编写分布式的 Erlang 程序：陷阱和对策

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

Lars- Åke Fredlund*
Facultad de Informatica, Universidad Politecnica de
Madrid, Spain
fred@babel.ls.fi.upm.es

摘要

为了在 Erlang 运行时系统基础上开发更可靠的分布式系统和算法，我们研究了 Erlang 编程语言中分布式的部分。使用 Erlang，把一个运行在单个节点上的程序转换成完全分布式（运行在多个节点上）的应用程序可谓易如反掌（只需要修改对 spawn 函数的调用，使之在不同节点上产生进程）；但尽管如此，Erlang 语言和 API 中仍然有一些阴暗的角落可能在引入分布式运算时带来问题。在本文中，我们将介绍几个这样的陷阱：在这些地方，取决于进程是否运行在同一个节点上，进程间通信的语义会有显著的差异。我们同时还提供了一些关于“编写安全的分布式系统”的指导原则。

分类和主题描述 *D.3.3【编程语言】：* 语言构造和特性

关键字： 可靠性

1. 简介

我们希望能够编写和调试用到 Erlang 的分布式进程通信机制的分布式算法，为此我们必须清楚 Erlang 的分布机制对进程间通信提供了哪些保障——要判断这些保障是否与我们的分布式算法的各种需求相符，首先必须了解它们。很大部分的研究工作都是在为 Erlang 编程语言（包括分布机制）开发形式化语义[CS05]。在实现“分布式 Erlang”的模型检查器[FS07]时，我们有几处无法完全肯定形式化语义是否精确描述了 Erlang 分布层的行为。但由于并非所有关于分布式支持的重要部分都有文档记录[1]，做一些试探工作自然是必不可少的。我们编写了大量程序来测试运行时系统的各种基本特性，同时对运行时系统的源代码也做了检验，从而逐渐勾勒出 Erlang 语言中分布式部分的真实行为。

我们得到的成果是另一篇关于精化 Erlang 分布式语义的论文[SF07]，以及本文：我们将在文中着重关注 Erlang 目前提供的分布式支持带来的实际效果——我们将展示哪些代码会出错，并就“如何借助 Erlang 的分布机制编写可靠的分布式应用”提出我们的建议。

---

* 该作者由西班牙教育与科学部提供的拉蒙卡哈基金（Ramón y Cajal grant）和 DESEFIOS（TIN2006-15660-C02-02）、PROMESAS（S-0505/TIC/0407）等项目共同资助。

[1] 当然了，有源代码，如果那也算文档的话……

2. 节点内编程

Erlang 节点内编程的基本工具可说是人所共知了：用 send 和 receive 来实现通信；用链接（link）和监视器（monitor）来构造健壮的、在单个进程失败时也不会崩溃的应用程序。

正如前文所说，链接（link）和监视器（monitor）是编写具有高容错性的 Erlang 程序的基本工具：借助这两种语言特性，当一个进程终止时，它可以向另一个进程发送失败信息。在分布式应用开发中有一个常见的抽象机制叫做失败侦测器（failure detector），其用途跟 Erlang 的链接和监视器毫无二致。

请注意，“链接和监视器”机制——监视同一节点上的另一个进程——并不保证被监视的进程在语义上正确：被监视的进程有可能在等待一个永远不会到来的消息，这时它实际上等于已经死掉了，但监视它的进程永远也不会收到“进程终止”的消息。为此（以及其他一些原因）有必要用计时器（timer）来限制进程通信的等待时间，即便各个进程都在同一节点内。

下面我们将逐一展示节点内通信能够得到的基本保障。

2.1 基本消息传递保障：流语义

节点内消息传递的基本保障是：由一个进程发送给另一个进程的消息，只要消息能够送达而没有丢失或重复，就必定是按发送顺序送达的。在这里，“将一条消息送达进程 P”意味着这条消息被放入进程 P 的收件箱；同时只要消息 m 被送达 P，我们就说 P 收到了消息 m 的值——这并不代表 P 在此时用 receive 语句从收件箱中取出的**一定是**消息 m，只代表消息 m 一定是在收件箱里。

