Modular Fault Handling in a Network-Transparent Programming Language

Géry Debongnie, Raphaël Collet, Sébastien Doeraene, Peter Van Roy
Université Catholique de Louvain
Louvain-la-Neuve, Belgium

***Abstract*—A programming language is network-transparent if the same program code executes with the same results, whether it is run in a centralized or distributed setting, provided there is no partial failure. The Erlang programming language is network-transparent and handle failures by message passing. We propose in this paper a generalization of the Erlang failure handling model which can be used for more expressive network-transparent languages.**

The new design introduces two concepts: entity fault states and fault streams. The failure of an entity is modeled in the system as a language entity, and is visible to the programmer via its corresponding fault stream. We also describe an implementation in Mozart 1.4.0. Evaluation shows that this model incurs a slight overhead in performance, but yields much more modular program code.

***Keywords*-network transparency; failure handling; fault stream**

I. Introduction

Distributed programming is often done by means of libraries at various levels of abstraction. For example, Java provides RMI and libraries that use RMI to provide higher levels of abstraction, such as JavaSpaces. In this paradigm, distribution is explicit, and issues regarding concurrency (such as failures) must be tightly coupled to functional code.

Other languages provide a different perspective: network transparency. A language is said to be network transparent if the same code can be used whether distribution is used or not, and if this code has the same semantics in both cases (at least, if no partial failures occur during the execution of the program, i.e., nodes and network links are assumed not to crash).

For example, the Erlang programming language, designed for distributed application, is network transparent. In the Erlang model, every concurrent computation is run in a process. Process are independent, and can communicate only by message-passing. The Erlang model addresses the issues regarding fault-tolerance with process linking. By default, when processes A and B are linked, if A dies, then B dies too. A process A can be set up as a system process. In this case, if B dies, then A receives a special message reporting the failure. Hence, the Erlang model uses only asynchronous failure handling. This proved to be practical, easy to use and robust.

In Erlang, partial failures are handled in a simple way: whenever a process crashes, messages are sent and the error is handled at some point, for example by restarting the process. It works because asynchronous failure notifications behave well with asynchronous remote invocations.

Contributions: In this paper, we explain how the Erlang failure handling model can be generalized for more expressive network-transparent languages. The new design introduces two concepts: entity fault states and fault streams (see section II). The failure of an entity is modeled in the system as a language entity, and is visible to the programmer via its corresponding fault stream.

As a proof of concept, these ideas have been implemented in the Oz programming language (a brief presentation of Oz can be found in section III), and a complete implementation of Erlang-like processes in Oz is shown in section IV.

II. Failure handling model

We use the word *site* to denote a computer used in a distributed system. An *entity* is any value that can appear in a programming language (for example, number, string, array, function, object, ...). A *distributed entity* is an entity used in the context of a distributed system. Some or all of the sites may have a reference on it.

Two simple ideas lie at the core of our failure handling model: fault states and fault streams. Each site monitors every distributed entity that it knows of, and whenever the fault state changes, it updates a (local) fault stream with the newest state. That fault stream is a language entity that can be accessed by the programmer to handle failure in the most appropriate way.

A. Entity Fault States

Every site that has a reference on an entity uses failure detectors to monitor it. From the point of view of a site, any language entity can be in one of the following state:

- `ok` means that the entity is not suspected by any of the basic detectors,
- `tempFail` means that the site is temporarily unable to perform an operation on the entity,
- `localFail` means that the entity has permanently failed on this site, but it might still be available on other sites,
- `permFail` means that the entity has permanently crashed on all sites.

The local fault state for the entity is a local view of the global fault state. Every possible fault state transitions is depicted in Figure 1. The state changes when the failure

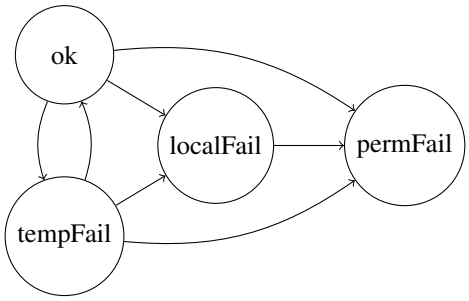

Figure 1. Local fault state of an entity

detectors suspect a problem, or receive a heartbeat. When the state changes, the detector sends a notification with the new fault state to the program.

This simple model provides the programmer with necessary information to reason about failures and handle them adequately. Network communications can be unreliable, so it is quite natural to make a distinction between `tempFail` and `permFail`.

B. Fault Stream

As explained previously, both temporary and permanent failures are modeled with fault states. This fault state is reified in the language as a fault stream, i.e., a growing list of state transitions. This fault stream can be read by a separate thread, which thus monitors the entity.

Besides, any operation on a failed distributed entity blocks until the entity is `ok` again, or forever if it has permanently failed. This prevents distribution errors to inject distribution-related behavior in the program (e.g., exceptions), which in turn prevents failures to break modularity. Note that it allows a clean separation of distribution-related concerns from the functional program.

