Programming Distributed Erlang Applications: Pitfalls and Recipes

Hans Svensson

Dept. of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
hanssv@cs.chalmers.se

Lars-Åke Fredlund [*]

Facultad de Informática, Universidad Politécnica de Madrid, Spain
fred@babel.ls.fi.upm.es

Abstract

We investigate the distributed part of the Erlang programming language, with an aim to develop robust distributed systems and algorithms running on top of Erlang runtime systems. Although the step to convert an application running on a single node to a fully distributed (multi-node) application is deceptively simple (changing calls to `spawn` so that processes are spawned on different nodes), there are some corner cases in the Erlang language and API where the introduction of distribution can cause problems. In this paper we discuss a number of such pitfalls, where the semantics of communicating processes differs significantly depending if the processes reside on the same node or not, we also provide some guidelines for safe programming of distributed systems.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Reliability

1. Introduction

Our aim is to write and debug distributed algorithms that uses the Erlang distribution mechanism for process communication. To do this in a successful way we must have a precise knowledge of the communication guarantees given by the Erlang distribution mechanism. This is needed in order to determine whether the guarantees matches the (varying) requirements of the distributed algorithms. A large part of this research program is devoted to the development of a formal semantics, which includes distribution, for the Erlang programming language [**?**]. In the task of writing a model checker implementing the "distributed Erlang" [**?**], we came across a number of areas where we were not absolutely certain that the formal semantics actually were an accurate account of the behavior of the Erlang distribution layer. However, since not every critical aspect of the distribution support is documented[1], it is necessary to do some experimental work. We have written programs that test various basic features of the runtime system, and also examined the source code of the runtime system, in order to draw conclusions regarding the real behavior of the distributed parts of Erlang.

The outcome is the companion paper refining the distributed semantics of Erlang [**?**] and this paper where we focus on the practical consequences of the support for distribution available in Erlang. That is, illustrating what can go wrong, and providing advice on how can we program reliable distributed applications using the Erlang distributed mechanism.

2. Intra-Node Programming

The basic tools for programming Erlang intra-node applications are well known. Communication is achieved using the send and receive constructs; links and monitors are used to build robust applications that survive process failures.

The basic tools for programming fault tolerant Erlang applications are the *link* and *monitor* constructs. They permit one process to receive a failure indication when another process has terminated. A common abstraction mechanism used in implementations of distributed applications is *failure detectors*, which serves the same purpose as the Erlang links and monitors.

Note that the link and monitor mechanism, when monitoring another process at the same node, cannot guarantee any kind of "semantic liveness" of the monitored process. It may be that the monitored process is waiting for the reception of a message that is sure never to arrive. It is defacto dead, but no process termination message will ever reach the monitoring process. For that, and other reasons, the use of timers to bound the waiting time for process communication is needed even in the intra-node case.

Below we exemplify in subsections the basic communication guarantees provided for intra-node communication.

2.1 Basic Message Passing Guarantee: Stream Semantics

The basic guarantee for message passing, in the intra-node case, is that messages sent from one process to another are delivered in order, if they are delivered at all, without message corruption or duplication. When we say that a message has been delivered to a process P we mean concretely that the message has been left in the mailbox of process P. Also, which is unfortunate, when a message m has been delivered to P we say that P has received the value. This does not mean that P has chosen to *extract* the message from its mailbox using a `receive` construct, but only that the message is available in its mailbox.

In practice the guarantee means, for instance, that if two messages m_1 and m_2 are sent, in order, from a process Q to a process P then either

[*] The author was supported by a Ramón y Cajal grant from the Spanish Ministerio de Educación y Ciencia, and the DESAFIOS (TIN2006-15660-C02-02) and PROMESAS (S-0505/TIC/0407) projects.

[1] except, of course, in the source code…

Erlang'07, October 5, 2007, Freiburg, Germany.
Copyright © 2007 ACM 978-1-59593-675-2/07/0010…$5.00

- the messages are delivered, in order, at P
- only the m_1 message is delivered (effectively P crashed sometime after the reception of m_1)
- none of the messages are delivered

Such a message sending and reception guarantee matches well the communication guarantees provided by TCP/IP channels, which are commonly used to connect Erlang distributed nodes. We refer to this communication guarantee as the **streaming guarantee**.