在实际编程中，这项保障意味着：如果进程 Q 先后发送两条消息（m_1 和 m_2）给进程 P，那么消息送达的情况必定是下列三种情况之一：

- 两条消息按发送顺序送达 P
- 只有 m_1 被送达 P（等效于 P 在收到 m_1 之后崩溃）
- 两条消息都不被送达

这种消息发送和接收的顺序保障类似与 TCP/IP 通道所提供的通信保障——后者常被用于连接多个分布的 Erlang 节点。我们把这种通信保障称作流保障（streaming guarantee）。

请注意，这个保障与接收进程的语义无关：该进程是否确实处理收到的消息、是否作出反应，都不在保障之内。譬如说，进程 P 可能有个 bug，会导致它在收到 m_1 之后立即崩溃；而进程 Q 则希望 P 对自己发送的消息作出响应，并且两者已经建立了双向协议。在这种情况下，前面介绍的通信保障唯一能保证的事情是：如果 P 没有首先收到并处理 m_1，它绝不会收到（当然更不会处理）m_2。

当然，这里所说的“处理 m_1”需要在更广泛的意义上来理解：包括忽略 m_1、不把 m_1 从收件箱中取出（而是等 m_2 送达后先取出后者）、使用“选择性的”receive 语句等都应该算作对 m_1 进行了处理。

2.2 多方通信

在节点内通信的情况下，还可以对通信模式做更强的假设。Claessen 和 Svensson[CS05]给出了一个例子，三个进程（P_1、P_2和 P_3）通信如下：

- P_1 发送消息 m_1 给 P_2
- P_1 发送消息 m_2 给 P_3
- P_3 把消息 m_2 转发给 P_2

如果上述进程都位于同一节点，那么消息 m_1 必定会在 m_2 之前被送达 P_2，因为运行时系统只有在 P_1 完成发送之后才会把消息放入接收进程的收件箱。换句话说，只要各个进程都位于同一节点，应用协议就可以放心地假设 m_2 不会在 m_1 之前被送达 P_2；但同样的假设在 Erlang/OTP 将来的版本中可能会有问题[2]。

3. 跨节点编程

当 Erlang 程序涉及不同节点上的进程时，情况将迥异于单节点编程：不再有一个“大一统”的运行时系统了解当进程崩溃时作出明智决策所需的全部信息。在一个分布式、多节点的应用中，节点可能在地理上彼此远离，因此一般而言没有办法区分与远端进程（位于另一节点）的通信失败究竟是由于：（1）该节点与发送节点被暂时隔离（例如因为网络故障），还是（2）远端节点的运行时系统崩溃从而终止了其上的所有进程。为了找出（尽管有可能错误）发生故障的节点，Erlang 运行时系统会定期在节点之间发送消息（节拍信号，tick）。如果 N_1 节点没有收到来自 N_2 节点的 tick，那么 N_2 节点就会被认为已经崩溃，N_1 上所有“链接或者监视 N2 上进程”的进程就会收到错误通知。但这时的通信失败有可能只是暂时的：远端节点并未崩溃，稍后可以重新与之建立通信。

尽管分布情况下的通信实现与节点内编程时差异很大，但 Erlang 承诺通信本身并无不同。以下文字出自（老的那本）《Concurrent Programming in Erlang》：

> 消息可以被发送给远端进程，本地进程和远端进程之间也可以建立链接，这一切都好像所有进程都运行在同一个节点一样。远端 Pid 的另一个特点是：不论从语法上还是语义上，“向远端进程发送消息”都与“向本地进程发送消息”完全一样。这也就是说，发送给远端进程的消息始终以发送时的顺序送达，不会损坏也不会丢失。