In our Oz implementation, we can fetch the fault stream of an entity x with the primitive `GetFaultStream`: `S = {GetFaultStream x}`. `S` becomes a stream of the successive fault states of x. The stream is closed when the entity disappears from the memory.

Figure 2 illustrates a failure handler that displays the successive fault states of an entity on the standard output. It is simply a thread waiting for state transitions in the fault stream of `E`. Each state transition is then displayed until the fault stream is closed.

III. Distributed programming in Mozart/Oz

Mozart is an implementation of the multi-paradigm programming language Oz. This section explains the most relevant features of its distributed subsystem (Mozart/DSS).

Entities: Mozart has three kind of entities: stateless entities (numbers, records, atoms, procedures, ...), single assignment entities (unbound variable) and stateful entities (cells and ports).

```
thread
   for State in {GetFaultStream E} do
     T = case State
         of ok         then "entity is fine"
         [] tempFail   then "temporary fail"
         [] localFail  then "failed locally"
         [] permFail   then "failed globally"
         end
   in
     {Show T}
   end
   {Show "entity disappeared"}
end
```

Figure 2. A thread that prints messages when entity `E`'s fault state changes

Dataflow behaviour: When the value of an unbound variable is needed, the virtual machine stops and wait until it is bound[1] (for example, in a different thread, or in a different site). It is a nice behaviour when dealing with asynchronous calls.

Network transparency: Any entity in Mozart-Oz can be distributed, one needs only to get a reference and the virtual machine will transparently convert any operations into the appropriate network protocol operations, in such a way that the distinction is not visible at the language level. By default, Mozart/DSS uses the following distribution protocols:

- stateless entities: the value is sent together with the reference (so it is replicated on every site which has a reference to it).
- single assignment entities: the binding is done on the first site that receives the reference.
- stateful entities: the state of the entity migrates from one site to another, and operations are executed locally.

Network awareness: The default distribution protocol might not be optimal for a given situation. So, there is a need for the programmer to be able to control somehow the distribution behaviour (network awareness). This is done by using annotations. Annotations, which are simple statements such as `{Annotate x p}`, do not change the semantic of the program (if there are no failures). They only change the network protocol used to distribute an entity.

IV. Erlang model

It is possible to implement Erlang-like processes using the model presented in this paper. Message-passing is implemented using *ports*, and monitoring links are made up of asynchronous failure handlers.

A. Erlang-Like Process without Failure Handling

First, let us ignore failures. In that simple case, Figure 3 shows how to spawn a simple Erlang-like process in Oz.

[1]Using this mechanism with stateless entities is actually a simple multithreaded programming model.

```
%% spawn a process with procedure Process
fun {Spawn Process}
   Xs Ys
   Self={NewPort Xs}
   fun {Loop Xs}
      case Xs of user(M)|Xt then
         M|{Loop Xt}
      end
   end
in
   thread Ys={Loop Xs} end
   thread {Process Ys} end
   Self
end

%% send message M to process A
proc {SendMsg A M}
   {Send A user(M)}
end
```

Figure 3. Erlang-like processes without failures

The function `Spawn` takes an unary procedure as input and return a port: `Self` (last line of the definition). Any message `M` sent to the port `Self` by the procedure `SendMsg` is wrapped in a record `user(M)`. One sent, it is put into the stream `Xs`, then unwrapped by the function `Loop`. This makes a new stream `Ys` which is finally given as input to the procedure `Spawn`. The statements **thread** ... **end** creating threads are required, because otherwise the function `Spawn` would wait forever.

In Erlang, the statement `receive` is very specific, and non declarative by nature. In this model, we can choose to handle the stream in a declarative way, if this is suitable.

This definition, however, does not allow processes to be linked, nor does it handle failures.

B. Erlang-Like Process with Failure Handling

A complete implementation of Erlang-like process can be found in Figure 4. In this implementation, the `Loop` procedure maintains the administration of the process. The variables `Linkset` and `Sys` encode respectively the set of linked processes and a flag telling if the process is a system process. The pattern matching statement forwards user messages to the user process. It takes care of administrative messages.

`process` and `link` messages offer a way to control the administrative variables `Linkset` and `Sys`. `exit` messages are sent by the `Notify` procedure, or by `Monitor` for processes that would not be able to notify properly (e.g., because of a crash). Note that `Monitor` is an asynchronous failure handler reading the fault stream of the linked process. It is started when the process gets linked to another process.