Note that the guarantee says nothing about the semantics of the receiving process; i.e., whether the process really processed the received value and acted on it. The process P could for instance have a bug that causes it to crash just after receiving $m1$. If the sender, process Q, wishes to establish that P has acted on its messages, a two-way protocol has to be established. The communication guarantees listed above guarantees only that there is no way that P could have received, and acted on m_2, without first having received and acted on m_1.

Of course acted on m_1 before m_2 here should be understood in a wide sense; it could mean to ignore m_1, i.e., not extracting it from the mailbox until m_2 is extracted first, using a "selective" `receive` statement.

2.2 Multi-Party Communication

In the case of intra-node communication even stronger assumptions about communication patterns can be valid. In Claessen and Svensson [**?**] an example is given where three processes P_1, P_2 and P_3 communicate as follows:

- P_1 sends message m_1 to P_2
- P_1 sends message m_2 to P_3
- P_3 forwards message m_2 to P_2

In the case when all processes are located on the same node, message m_1 is going to be delivered to P_2 before m_2 is, as a message can essentially just be stuffed into its mailbox by the runtime system upon the send by P_1. In other words, in the intra-node case an application protocol might be right to assume that there is no chance that m_2 is going to be delivered before m_1, however it is a risky assumption to rely on for future versions of Erlang/OTP[2].

3. Inter-Node Programming

When an Erlang application is composed of processes situated on different nodes, the situation is very different from the single node case. There is no longer a single unique runtime system, that conceptually has all the information needed to make an informed decisions whether a process has crashed. In a distributed, multi-node application, nodes may be geographically remote from each other and there is in general no way to distinguish whether a communication failure regarding a remote process (on a different node) is due to; (1) that its node is temporarily isolated (for example because of a network failure) from the sending node, or (2) because the runtime system at the remote node has crashed, and therefore all the processes on the remote node has terminated as well. To identify, possibly erroneously, failed nodes the Erlang runtime system regularly sends messages between nodes (ticks). If on a node N_1 a tick fails to arrive from a node N_2, that node is considered crashed and process on N_1 that have linked or monitored to a process on N_2 will get an error indication. However, the possibility exists that the communication failure is only temporary, i.e. not due to a remote node crash, and that the communication can be reestablished with the remote node in the future.

```
#!/bin/sh
NODE=$1

while [ 1 -lt 2 ]; do
  erl -sname $NODE
  sleep 1
done
```

Figure 1. Code example: Node restart script

Even though the implementation of communication in the distributed case is very different from the intra-node case, Erlang promises that communication is not different. Quoting from the (old) Concurrent Programming in Erlang book:

> Messages can be sent to remote processes and links can be created between local and remote processes just as if the processes were executing on a local node. Another property of remote Pids is that sending messages to a remote process is syntactically and semantically identical to sending to a local process. This means, for example, that messages to remote process are always delivered in the same order they were sent, never corrupted and never lost.

4. Pitfalls of Inter-Node Programming

To revise the distributed semantics of Erlang we set out to perform a number of experiments to check to which extent the vague promise regarding the non-difference between intra and inter-node programming actually holds.

To summarize the findings, two new significant differences were encountered, both breaking the basic communication guarantee in the intra-node case (**streaming guarantee**). It turns out that there are situations when some messages may be lost, but later messages not. That is, a process Q that sends in sequence two messages m_1 and m_2 to a remote node P may find that only message m_2 was delivered, never m_1. The first problem stems from too early reuse of process identifiers (pids), as in between the sending m_1 the node on which process P is executing may crash, eventually restart, and when message m_2 is sent it may then be delivered to a new process spawned on the newly restarted node.

The second situation concerns temporary communication problems between two nodes, leading to an erroneous detection of nodes as crashed, which can cause messages sent to be silently dropped.

In this section we also comment on a couple of other peculiarities of distributed communication (pitfall 3 and pitfall 4) that can lead to application errors if not taken into account.

4.1 Pitfall 1: Pid Reuse

Process identifiers, are normally thought of as globally unique. However, they also consists of finite structures, and can therefore not be unique at all times. Using sufficiently large finite structures, however, it should be unnecessary for practical purposes to consider the risk of process identifier reuse.

Thus it was as a big surprise to see just how easy it was to create processes with the same pid, after node crashes. In the following example, we spawn a process, make sure that it is dead, and then successfully communicate with it!

To run the example we use a simple shell-script in Fig. **??** that restarts an Erlang-node whenever it is killed.

