Failure detectors in Erlang

Johan Montelius

October 2, 2016

Introduction

In this assignment you will look at how failure detectors work in Erlang. You will first set up a system of two processes running in one Erlang node and then divide the system so that the processes run in two nodes on different machines.

1 Send and receive

Let us start with a two simple processes, one that keeps sending ping messages every second and one that happily consume these messages. The producer will be started first and wait for a consumer to attach to it. Once the consumer has been connected the ping messages can be sent, each tagged with an increasing number. We should be able to stop the producer and it will then send a final message to the consumer.

1.1 the producer

The producer could look like this, we start by declaring the module and the interface.

```
-module(producer).

-export([start/1, stop/0, crash/0]).

start(Delay) ->
    Producer = spawn(fun() -> init(Delay) end),
    register(producer, Producer).

stop() ->
    producer ! stop.

crash() ->
    producer ! crash.
```

The function `start/1` will spawn the new process and register it under the name *producer*. It is given a parameter called *Delay*, that will be the number of milliseconds between messages. The function `stop/0` will simply send a stop message to the process registered under the name *producer*.

In the initialization of the producer we wait for a consumer to send us a message. A consumer will send us a message {`hello, Consumer`} where *Consumer* is the process identifier of the process that wants to receive the messages.

```
init(Delay) ->
     receive
         {hello, Consumer} ->
             producer(Consumer, 0, Delay);
         stop ->
             ok
     end.
```

The process is then implemented using the `after` construct that allows us to wait for a message a certain time before carrying on. If no stop message is received we send a *ping message* to the consumer.

```
producer(Consumer, N, Delay) ->
    receive
        stop ->
            Consumer ! bye;
        crash ->
            42/0  %% this will give you a warning, but it is ok
    after Delay ->
            Consumer ! {ping, N},
            producer(Consumer, N+1, Delay)
    end.
```

If we receive a *stop message* we send a *bye message* to the consumer and terminate the execution. This is the controlled way and let the consumer know that it should not expect to see any more messages. If we receive a *crash message* we simply terminate without informing the consumer. We will see that this of course leads to problems.

1.2 the consumer

The consumer is equally simple, it should have the following properties.

- It should export two functions: `start/1` that takes a process identifier or registered name of a producer as argument and `stop/` that terminates the process.
- When initialized it should send a *hello message* to the producer before entering its recursive state with a *expected value* set to 0.

- It should receive a sequence of *ping messages* and check that the message contains the *expected value*. If it is the correct value it should print the number on the screen and continue. If a higher value is received it should print a warning before continuing. In both cases the next expected value is one more than the received value.
- If the process receives a *bye message* (sent by the producer) or a *stop message* (sent when calling `stop/0`) the process should terminate.

2 Detecting a crash

Start the producer and then start the consumer in the same Erlang shell. The parameter to the consumer is simply `producer` since this is the local name under which it is registered.

You should be able to start the producer and then consumer and see how the messages are received by the consumer. If you *stop* the producer, the consumer is informed through a *bye messages* and terminates gracefully.

The problem is if we simulate a crash and the producer simply terminates. The consumer will now be stuck, waiting for a ping or bye message that will never come. This can be solved by implementing the consumer using a *after* construct. If no message has been received for ten seconds we can expect that the producer is dead and do something else. This is a rather crude way of solving the problem and there is a better way.

The Erlang runtime system can help us since it knows if the producer is alive or not. In Erlang we have a construct called *monitor* that asks the runtime system to give us information of the state of another process. To use this feature we start a monitor for the producer and add one additional clause to the recursive definition.

The monitor is started as follows:

```
init(Producer) ->
    Monitor = monitor(process, Producer),
    Producer ! {hello, self()},
    consumer(0, Monitor).
```

The *Monitor* that is returned from the call to *monitor/2* is simply a unique reference so that we can determine which monitor that was triggered.

Apart from now having one additional parameter, one more message is added to the recursive definition.

```
{'DOWN', Monitor, process, Object, Info}  ->
     io:format("~w died; ~w~n", [Object, Info]),
     consumer(N, Monitor);
```

This message is what we should receive if the producer crashes or terminates. We knows that the message pertains to the monitor that we started since the second element of the tuple is identical to our *Monitor*. The *Object* element is the process identifier (or registered name) of the dead process and the *Info* element gives us information in why it terminated.

You might ask why are we doing a recursive call and it is of course rather pointless. If the producer died we will not receive any more messages and we are then stuck in the same situations as before. We will however later use this feature to show some strange behaviour.

Run the producer and consumer and then crash the producer to see that it is working.

3 A two node experiment

Erlang is, as you know, quite transparent when it comes to distribution. We can quite easily run our producer and consumer in different Erlang nodes.

3.1 on the same host

Use two Erlang nodes, gold and silver, running on the same machine. Start them in distributed mode and make sure that you start hem with the same cookie so that they can communicate.

```
>erl -sname gold -setcookie foo
```

If the producer is started on the node *silver* the consumer should be given the argument {`producer, 'silver@host'`} where `host` is the name of the machine we are running.

No we can not only fake a crashing producer but kill the Erlang node that it is running in. What is the message that is given as reason when the node is killed, why?

3.2 a distributed experiment

Now things are getting interesting; run each of the nodes on separate machines. As first things should be as normal and we should be able to crash the process. What happens if we kill the Erlang node of the producer?

Now unplug the Ethernet cable to the producer and put it back in again after a few seconds. What happens? Remove the cable for longer time periods, what happens?

What does it mean that we have received a *DOWN message*? When should we trust it?

Have you received any messages out of order even without having received a *DOWN message*? What does the manual say about message delivery guarantees?

The questions raised are the core problems when implementing distributed systems.