, one part which is integrated in the runtime system, and one part which is network dependent. The network dependent part is something called a *linked-in driver*[1]. The driver is easily replaceable when we want to run distributed ERLANG on top of different kinds of network media. The driver is responsible for setting up connections to remote nodes, monitoring the well being of these connections, reading and writing buffers of bytes on those channels, Currently only a driver for TCP/IP is implemented. Figure 1. depicts our architecture.

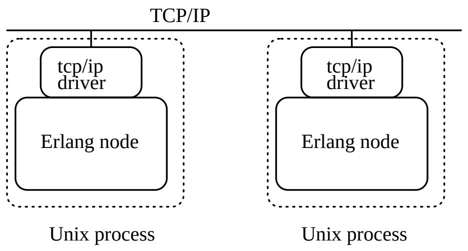

Figure .1 Architecture

4.1 External representation

One key to the implementation is the observation that in ERLANG (and in most symbolic programming systems) all data objects are identifiable at runtime. ERLANG data objects are internally tagged with their type. For example, an integer in ERLANG is represented as one 32-bit word with 4 bits set to a special value reserved for integers and the 28 remaining bits holding the actual integer value and a tuple is represented as a tagged

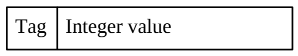

Figure .2 Integer Representation

pointer to a consecutive area of words, each word holding the value of each element in the tuple. Hence its possible 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

[1]The concept of linked-in drivers in ERLANG is a way to link any type of software into the kernel of the runtime system. Very much like a device driver in UNIX, this allows for different configurations of the systems on different hardware. A driver communicates with the runtime system through a structure of function pointers. One special driver is designated as the *distribution* driver.

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 we send a message and the runtime system discovers that the recipient resides on a remote node, it applies the function f to the message in order to pack the message in a buffer before it sends it. The runtime system on the receiving side applies the function f^{-1} to reconstruct the original term. An alternative to the usage of a machine independent external representation of terms would have been to send terms as they are internally represented in memory. This may be slightly faster. However, with an external representation we achieve interoperability between different implementations of Erlang on different hardware platforms. Hence we can write distributed applications which run in heterogeneous environments.

This simple idea is applicable to all programming systems which have the ability to identify the type of a data object at runtime.

4.2 Atom cache

In real-life applications a large fraction of the data inside messages are atoms. In typical distributed applications we have measured 45 % of all objects in inter-node messages to be atoms. Atoms are internally represented as a tag and an index in an atom table. When we encode an atom, we would like to just send the index instead of the textual representation of the atom. This can only be done if we have identical atom tables on all nodes. This however requires that the nodes either are statically configured or rooted at a master node or that we employ a complex protocol to maintain the atom tables identical. These solutions are unattractive. In the implementation of distributed Erlang each node maintains a cache for each connection to a remote node. Each cache has 255 slots. When an atom is to be encoded (to be sent a away) the cache associated with the node of the recipient is consulted. If the atom is present in the cache the index in the cache is sent, if the atom is not present, the atom is inserted in the cache (possibly overwriting some old value) and the atom in its entirety together with the newly acquired cache index is sent. The receiving node will then insert the atom in its cache. Measurements of cache hit ratio in distributed telephony systems applications indicate a hit ratio around 95 %. This means that 19 out of 20 atoms sent to a remote node are externally represented as a single byte, being the index in the cache. It also means that the receiving side need not calculate a hash value for the atom in order to insert the atom in the system atom table. Also this idea is applicable to other programming systems.

4.3 Code management

We have to be able to easily load code in a runtime system over the network. This has been implemented through the introduction of a builtin datatype called *binary* in the language. A *binary* data object merely represents a piece of untyped memory. In order to load a module, we first obtain the object code for the module in a *binary* object `Bin`. Then we call the code loading BIF with `Bin` as parameter. If we wish to load the object code on a remote node, we simply use the RPC facility to load the object code remotely. By separating the obtaining of object code from the loading, we can implement dynamic code management systems in Erlang itself.

4.4 Distributed I/O

Applications that do I/O in a distributed system typically want all I/O that is produced to go back to the node where the original call was made. We want to print all I/O that is produced by an application on a tty that is connected to the node where the application was started. This is solved in Erlang by the introduction of process groups. Every process has a process group leader and all I/O that is produced will be sent to the process group leader. When new processes are created the new process will have the same group leader as the 'spawning' process, thus organizing all processes in a tree. The Erlang I/O system is entirely implemented in Erlang itself, and this system has been rewritten to accommodate for process groups.