The example, which is shown in Fig. **??**, works as follows; First we spawn a process on one node (N1) which sets up the experiment, executing the function `run`. We assume that node N2 has already been started using the shell script in Fig **??**. The process at node

[2] There is a conjecture that an interaction with garbage collection could render the assumption invalid even for the current Erlang/OTP implementation; we have not observed such a case in practise.

```
-module(pidReuse).

-export([start/0,run/0,echo/0,communicator/1]).

-define(N1,'n1@localhost').
-define(N2,'n2@localhost').

start() ->
    spawn(?N1,?MODULE,run,[]).

run() ->
    erlang:process_flag(trap_exit,true),
    Pids =
        lists:map
        (fun(N) ->
             Pid1 = spawn_link(?N2,erlang,self,[]),
             spawn(?N2,erlang,halt,[]),
             timer:sleep(2000),
             Pid1
         end, lists:seq(1,3)),
    lists:foreach(fun(Pid) ->
                    receive {'EXIT',Pid,_} -> ok end
                  end,Pids),
    spawn(?N2,?MODULE,echo,[]),
    communicator(Pids).

echo() ->
    receive {From,N} -> From!{self(),(N+1)} end.

communicator(Pids) ->
    lists:foreach(fun(Pid) ->
      io:format("Trying to communicate with: ~w\n(~w)\n",
                [Pid,term_to_binary(Pid)]),
      Pid!{self(),5},
      receive {Pid2,N} ->
          io:format("Recieved ~w from ~w\n(~w)\n",
                    [N,Pid2,term_to_binary(Pid2)])
      after 2000 ->
          io:format("No reply!\n")
      end
    end, Pids).
```

Figure 2. Code Example: Pid reuse

N1 begins by trapping exits and then executes three times in row the following sequence of instructions: create and link to a process executing a terminating function at node N2, then kill node N2 (using the `halt` command), wait 2 seconds (to permit a restart of N2). Having spawned three processes, whose pids have been stored in the `Pid` variable we next wait for three exit-messages to make sure that all spawned processes spawned at (the different instances of) N2 are actually dead. Finally we spawn yet another process at N2 which executes the function `echo` and then we call the function `communicator` at N1 giving the three collected pids as argument. The `communicator` function tries to communicate with the dead processes, and as we can see from the execution log in Fig. **??** we do indeed get a reply from one of the processes! Apparently the new process executing `echo` at N2 was spawned with the same process identifier as a process spawned on an earlier instance of N2.

4.1.1 Analysis

Apparently pids are reused very quickly indeed after node crashes and restarts. Clearly it is not safe to use pid equality, when communicating with remote processes, to identify uniquely another process. Thus in a distributed protocol one likely has to resort to developing and using another mechanism to accurately identify communicating partners such as e.g., adding an instance counter to every message, that is safely increased during every node restart (a standard mechanism anyway in many distributed protocols).

```
Trying to communicate with: <4888.40.0>
(<<131,103,100,0,12,...,49,49,56,0,0,0,40,0,0,0,0,3>>)
Recieved 6 from <4888.40.0>
(<<131,103,100,0,12,...,49,49,56,0,0,0,40,0,0,0,0,3>>)
Trying to communicate with: <4888.40.0>
(<<131,103,100,0,12,...,49,49,56,0,0,0,40,0,0,0,0,1>>)
No reply!
Trying to communicate with: <4888.40.0>
(<<131,103,100,0,12,...,49,49,56,0,0,0,40,0,0,0,0,2>>)
No reply!
```

Figure 3. Output: Pid reuse

```
-module(comm).

-export([start/0,snd/2,rcv/0]).

-define(N1,'n1@host1.domain.com').
-define(N2,'n2@host2.domain.com').

start() ->
    Pid = spawn(?N1,?MODULE,rcv,[]),
    spawn(?N2,?MODULE,snd,[Pid,1]).

rcv() ->
  receive
      N -> io:format("got ~p~n",[N]), rcv()
  end.

snd(Pid,N) ->
    Pid!N,
    timer:sleep(1000),
    snd(Pid,N+1).
```

Figure 4. Basic Erlang communication

One should note that it is not a coincidence that we used exactly three processes in the example above. Internally in the nodes there is an incarnation counter taking the values 1,2 and 3 and after three restarts it has wrapped.

To prevent problems, we consider it *highly advisable* that the current Erlang/OTP implementation is changed to prevent pids from being reused so soon (having a larger space in the pid structure for the node restart counter).