4. 跨节点编程的陷阱

为了厘清 Erlang 的分布语义，我们做了一系列实验来验证这个“节点内编程与跨节点编程无差异”的承诺究竟在多大程度上有效。

作为总结，我们发现两处显著的差异，它们都会在跨节点编程中破坏流保障——这是节点内编程的基本通信保障。在某些情况下，先发送的消息会丢失，而后发送的消息则会顺利送达。也就是说，进程 Q 先后发送 m_1 和 m_2 两条消息给远端节点上的进程 P，却有可能发现只有 m_2 被送达，m_1 则丢

[2] 据我们推测，即便在当前的 Erlang/OTP 实现下，垃圾收集机制的存在也会使上述假设不能成立；但我们尚未实际观察到这样的情况。

失了。第一种可能出错的情况是遇到了重复的进程标识符（Pid）：在发送 m_1 的过程中进程 P 所在的节点可能崩溃，然后重启，此时 m_2 就可能被送达重启之后的节点上新生的进程。

第二种情况源自节点之间暂时性的通信故障，这会导致 Erlang 错误地判定节点已经崩溃，从而悄无声息地抛弃已经发送的消息。

在本节里，我们还会指出另外一些分布式通信的特质（陷阱 3 和陷阱 4），忽视它们也可能导致应用程序出错。

4.1 陷阱 1：重复的 Pid

进程标识符通常被认为是全局唯一的，但它们遵循同一个有穷的结构，因此有可能不是唯一的。不过只要这个有穷结构足够大，在实际应用中应该可以不用担心会用到重复的进程标识符。

所以，要是你知道在节点崩溃之后创建一个相同 Pid 的进程有多容易，你肯定会大吃一惊。在下面的例子中，我们将生成一个进程，确保将它弄死，然后再次与它通信！

我们用图 1 所示的 shell 脚本来运行这个例子：它一旦发现一个 Erlang 节点被杀死，就立即重启它。

```
#!/bin/sh
NODE=$1
while [ 1 -lt 2 ]; do
 erl -sname $NODE
 sleep 1
done
```

图 1. 代码示例：重启节点的脚本

这个例子（源代码如图 2 所示）的原理如下：首先在节点 N_1 上生成一个进程，它负责执行 run 函数启动实验。我们假设节点 N_2 已经用图 1 所示的脚本启动起来。N_1 上的进程首先注册退出陷阱（trapping exit），然后把下列指令执行三遍：在 N_2 上创建一个进程并链接到它，然后让它杀死节点 N_2（调用 halt 函数），最后等待 2 秒（以确保 N_2 重启）。这样一来，我们总共会生成三个进程，它们的 Pid 都被保存在 Pids 变量里。然后我们就等待三条退出消息，以确定所有在 N_2 节点（的不同实例）上生成的进程都已经被杀死了。最后我们再在 N_2 上生成一个进程，让它执行 echo 函数。随后，在 N_1 上调用 communicator 函数，并传入此前收集到的三个 Pid 作为参数。communicator 函数会尝试与已经死去的进程通信，而正如我们在图 3 所示的执行日志中看到的，其中一个被认为早已死去的进程竟然回复了！显然 N_2 上正在执行 echo 函数的那个新进程有着与之前某个进程相同的标识符。

```
-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).
```

图 2. 代码示例：Pid 被重复使用

```
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!
```

图 3. 输出：Pid 被重复使用

4.1.1 分析

显然当节点崩溃并重启之后，很快 Pid 就出现了重复。这样一来，在与远端进程通信时就不能那么放心地用 Pid 来唯一标识另一个进程了。因此在分布式协议中，人们往往会用别的方式来准确地标

识通信的另一端，例如给每个消息加上实例计数器，并在节点重启时递增这个计数器（这也是很多分布式协议中的标准做法）。

也许有些读者已经注意到了：我们在上面的例子中生成并杀死了三个进程，这个数字并非随便选择的。在节点内部有一个 incarnation 计数器，它的值可以是 1、2 和 3，重启超过 3 次则从头再来。

为预防此类问题，我们认为应该**强烈建议**对当前的 Erlang/OTP 实现加以修改，至少避免如此之快就重复使用 Pid（可以在 Pid 的结构中预留一块较大的空间用作节点重启计数器）。