4.5 Security

Another matter which is addressed is security and user authentication. Each node is assigned a secret string, its *magic cookie*. In order for a node, say N1 to communicate with an other node N2, the node N1 must know the *magic cookie* of N2. Every message which N1 sends to N2 will be patched with an atom which is the atom that N1 believes that N2 has as its cookie. If a message arrives to a node with the wrong *magic cookie*, the runtime system ensures that the message is sent to a special ERLANG process which is responsible for unauthorized messages. The matter of obtaining other nodes cookies is a local issue. In the case of the UNIX implementation, each node initially reads a string from the file $HOME/.erlang.cookie, and sets its own cookie to the string found in the file. Thus all nodes running with the same UNIX user id can communicate without being aware of the authentication schema at all. This authentication schema can be easily adapted to be used in conjunction with the authentication schema of any local operating system. The *magic cookie* is an atom, so it will most probably end up in the atom-cache, hence authentication costs virtually nothing.

4.6 Miscellaneous

Common constructs from logic and functional programming languages that are missing in ERLANG are for example logical variables and higher order functions. This makes the implementation of distribution simpler.

Each node is autonomous, and since there are no logical variables and network messages are always copied, there are never any references between nodes. This means that we can do local garbage collection. Some algorithms for distributed garbage collection have been proposed, for example [11]. We feel however that none of the proposed algorithms are appropriate for the type of fault tolerant, soft real-time systems that ERLANG is intended for.

Distributed ERLANG is implemented on a number of different computer systems including several versions of UNIX, VxWorks, QNX, and Windows NT allowing for transparent communication between applications running on all of these systems.

5 Discussion

Processes in ERLANG are primarily intended to act as an abstraction to aid the system designer to map a real world problem on a software architecture. A similar argument can be applied to nodes in ERLANG. Our concern is to provide abstractions that aid the design of systems that already are inherently distributed.

Networking is expensive in a loosely coupled LAN based system and many distributed algorithms try to minimize network traffic. System design becomes more complex since we have to be aware of this. For example, it takes considerably longer time to spawn a process on a remote node than on the local node. The process which spawns a remote process is suspended for the time it takes for the spawn request to traverse the network and the time for the new process identifier to travel back. This time can be significant and depends on the speed, quality and load of the network.

When a number of nodes are attached to an Ethernet LAN there is a significant difference in the time between the creation of a local process and a remote process. The same goes for local versus remote server calls. We have the following table indicating the cpu time and the real-time to create local and remote processes:[2] This shows

	Local spawn	Remote spawn
CPU time	0.08	0.8
Real-time	0.09	6.1

Table 1: Process creation times

that it takes about 100 times longer real-time to create a remote process than a local process. We also spend 10 times more local cpu time to create it. It is important to bear this in mind when designing a distributed application, since one of the goals with distributed computing is to make the application not only more reliable but also faster.

If we examine the RPC facility from Section 3. it is worth noting that we did not have to specify the interface to the RPC server. In traditional RPC systems, there is usually an interface description language involved. An example of such a language is Sun Microsystems RPC language [13].

These languages do not primarily exist because one wishes to have the interface to the RPC server in a separate document from the code, but because one has to use the interface specification to generate stub routines

[2] All times are milli seconds and the test was run on Sun Sparc 1+ workstations. These times actually include the time to run and terminate a process.

which marshal the arguments to the function. In ERLANG there is no need to have an interface description to generate stub routines, since ERLANG data structures are self identifying. For example the *pseudo server* is not possible to implement if the interface to the server have to be defined at compile time.

An other example where it is most inconvenient to specify interfaces is when we wish to move some large data structure from one node to an other. For example, assume we wish to bring one node down. Before we do that we want to transfer an entire data base to a standby node. In ERLANG we just send the data base in its entirety in a single send operation.

This leads to very dynamic systems, where interfaces to services available on remote nodes can be changed in a running system. An ERLANG programmer can concentrate on specifying the important logical interfaces in his system without having to be concerned with the physical interfaces or lower level protocols between computers.

