Submission to PLDI 2017 Student Research Competition, Graduate Category

Safe Sessions for Erlang

Adrián Palacios

ACM member 3580926
Research advisor: Germán Vidal

MiST, DSIC, Universitat Politècnica de València
Camino de Vera, s/n, 46022 Valencia, Spain
apalacios@dsic.upv.es

Abstract

Erlang systems have become an example of fault-tolerant systems. Mainly, this fault-tolerance is achieved by following a simple strategy: If a process is about to crash, just let it crash and then restart it. Despite its simplicity, and without a proper design of the system architecture, this strategy can often lead to an inconsistent system state.

In this paper, we propose *safe sessions*, an automatic recovery strategy for Erlang. In safe sessions, concurrent dependencies between processes are registered, so that the system can return to a safe state (i.e., preserving causal consistency) in case of error.

1. Introduction

Erlang [1] is a concurrent language based on the actor model. Erlang programmers follow the *Let It Crash* philosophy, which discourages excessive error handling in programs, and whose advice is to let processes crash instead (and swiftly restart them after crashing). This is possible by building a process supervision tree structure where, if a process crashes, its supervisor process is able to detect it and restart the process.

However, the supervisor does not perform any kind of verification and, as a result, messages (or even processes) could appear somewhere in the system when they were not expected to be there—an inconsistent system state.

Recently, we introduced a reversible semantics [4] for Erlang, where processes record all the information required to ensure causal consistency [2] (i.e., an action may be undone only if every action caused by that action has not been executed or has been undone) when undoing some actions.

Due to the accuracy of this semantics, this information is continuously increasing, ever since the creation of a process. Therefore, the proposal from [4] is helpful to understand the problems in this setting, but is unfeasible in practice because of the introduced overhead.

As an alternative to [4], one could periodically store a *snapshot* (checkpoint) of the entire system state, and return to the snapshot in case of error. Again, this is not feasible in practice, since a system could be composed of thousands (or even millions) of processes at a given time.

Our proposal is a combination of both approaches. Here, the snapshots are taken at the process level, but we trace the process actions to ensure that we take the system back to a causally consistent state.

2. The language

Erlang [1] is a functional and concurrent programming language based on the actor model. In this model, each process has its local memory, and processes communicate through message-passing. Moreover, communication in Erlang is asynchronous (i.e., a process continues its computation after sending a message), causing the design of safe sessions to become more complicated.

In summary, the sequential part of Erlang is similar to most functional languages, and the concurrent actions of a process are the following:

- `spawn`: create a new process.
- `send`[1]: send a message to another process
- `receive`: suspend the execution of the process until a message from the mailbox matches any of the clauses (the message is consumed).

An example of Erlang computation can be seen in Figure 1, where *client*, *proxy* and *server* are separate processes. Here, *client* sends a request to *proxy*. After receiving the request, *proxy* forwards this request to *server*. Then, *server* performs the request and returns the result to *proxy* which, in turn, sends the result to *client*.

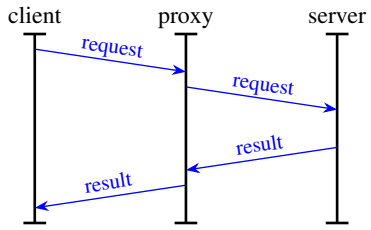

Figure 1. Example of computation in Erlang

Many things can go wrong in the example from Figure 1. For example, any of the messages can be lost, or an exception can be raised in any of the processes. Safe sessions can handle all kinds of error in a generic way, resetting the processes and trying again.

[1] In Erlang syntax: $Pid\,!\,Msg$. Here, Pid is the identifier of the destinatary process.

3. Safe sessions

We propose safe sessions as an alternative to the *Let It Crash* philosophy. Instead of letting a process to crash, we restore the process state back to a previous state, and let the process continue its execution from there.

To avoid storing too much information, we take a snapshot of the process at the beginning of a safe session, and we discard it once the session is finished. Hence, safe sessions are expected to be carried out during the evaluation of small blocks of code and, typically, these sessions will involve two or more processes that collaborate on some task.

In order to determine the start and the end of a session, we add the following construct to our language:

$$\mathsf{safetry}\ \ expr\ \ \mathsf{end}$$

We call this new construct a *safe block*. If the evaluation of $expr$ causes the process to crash, then the process should be restored with the information available in the snapshot.

Additionally, and this is more challenging, the processes that have been spawned or sent a message by this process (during the evaluation of $expr$) should be restored as well. The same applies to the processes spawned or sent a message by those processes, and so forth. This way, we ensure that the system returns to a causally consistent state.

Otherwise, if the evaluation of $expr$ is successful, the snapshots are discarded.

3.1 Safe computation

A sequential session—i.e., a session with no concurrent actions—is the trivial case of a safe session. If an error occurs, replacing the process state by the backup state is enough.

On a safe session with concurrent actions, restoring the backup state is not enough to ensure causal consistency [2]. In order to completely undo a process computation, we must also undo the computation caused by its concurrent actions.

Essentially, we only need to focus on spawn and send actions [4]. In our work, this is simplified by "propagating the safety" to the child processes and the receiving processes.