4.2 Pitfall 2: Misunderstood Basic Communication Guarantees under Distribution

In the second example we focus on the commonly held belief that the communication channels provided between two Erlang processes cannot silently loose messages, and show that it doesn't hold if two nodes can be disconnected from each other, and later reconnected.

The problem is illustrated by the following simple Erlang program, depicted in Fig **??**. The program spawns a process repeatedly sending increasing natural numbers, located at node N1, and a receiving process that simply prints the stream of incoming numbers located on node N2.

A log of the information printed by the receiving process in a typical run is included in Fig **??**.

In the log we can see that the number increase in the expected fashion until the jump between 74 and 146, signifying that a number of messages were lost, clearly breaking the property of no message loss. What caused this to happen? Concretely we simply yanked the Ethernet cable connecting one of the nodes, and waited

```
...
got 71
got 72
got 73
got 74
got 146
got 147
got 148
...
```

Figure 5. Output: Process Communication

for a short interval, until replugging the cable. The operation is time sensitive, if we wait too short an interval until replugging there is no message loss (corresponding to the frequency of sending tick messages [3]).

Apparently we cannot rely on the normal TCP/IP channel semantics of message passing when node communication failures takes place. The conclusion is that when building distributed algorithms on top of a distributed Erlang channel we have to assume that messages can be silent dropped, or, we have to introduce code that monitors every process-to-process communication to detect possibly node communication failures (see examples further on in the article).

From studying further the Erlang literature we can see that the phenomenon is actually acknowledged in Barklund and Virdings carefully written natural language semantics for Erlang [**?**]. We quote:

> **10.6.2. Order of signals**
> ...It is guaranteed that if a process P_1 dispatches two signals s_1 and s_2 to the same process P_2, in that order, then signal s_1 will never arrive after s_2 at P_2. It is ensured that whenever possible, a signal dispatched to a process should eventually arrive at it. There are situations when it is not reasonable to require that all signals arrive at their destination, in particular when a signal is sent to a process on a different node and communication between the nodes is temporarily lost.

Note that in this context a message is a signal instance. In other words, there are no promises regarding safe delivery of signals (except no reordering), especially during temporary communication failures.

4.2.1 Analysis

For building reliable applications that can be distributed there seems to be only a few options, considering the failure to enforce the TCP channel semantics for distributed communication:

- Construct only applications whose protocols for distributed process communication are insensitive to dropped messages. This requirement is easy to satisfy if we can afford to send along the whole protocol state in all communication but in practice this will often be too expensive in terms of message traffic/time. An alternative is to to add sequence counters messages to all messages and drop (or postpone delivery) messages that arrive out-of-order. In addition resending of messages has to be implemented. In other words, implementing in the application a protocol with TCP characteristics on top of the Erlang distribution mechanism.
- Instrument the protocol to assume an irrecoverable communication failure every time a inter-node communication failure (may) have taken place. Such failures can be detected by using the link or the monitor mechanism to always link or monitor remote communicating processes. When such a failure has taken place, no more messages arriving from the remote process can be trusted, as there may be messages missing. That is, we forbid further communication with the suspect process.

Alternatively, and more speculatively, one could require that the Erlang distribution mechanism change:

- Implement a proper Erlang transmission protocol, with communication guarantees similar to TCP, but which can recover (by keeping information about sending and reception sequence numbers, and messages sent but not yet acknowledged) from TCP channel failures. That is, whenever a TCP channel is broken down, and another one is created between two nodes, the nodes remember which messages have been sent, and which have not yet been received, so that the sending node can resend them. In other words, implementing in the Erlang runtime system a distribution protocol with TCP characteristics replacing the normal Erlang distribution protocol.
- Forbid reconnection of separated nodes. Simply never allow two nodes that have been previously connected, and later separated, from ever reconnecting. In essence, this forces one of the nodes to restart. This may appear overly brutal, but for some applications it would be a useful behavior.

4.2.2 A Revised Message Passing Guarantee

Note that we can rephrase the streaming guarantee from Sect. **??** if we let Q monitor the state of the receiving process P, either using a link or a monitor construct:

If two messages m_1 and m_2 are sent, in order, from a process Q to a process P, and Q is linked (or monitors) to P, and no exit message from P is received at Q[4], then it is guaranteed that the messages have been delivered, in order, at P.

In case an exit message from P is received at Q, any of the following sequences of messages may have been delivered at P: ϵ, $\langle m_1 \rangle$, $\langle m_2 \rangle$, $\langle m_1, m_2 \rangle$.