4.2 陷阱 2：分布环境下被误解的基本通信保障

很多人对此坚信不疑：两个 Erlang 进程之间的信道绝不会悄无声息地丢失消息。在第二个例子中，我们将做一个实验：如果两个节点在通信过程中被断开、然后又重新连接，信道的可靠性就不那么可靠了。

图 4 的 Erlang 程序揭示了问题所在：这段程序在节点 N_1 上生成一个进程，它不断地发送递增的自然数；位于节点 N_2 的接收进程则只管把收到的数字打印出来。

```
-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).
```

图 4. 基本的 Erlang 通信

在一次运行之后，接收进程打印出的日志信息如图 5 所示。

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

```
got 148
...
```

图 5. 输出：进程间通信

在日志里我们看到，一开始数列如我们预期地递增，然后突然从 74 跳到了 146，中间的数都被丢掉了，这显然与“消息永不丢失”的特性不符。原因是什么？没什么，只是我们拔掉了其中一个节点的网线，等了一小会又重新插上了。这个操作具有时间敏感性：如果拔网线的时间太短（小于发送 tick 消息的间隔[3]），就可能不会有消息丢失。

显然如果节点间通信出现故障，我们也不能依赖普通的 TCP/IP 通道语义来传递消息。所以结论是：在分布式 Erlang 信道基础上构造分布式算法时，必须假设消息是可能悄无声息地丢失的；否则就得用代码来监视每一条进程间通信的信道，侦测所有可能的节点间通信故障（本文稍后会给出例子）。

研读更多 Erlang 文献之后就能发现，这种现象在 Barklund 和 Virdings 精心写就的 Erlang 的自然语言语义[BV99]中其实就已经被提到过了。引用原文如下：

> **10.6.2 信号的顺序**
>
> ……可以保证的是，如果进程 P_1 先后向进程 P_2 发出两个信号 S_1 和 S_2，那么信号 S_1 绝不会在 S_2 之后到达 P_2。这也就确保了只要有可能，发送给一个进程的信号最终应该抵达目的地。在有些情况下，无法要求所有信号都能抵达目的地，特别是向位于另一个节点的进程发送信号、并且节点间的通信暂时断开的时候。

请注意这里的讨论的“消息”是信号实例。换句话说，没有任何办法承诺信号在不改变顺序的前提下安全送达，尤其是当通信链路可能出现故障时。

4.2.1 分析

既然不能靠 TCP 通道语义来进行分布式通信，要想构建可靠的分布式应用程序，我们的选择也就不多了：

- 只采用对信息丢失不敏感的进程间通信协议。这个要求并不难满足，只要在每次通信中都携带整个协议的状态就行了；但在实际应用中这常常会给消息流量和时间造成过大的开销。另一种方案是给所有消息都加上顺序的计数器，并且抛弃（或者推迟发送）顺序不对的消息；此外这种方案还需要实现消息的重发，换句话说就是在 Erlang 分布机制的基础上重新实现一个具备 TCP 特质的应用协议。

- 在构造协议时假设：每次节点间通信失败时，都（可能）有不可恢复的通信故障发生。只要始终链接或者监视通信的远端进程，就可以侦测到这类故障。当故障发生时，任何来自该远端进程的消息都将不被信任，因为其中可能有消息丢失。也就是说，不再与可疑的进程做任何通信。

此外还有两种仅存在于我们设想中的替代方案，它们都要求改变 Erlang 的分布机制：

---

[3] 参见 Erlang 中关于设置 net_ticktime 的介绍。

- 实现一个有效的的 Erlang 传输协议，它具有类似于 TCP 的通信保障，同时能够从 TCP 通道失败中恢复（记录发送和接收的顺序号，以及发出但没有收到应答的信息）。也就是说，如果一条 TCP 通道出现故障、又新建了另一条通道，节点会记得哪些消息已经被发送、哪些尚未被接收，于是就可以进行重发。换句话说，在 Erlang 运行时系统内部实现一个具有 TCP 特质的分布式协议，取代现在的 Erlang 分布式协议。