3.2 Implementation

For safe sessions to work, each Erlang process must have an associated monitor that intercepts both incoming and outgoing messages. Otherwise, it would not be possible to add any process to a safe session before the reception of a message from a process that is already in the session.

$$
\begin{array}{lcl}
[\![\mathsf{safetry}\ expr\ \mathsf{end}]\!]_M & \to & M\ !\ \langle \mathsf{start\_session}\rangle, \\
 & & [\![expr]\!]_M, \\
 & & M\ !\ \langle \mathsf{end\_session}\rangle \\
[\![\mathsf{spawn}(\ldots)]\!]_M & \to & M\ !\ \langle \mathsf{spawn}(\ldots)\rangle, \\
 & & \mathsf{receive}\ \langle \mathsf{spawn\_with}, P\rangle \to P\ \mathsf{end} \\
[\![\mathsf{self}()]\!]_M & \to & M \\
[\![Pid\ !\ expr]\!]_M & \to & M\ !\ \langle \mathsf{send}(Pid), [\![expr]\!]_M\rangle, \\
 & & \mathsf{receive}\ \langle \mathsf{sent\_as}, E\rangle \to E\ \mathsf{end} \\
[\![\mathsf{receive}\ clauses\ \mathsf{end}]\!]_M & \to & M\ !\ \langle \mathsf{receive}, clauses\rangle, \\
 & & Arg = \mathsf{receive} \\
 & & \quad \langle \mathsf{rec\_msg}, Msg\rangle \to Msg\ \mathsf{end}, \\
 & & \mathsf{case}\ Arg\ \mathsf{of}\ [\![clauses]\!]_M\ \mathsf{end}
\end{array}
$$

Figure 2. Program instrumentation of safe blocks

Monitors are expected to send signals between themselves to propagate the safety from their monitored processes.

In order to enable the interaction of process with their monitors, we must perform an instrumentation of the code within the safe blocks. The auxiliary function $[\![\ ]\!]$ executes this program instrumentation as it is shown in Figure 2.

For simplicity, we only show here the instrumentation for concurrent expressions. Here, M is the identifier of the monitor process, and the introduced variables (such as Arg) are supposed to be fresh.

Basically, all concurrent actions are replaced by queries to the monitor. For instance, if the process would spawn a new process, it sends a message to the monitor so that the monitor spawns the process (and its monitor) instead.

3.3 An example

Let us go back to our running example, and show how safe sessions work in Figure 3. Here, *client* enters a safe block before sending the request to *proxy*. When *server* receives the message from *proxy*, all the processes are safe (backup states are indicated by horizontal blue lines on the top).

In this case, the message from *proxy* to *client* is lost. When the monitor of *client* detects this, it sends a signal to the monitor of *proxy* (that will be propagated to *server*'s monitor) requesting their monitored processes to recover (blue arrows, processes go back to their backup state). Then, the monitor of *client* restores the state of its monitored process. Now, the computation is resumed before *client* sends the request to *proxy*, and the computation runs as expected (i.e., as in Figure 1).

Finally, when *client* receives the reply from *proxy* and exits the safe block, the backup states are dropped, and the session finishes.

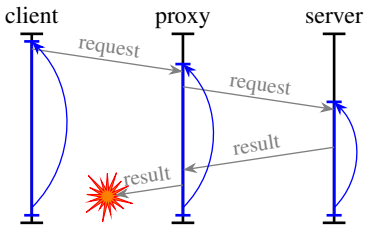

Figure 3. Example of safe session in Erlang

4. Related Work

Our proposal has some similarities with the checkpoint-based approach from [3], although they aim at defining a new language, rather than extending an existing one.

The aim of [5] is to prepare an interprocedural recovery strategy based on session types which determines the processes that must be restarted in case of error. Our proposal is more fine-grained, and it would allow us to define an intraprocedural recovery strategy in addition to the interprocedural one.

5. Conclusions and future work

In this paper, we have presented the basic aspects of an automatic technique for recovery in Erlang systems, and we have shown how it can be applied on a practical problem.

In the future, we will continue to refine our design of safe sessions and develop an implementation. Later, the implementation will be evaluated and compared against the aforementioned approaches.

This work is being developed in the context of COST Action IC1405 on Reversible Computation - extending horizons of computing.

References

[1] J. Armstrong, R. Virding, and M. Williams. Concurrent programming in Erlang (2nd edition). Prentice Hall, 1996.

[2] V. Danos and J. Krivine. Reversible communicating systems. In Proc. of CONCUR 2004, volume 3170 of LNCS, pages 292–307. Springer, 2004.

[3] J. Field and C. A. Varela. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In Proc. of POPL 2005, pages 195–208. ACM, 2005.

[4] N. Nishida, A. Palacios, and G. Vidal. Towards reversible computation in erlang. CoRR, abs/1608.05521, 2016. URL `http://arxiv.org/abs/1608.05521`.

[5] R. Neykova and N. Yoshida. Let it recover: Multiparty protocol-induced recovery. In Proc. of CC 2017, pages 98–108, ACM, 2017.