4.3 Pitfall 3: Weaker Multi-Party Communication Guarantees

In Sect. **??** we saw that multi-party communication in the intra-node case is rather deterministic, with regards to message orderings. As is well known, in the inter-node communication case this is not so.

- P_1 sends message m_1 to P_2
- P_1 sends message m_2 to P_3
- P_3 forwards message m_2 to P_2

If, in the above example, all three processes reside on different nodes there exists the possibility that message m_2 will be delivered to process P_2 before message m_1 is delivered.

Interestingly, in the current Erlang implementation it appears that the stronger communication guarantee concerning guaranteed delivery of m_1 before m_2 still holds if the processes P_2 and P_3 (or P_1 and P_3, or P_1 and P_2) are located on the same node. This is because Erlang uses essentially, a single stream for all communication between a pair of nodes (instead of creating a dedicated stream for each couple of processes communicating over the node barrier). Thus when m_1 is sent to P_2 in the above example it will always be handled and delivered to P_2 before m_2 (unless there is a network disconnect as discussed earlier!) since m_1 and

[3] see Erlang documentation for setting `net_ticktime`

[4] How long to wait for the reception of an exit message depends on the setting of `net_ticktime`.

m_2 are sent on the same stream and thus serialised in the order of m_1 before m_2. Note that it is highly dubious to let an application rely on this obscure guarantee, as one could imagine that a re-implementation of the Erlang distribution mechanism could change this behavioral characteristic

4.4 Pitfall 4: Failure Detectors are not Perfect

In the intra-node case whenever a process has died, linked or monitoring processes will be informed of that fact. Since all processes execute within the same runtime system, the detection of a terminated process is of course perfect. In other words, a process reported terminated will never reappear.

In the inter-node case, when one process monitors (or links to) a process on another node, the remote process may be reported dead while it is indeed still alive. This happens for instance when two nodes are separated (the network tick algorithm times out), causing remote monitored or linked processes to be reported dead. However, if the connection between the nodes are later reestablished, these remote processes can continue communicating.

Again, if re-connections between separated nodes are allowed, failure detectors can never be perfect as in general there is no possibility to distinguish a failed network link (which can be reestablished) from a failed node (which cannot).

5. A Programming Discipline for Distributed Applications

From the communication example in Figs. **??** and **??** it is clear that when trusted communication is needed we must we must supervise the communicating processes. There are no special built-in mechanisms for doing exactly this, but instead we have to rely on the general mechanisms link and monitor and build our own programming discipline. In this section we present one simplistic such programming discipline, to illustrate our point. We also present some speculative possible additions to the distributed mechanisms in Erlang, that could make writing robust distributed applications easier.

5.1 A Simple Programming Discipline

In the revised example in Fig. **??** we have added some extra code to make sure that messages are not lost. (Note: There are a gazillion of other things that can go wrong, this example only deals with communication errors). In this example the sending process is monitoring the receiving process. As soon as the sender gets an error it stops sending messages and instead goes into ’synchronization mode’. The sender continue to monitor the receiver, and eventually (in the case of a node disconnect and reconnect) the receiver will re-appear. The two processes then synchronize, which involves transmitting the state of the receiver at the time of the network disconnect to the sender. The sender then continues from the point where the connection broke down. There are a few drawbacks with this simplistic approach (such as the the semi busy-wait loop, and the fact that the sender must cache everything it has sent), and more intricate schemes are certainly possible. However, it does illustrate our point, and in Fig. **??** we see the output from an example run. We see that although there is a communication failure in the middle, the sequence is not broken.

```
-module(commFixed).

-export([init/0,rcv/1,snd/2,sync/1]).

-define(N1,’n1@host1.domain.com’).
-define(N2,’n2@host2.domain.com’).

init() ->
    Rcv = spawn(?N2,?MODULE,rcv,[none]),
    spawn(?N1,?MODULE,snd,[Rcv,1]).

rcv(N) ->
    receive
        {sync,Snd} ->
            Snd ! {sync,N},
            N1 = N;
        X ->
            io:format("got ~p\n",[X]),
            N1 = X
    end,
    rcv(N1).

snd(Rcv,N) ->
    erlang:monitor(process,Rcv),
    snd_(Rcv,N).

snd_(Rcv,N) ->
    receive
        {’DOWN’,_,_,_,noconnection} ->
            N = sync(Rcv)
    after 1000 ->
        ok
    end,
    Rcv ! N,
    snd_(Rcv,N+1).

sync(Rcv) ->
    erlang:monitor(process,Rcv),
    Rcv ! {sync,self()},
    receive
        {’DOWN’,_,_,_,noconnection} ->
            timer:sleep(5000),
            sync(Rcv);
        {sync,N} ->
            snd_(Rcv,N+1)
    end.
```