6 Performance

Since distributed ERLANG is used in industrial applications, performance is of paramount importance. There is a tradeoff between reliability and speed. Since message passing has to be completely reliable TCP is used as transport medium as opposed to UDP. We have also measured the speed of the RPC application in the paper and compared it to the RPC facility in SunOs 4.1 [13]. We have measured the time to do a single RPC. The function we call is a function which simply returns it's argument where the argument is an integer or a structure/tuple containing one integer and three short strings. The c-structure is defined as:

```
struct s {
     int i;
     char a[25], b[25], c[25];
};
```

and the corresponding ERLANG data structure is

```
{Int, Atom, Atom,Atom}
```

We get the following figures running on Sun Sparcstations 1+. (All figures are milli seconds) The last line in the table ('Erlang Server Call') is the time to call a function which sends a message to a server and have that server call a function which return the message to the sender. This functionality is close to the functionality of the SunOs RPC. The functionality of the ERLANG RPC is higher than the SunOs RPC, since it is completely generic.

	Integer	Tuple
Sun RPC	3.6	6.1
Erlang RPC	5.1	5.3
Erlang Server call	3.9	4.1

Table 2: RPC execution times

We can see that an RPC in ERLANG is slower than a SunOs RPC when we merely send an integer, but when we send a structure as the argument the ERLANG RPC is faster than the SunOs RPC[3].

If we turn off the atom cache mechanism we get the figures in Table 3 where we can see that the atom cache gives us a considerable speedup. It is also worth to mention that the additional time we get with the atom cache turned off is wasted execution time in the CPU as opposed to idle time waiting for a response since all messages in this test will fit in a single Ethernet frame.

	Integer	Tuple
Erlang RPC	6.0	6.6
Erlang Server call	4.5	5.1

Table 3: RPC execution times without atom cache

7 Future work

We are currently experimenting with different models for process migration. We also plan to have the ability to have multiple network drivers. This would then make it possible to have a node to act as a router between for example the telephony network and a LAN.

8 Conclusions

The aim has been to provide an industrial programming environment which makes the construction of distributed

[3]The actual code to perform the test consists of 76 lines of c code and 12 lines of interface specification code, whereas the ERLANG code to run the test consists of 15 lines of code.

systems easier. We believe that distributed ERLANG is precisely this. The small number of concepts including, the creation of new processes, asynchronous message passing and the links provide very easy to understand semantics. Especially when constructing very large systems it is of paramount importance to have well defined and easy-to-understand bottom level mechanisms. The unit of distribution, the process, is the same as the unit of concurrency. This is important when we construct transparent distributed systems.

In this papper we have showed how:

- We can construct a completely generic RPC.
- No interface description language is necessary.
- It is possible to efficently implement dynamic systems that do not need interface descriptions.

and we claim that the increased level of abstraction in distributed ERLANG programs makes it considerably easier to design and construct large efficient distributed systems.

9 Acknowledgments

We would firstly like to thank the original creators of ERLANG Joe Armstrong, Robert Virding and Mike Williams. Thanks goes also to Per Hedeland for invaluable advice and comments to the TCP/IP part of the implementation.

References

[1] Armstrong, J. L., Williams, M. C. and Virding, S. R., *Concurrent Programming in Erlang,* Prentice Hall (1993)

[2] Bal, H. *Programming distributed systems* Prentice Hall 1990.

[3] Birell, A.D., Nelsson, B. J. *Implementing remote procedure calls* ACM Trans. Comp. Syst. Vol. 2 No. 1 1984.

[4] Birman, K., Cooper, R., Marzullo, K., Makpangou, M., Kane, K., Schmuck, F., Wood, M. *The ISIS system manual 2.1* Cornell University 1990

[5] Diaz, M., Rubio, B., Troya, J.M. *Implementation issues of a distributed real-time logic language*, International Conference of Logic Programming, Workshop on 'Integration of Deductive Paradigms', 1994.

[6] Foster, I. *Parallel implementation of PARLOG*, Proc. 1988 Int. Conf. Parallel Processing (Vol. II), pp 9-16, St. Charles, IL.

[7] Gehani, N., Roome, W. *Concurrent C, The Programming Language* Prentice Hall 1989.

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

[9] Kunz, T. *The influence of different workload descriptions on a heuristic load balancing scheme.* IEEE Trans. Software Eng., Vol. 17, No. 7, July 1991 pp. 725-730

[10] Meer, J. A Prototype demonstrating User Mobility and Flexible Service Profiles. Ericsson Review. no 1, 1994.

[11] Ladin, R., Liskov, B. *Garbage Collection of a distributed heap* The 12th International Conference on Distributed System, Yokohama Japan, 1992.

[12] Liskov, B. *Linguistic Support for Efficient Asynchronous Calls in Distributed Systems* Proceedings of the SIGPLAN 88.

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