- 禁止节点间重建连接。一旦两个节点之间的连接断掉，就不允许在它们之间重建连接。从效果上来说，这就要求（至少）其中之一重启。这看来似乎有些太过严酷，但对于某些应用场景来说会有所帮助。

4.2.2 改进后的消息传递保障

如果让进程 Q 来监视（不管是用链接还是监视器）接收消息的进程 P 的状态，就可以换种方式来表述 2.1 节谈到的流保障：

如果进程 Q 先后向进程 P 发送消息 m_1 和 m_2，并且将 Q 链接到（或者监视）P，并且 Q 没有收到来自 P 的退出消息[4]，那么就可以保证两条消息会按照发送顺序被送达。

如果 Q 收到来自 P 的退出消息，那么 P 有可能收到下列消息序列之一：P: ∈, <m1>, <m2>, <m1, m2>。

4.3 陷阱 3：更弱的多方通信保障

在 2.2 节我们已经看到，当所有进程都位于同一节点时，多方通信的行为是相当确定的，消息的顺序能够得到保障。不过大家知道，跨节点通信时情况就不再如此理想了。

- P_1 发送消息 m_1 给 P_2

- P_1 发送消息 m_2 给 P_3

- P_3 把消息 m_2 转发给 P_2

如果在上面的例子中三个进程分别位于不同的节点上，那么就有可能出现消息 m_2 在 m_1 之前被送达进程 P_2 的情况。

有趣的是，在目前的 Erlang 实现中，只要 P_2 和 P_3（或者 P_1 和 P_3，或者 P_1 和 P_2）位于同一节点，那么更强的通信保障（确保 m_1 在 m_2 之前送达 P_2）仍然有效。这是因为 Erlang 从根本上是用同一个流来处理一对节点之间的所有通信的（而不是针对每对进程通信创建一个专用的流），所以当上例中 m_1 被发送给 P_2 时，它一定会在 m_2 之前被处理和送达（除非出现之前讨论过的网络连接断开的情况），因为 m_1 和 m_2 是在同一个流上发送的，它们的顺序不会改变。不过，让应用程序的行为依赖于如此晦暗的保障，显然是颇可置疑的：Erlang 分布机制的实现稍有改动，就有可能改变这一行为特质。

[4] 等待退出消息多久取决于 net_ticktime 的设置。

4.4 陷阱 4：失败侦测器并不完美

在节点内编程时，如果一个进程死亡，与其链接或者对其监视的进程会得知这一情况。由于所有进程都在同一个运行时系统中运行，对终止进程的侦测当然是完美无缺的。换句话说，只要一个进程被报告已经终止，你就可以确信它再也不会出现。

但在跨越节点时，如果一个进程监视（或者链接到）另一个节点上的进程，远端进程就有可能在实际上并未死亡时谎报自己已经死亡。譬如说当两个节点被隔离（网络 tick 算法超时），就会导致远端被监视或者被链接的进程被报告为“已死亡”。但如果稍后节点间的连接又重新建立起来，这些已经被判定死亡的远端进程又可以进入通信了。

同样，只要允许在被隔离的节点之间重建连接，失败侦测器就不可能完美，因为一般而言不可能判断究竟是发生了网络连接故障（这种情况下可以重建连接）还是节点本身失败（不可恢复）。

5. 分布式应用编程原则

从图 4 和图 5 所示的例子可以看出，如果需要可靠的通信，我们就必须对进行通信的进程加以监管。并没有提供这一功能的内建机制，但我们可以在通用的链接和监视器基础上建立自己的编程原则。在本节中我们将展示这组编程原则的一个简化版本，从中可以看出重点所在。此外我们还将介绍一些理论上可行的对于 Erlang 分布机制的补充，我们认为这些扩充能让开发可靠的分布式应用变得更容易。