Figure 6. Basic Erlang communication – With monitor

```
...
got 71
got 72
=ERROR REPORT==== 4-Jul-2007::15:14:06 ===
** Node ’n2@host2.domain.com’ not responding **
** Removing (timedout) connection **
got 73
got 74
...
```

Figure 7. Output: Process Communication – With monitor

5.2 Possible Extensions to Erlang

Having seen all the potential trouble caused by distributed programs so far, it is interesting to do some speculation about possible additions to the Erlang runtime system. Such changes could take place at several levels. One possibility is to add a new Erlang API that provided explicit control of the handling of node reconnect, offering the possibility to forbid such reconnects or to permit them (possibly after informing and preparing local node processes of that fact).

Another alternative is to extend the information provided to individual processes communicating with remote nodes. A very commonly used OTP-construction (in non-distributed systems) is a supervisor structure. It would be very nice to have something similar for process-structures distributed over several runtime systems. There are however several options on how such a thing should work. One plausible idea would perhaps be to have a node supervisor structure built using links and monitors, where different actions could be specified depending on the event (Connection of nodes, Disconnection, Re-connection, etc). The main problem with this is how to guarantee that these node supervisors are the first that starts/stops communicating when an event occur. Another idea would be to have one single node-supervisor at each node, to which process could register. In this case a process that is communicating with a process at a different node could register this at the node-supervisor together with a suitable action to take if errors occur. (Where suitable actions could be: crash, restart, ignore or inform).

A less intrusive addition to the Erlang API would be to inform not only of nodes going down (`nodedown` messages emitted by the `monitor_node` function) but also report re-connected (or restarted) nodes, i.e., let `monitor_node` emit also `nodeup` messages.

6. Conclusions

In the paper we have presented the results of some experimental work aimed to get a better understanding of the distributed functionality of Erlang. In our work with a model checker for Erlang (McErlang [**?**, **?**]) and our work with a more accurate semantics for distributed Erlang [**?**] we found some cases where we could not, short of writing experimental code, find out how the runtime system was working. The result was that we found two interesting significant differences between the vague promises of the documentation and the actual working of the runtime system; (1) the reuse of Pids and (2) silently dropped messages in the case of node disconnect and later reconnect.

The reuse of pids is maybe more of a technical remark, although it could have severe implications if a critical system is built that rely on pid uniqueness. To prevent problems, we consider it advisable that the current Erlang/OTP implementation is changed to prevent pids from being reused so soon (having a larger space in the pid structure for the node restart counter).

The often misunderstood guarantees regarding distributed communication is a more problematic issue. The whole problem stems from the more general (and notoriously hard) problem of handling network partitioning. Therefore we have tried to give some guidelines of how to deal with the situation in a safe way, as well as speculated about possible changes/additions to Erlang that could make handling the situation easier.

The final message is that, although most applications are not affected by these problematic issues, their implication should considered in most distributed application development. It may very well be that the correct decision is to not include any new code handling e.g., pid reuse etc, as the application protocol may not be sensitive to such problems. However, it is crucially important that the decision is an informed one.

Acknowledgement

Thanks are due to Joe Armstrong for his valuable comments on a draft of this paper.

References

[BV99] J. Barklund and R. Virding. Erlang 4.7.3 reference manual. Draft (0.7), Ericsson Computer Science Laboratory, 1999.

[CS05] K. Claessen and H. Svensson. A semantics for distributed Erlang. In *Proceedings of the ACM SIPGLAN 2005 Erlang Workshop*, 2005.

[FE06] L-Å. Fredlund and C. Benac Earle. Model checking Erlang programs: The functional approach. In *Proceedings of the ACM SIPGLAN 2006 Erlang Workshop*, 2006.

[FS07] L-Å. Fredlund and H. Svensson. McErlang: A model checker for a distributed functional programming language. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 2007.

[SF07] H. Svensson and L-Å. Fredlund. A more accurate semantics for distributed Erlang. In *Proceedings of the ACM SIPGLAN 2007 Erlang Workshop*, 2007.