Note also that abnormal termination of a linked process kills a non-system process. For a system process, the process is notified by a normal message.

```
%% link process Self to process A
proc {Link Self A}
   {Send Self link(A)} {Send A link(Self)}
end

%% change the 'system process' flag
proc {SetSystem Self B}
   X in {Send Self system(B X)} {Wait X}
end

%% spawn a process with procedure Process
fun {Spawn Process}
   Xs Ys Self={NewPort Xs} T
   fun {Loop Xs Linkset Sys}
      case Xs of X|Xt then
         case X
         of user(M) then
            M|{Loop Xt Linkset Sys}
         [] system(B X) then
            X=unit {Loop Xt Linkset B}
         [] link(A) then
            {Monitor A}
            {Loop Xt A|Linkset Sys}
         [] exit(E) then
            {Notify E Linkset} nil
         [] exit(A E) andthen Sys then
            X|{Loop Xt Linkset Sys}
         [] exit(A normal) then
            {Loop Xt Linkset Sys}
         [] exit(A E) then
            {Kill T} {Notify E Linkset} nil
         end
      end
   end
   proc {Monitor A}
      thread
         if {Member permFail
             {GetFaultStream A}} then
            {Send Self exit(A crashed)}
         end
      end
   end
   proc {Notify E Linkset}
      for A in Linkset do
         {Send A exit(Self E)}
      end
   end
in
   thread Ys={Loop Xs nil false} end
   thread
      T={Thread.this}
      try
         {Process Ys}
         {Send Self exit(normal)}
      catch E then
         {Send Self exit(E)}
      end
   end
   Self
end
```

Figure 4. Erlang-like processes with failure handling

V. Implementation

The model presented in this paper has been successfully implemented in the Mozart Oz programming system, and is known as Mozart/DSS. This implementation is the result of work from several developers, including Erik Klintskog, Zacharias El Banna, Boris Mejías and Raphaël Collet [8], [9].

One might be concerned about the performance aspect of such a model. The mechanisms required for monitoring each entity are not free. To answer that question, several experiments have been performed on Mozart/DSS (see [2]). They show that Mozart/DSS is 12% slower than Mozart for the same program. The measure- ment is done on CPU time only, i.e., network delays are ignored. This overhead is incurred by the new distribution subsystem, because of the fault stream man- agement.

VI. Conclusion

We present in this paper a model for fault-tolerant network-transparent distribution. In this model, failure handling is done exclusively asynchronously. An asynchronous failure handler executes in its own thread, and listens to fail state changes for an entity. Meanwhile, a statement requiring that an entity be correct will wait instead of throwing an exception. This model has many advantages compared do a traditional approach:

- It is more flexible: instead of having two states for an entity (dead or alive), our model is more granular. The different states of an entity (`ok`, `localFail`, `tempFail`, `permFail`) allow the programmer to take the appropriate decision at the correct level of abstraction. This is much more flexible than timeouts.
- It is asynchronous: distributed programs have to take into account network latency. Also, a remote request might take some time to complete. It is therefore better to handle failure asynchronously. It means that message are sent whenever needed and processes can continue to accomplish their task as soon as possible, instead of waiting.
- It is modular: clearly, this is the most important aspect. It allows for the code managing the failure of an entity to stay close to the place where the entity is created. It is a key point in separating functionality from failure handling.

These ideas have been implemented in Distributed Oz. The resulting system is network transparent, network aware, and handle failures modularly. Performance is sufficient to program large scale distributed systems, as done in [3], where a transactional peer-to-peer storage system, Beernet is implemented in Distributed Oz.

As a proof of the expressiveness of the model, Section IV shows a full implementation of the Erlang model. Our model thus generalizes the Erlang model, which can be encoded as a particular strategy, but is not limited to it. Other fault handling strategies can be programmed if needed.

References

[1] C. Cachin, R. Guerraoui, L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*, (2. ed.). Springer 2011.

[2] R. Collet, *The Limits of Network Transparency in a Distributed Programming Language*, Phd Thesis UCL, 2007.

[3] B. Mejías, *Beernet: A Relaxed Approach to the Design of Scalable Systems with Self-Managing Behaviour and Transactional Robust Storage*, Phd Thesis UCL, 2010.

[4] P. Van Roy, S. Haridi, *Concepts, Techniques, and Models of Computer Programming*, MIT Press, 2004.

[5] P. Van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, R. Scheidhauer, *Mobile Objects in Distributed Oz*, ACM Transactions on Programming Languages and Systems (TOPLAS), Sep. 1997, pp. 804-851.

[6] S. Haridi, P. Van Roy, P. Brand, M. Mehl, R. Scheidhauer, and G. Smolka, *Efficient Logic Variables for Distributed Computing*, ACM Transactions on Programming Languages and Systems (TOPLAS), May 1999, pp. 569-626.

[7] J. Armstrong, *Making reliable distributed systems in the presence of software errors*, Phd Thesis, Royal Institute of Technology, 2003.

[8] E. Klintskog, Z. El Banna, P. Brand, S. Haridi, *The design and evaluation of a middleware library for distribution of language entities*, in Vijay A. Saraswat, editor, *ASIAN*, volume 2896 of *Lecture Notes in Computer Science*, pp. 243–259. Springer, 2003.

[9] E. Klintskog, *Generic Distribution Support for Programming Systems*, Phd Thesis SICS, 2005.