5.1 简单的编程原则

在图 6 所示的经过改进的例子中，我们增加了一些代码来确保消息不会丢失。（请注意：出错的可能性有千万种，这个例子只解决了通信故障的问题。）在例子中，发送消息的进程对接收消息的进程进行监视。一旦发送方收到错误信息，就停止发送消息，转入“同步模式”：继续监视接收方，直到接收方再次出现（在这里，我们重新插上网线）；然后两个进程开始同步，接收方把失去连接时的状态告知发送方，发送方则接着失去连接之前的状态继续发送消息。这个简单的方案存在缺陷（例如它所采用的半忙等待循环，以及发送方必须保存自己发送的所有东西），设计出更复杂的方案也是完全可能的。但这个例子足以阐明重点，在图 7 里我们看到了它运行的输出：尽管中间发生了通信故障，但传输序列并没有被破坏。

```
-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.
```

图 6. 基本的 Erlang 通信——有监视器的情况

```
...
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
...
```

图 7. 输出：进程间通信——有监视器的情况

5.2 扩展 Erlang 的可能性

既然我们已经看到分布式程序可能造成的各种麻烦，想象一下能对 Erlang 运行时系统做怎样的扩展会是一个有趣的思想实验。可以在几个层面进行修改，其中一种可能性是新增一组 Erlang API，

用于显式处理节点重连的情况，从而能够禁止或者允许（可能需要首先告知本地节点上的进程，让它们做好准备）重连。

另一种可能性是让进程在与远端节点通信时能够获得更多的信息。一个（在非分布式系统中）很常用的 OTP 构造就是监管结构（supervisor structure），如果在跨越几个运行时系统通信时能有类似的进程结构那就好了。不过，具体的实现方案有好几种。一个貌似合理的办法是用链接和监视器来构造一个节点监管结构，这样就可以根据事件（连接到节点，失去连接，重连，等等）指定不同的行为。这个办法主要的问题在于：当事件发生时，如何确保监管器是最先开始/停止通信的。另一个办法是给每个节点分配一个监管器，进程可以把自己注册在上面。这时当进程与位于另一节点的进程通信时，它就可以在节点监管器上注册这次通信，同时告诉监管器在错误出现时应该采取什么行动（例如崩溃、重启、忽略或者通知其他节点）。

还有一种对 Erlang API 较少侵入的扩展是：不仅在节点消失时发出通知（由 monitor_node 函数发出 nodedown 消息），而且还报告节点重连（或者重启）——让 monitor_node 也发出 nodeup 消息。

6. 结论

本文展示了一些实验的结果，其目的是帮助读者更好地理解 Erlang 的分布式行为。在研究 Erlang 的模型检查器（McErlang[FE06, FS07]）和分布式 Erlang 的精确语义[SF07]时，我们发现了一些从前——由于缺乏实验——没有发现的运行时系统的工作机制。简而言之，我们发现，在节点失去连接、稍后又恢复连接的情况下，文档中含糊的承诺与实际的运行时系统之间存在两处显著而有趣的差异：（1）Pid 很快被重复使用，（2）某些消息悄无声息地丢失了。

Pid 的重复使用问题可能主要是技术问题，但可能造成严重的后果，如果关键性的系统依赖于 Pid 的全局唯一性的话。为了预防问题发生，我们认为应该提议修改当前的 Erlang/OTP 实现，至少避免如此之快就重复使用 Pid（可以在 Pid 的结构中预留一块较大的空间用作节点重启计数器）。

经常被误解的关于分布式通信的保障则可能造成更多的困扰。这个问题源自更普遍的（并且众所周知难以解决的）网络分割问题。因此我们尝试给出一些安全处理类似情况的指导，并且考虑了如何修改/扩充 Erlang 以便更轻松地解决这一问题。

最后我们要说，尽管大部分应用程序不会受这些问题的困扰，但在开发大部分分布式应用时都应该将它们考虑在内。很可能你不需要写任何额外的代码来处理 Pid 重复之类的事情，也许你的应用协议根本就不在乎这些问题，但在作出决策时你应该清楚这些问题的存在，这是很重要的。

致谢

感谢 Joe Armstrong 对本文初稿提出了颇具价值的评注。

参考文献

[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.