Failure Handling in a Network-Transparent Distributed Programming Language

Raphaël Collet and Peter Van Roy

Université catholique de Louvain,
B-1348 Louvain-la-Neuve, Belgium
{raph, pvr}@info.ucl.ac.be

Abstract. This paper shows that asynchronous fault detection is a practical way to reflect partial failure in a network-transparent distributed programming language. In the network-transparency approach, a program can be distributed over many sites without changing its source code. The semantics of the program's execution does not depend on how the program is distributed. We have experimented with various mechanisms for detecting and handling faults from within the language Oz. We present a new programming model that is based on asynchronous fault detection, is more consistent with the network-transparent nature of Oz, and improves the modularity of failure handling at the same time.

1 Introduction

A network-transparent programming language tries as much as possible to make distributed execution look like centralized execution. This illusion cannot be complete, because distributed execution introduces new elements that do not exist in centralized execution, namely latency and limited bandwidth between sites, partial failure, resources localized on sites, and security issues due to multiple users and security domains. In our view, these new elements should not be considered as making network transparency an undesirable or unrealistic goal. On the contrary, we consider that network transparency can be a realistic approximation that greatly simplifies distributed programming, and that it should be part of the design of a distributed programming language. We find that a network-transparent implementation can be practical if it starts from an appropriate base language. The execution model of the base language is crucial, e.g., Oz is an appropriate choice [1] but Java is not [2].

This paper focuses on one part of network transparency, namely failure handling. We propose a solution to the issue of reflecting partial failures in the language that provides a way to build fault-tolerance abstractions in the language, while maintaining transparency and separation of concerns.

1.1 Context of the Paper

This work is done in the context of the Distributed Oz project, which is a long-term research project whose aim is to simplify distributed programming. This

C. Dony et al. (Eds.):Exception Handling, LNCS 4119, pp. 121–140, 2006.

project started in 1995 with the goal of making a distributed implementation of the Oz language that is both network transparent and network aware [1,3]. That is, our main goal is to *separate* the distribution aspect from the functionality of the program. This was implemented in the Mozart Programming System, which was first released in 1999 [4].

In this system, an application is composed of several *sites* (i.e., system processes) that virtually share language entities. All language entities are implemented with distributed protocols that respect the language semantics in the case of no site failures. The difference between the protocols is in their network behavior. For example, objects were implemented with both a stationary protocol and a mobile (cached state) protocol. Single-assignment entities (dataflow variables) were implemented with a distributed binding protocol (that in its full generality is an implementation of distributed unification). These protocols were designed with a well-defined fault behavior, in the case of site failures, and the fault behavior was reflected in the language through a fault module.

The site and network faults are reflected as *data failures*, depending on how the faults affect the proper functioning of the data. For instance, a stationary object fails when the site holding its state crashes. The original fault module provided two ways to reflect failures in the language: a synchronous detection and an asynchronous detection. The present paper proposes a new model for reflecting partial failure based on our experience with this design.

1.2 Synchronous and Asynchronous Failure Handling

We make a clear distinction between two basic ways of handling entity failures, namely *synchronous* and *asynchronous* handlers. As we shall see, asynchronous failure handling is preferable to synchronous failure handling. A *synchronous* failure handler is executed in place of a statement that attempts to perform an operation on a failed entity. In other words, the failure handling of an entity is synchronized with the use of that entity in the program. Raising an exception is one possibility: the failure handler simply raises an exception. In contrast, an *asynchronous* failure handler is triggered by a change in the fault state of the entity. The handler is executed in its own thread. One could call it a "failure listener". It is up to the programmer to synchronize with the rest of the program, if that is required.

The following rules give small step semantics for both kinds of handlers. The symbol σ represents the store, i.e., the memory of the program. The system reflects the instantaneous fault state of an entity in the store through a system-defined function $\mathsf{fstate}(x)$, which gives the fault state of x. Each execution rule shows on its left side a statement and the store before execution, and on the right side the result of one execution step. Rule *(sync)* states that a statement S can be replaced by a handler H if the fault state of entity x is not ok, i.e., if x has failed. Rule *(async)* spawns a new thread running handler H whenever the fault state of x changes. Note that there may be more than one handler on x; we assume all handlers are run when the fault state changes.

$$\frac{S \,\|\!|\, H}{\sigma \,\|\!|\, \sigma} \quad \text{if} \quad \begin{array}{l}\text{statement } S \text{ uses entity } x \\ \text{and } \sigma \models \mathsf{fstate}(x) \neq ok\end{array} \qquad \text{(sync)}$$

$$\frac{\qquad\qquad \|\!|\; H}{\sigma \wedge \mathsf{fstate}(x){=}fs \,\|\!|\, \sigma \wedge \mathsf{fstate}(x){=}fs'} \quad \text{if} \quad fs \rightarrow fs' \text{ is valid} \qquad \text{(async)}$$

Synchronous failure handlers are natural in single-threaded programs because they follow the structure of the program. Exceptions are handy in this case because the failures can be handled at the right level of abstraction. But the failure modes can become very complex in a highly concurrent application. Such applications are common in Oz and they are becoming more common in other languages as well. Because of the various kinds of entities and distribution protocols, there are many more interaction schemes than the usual client-server scheme. Handlers for the same entity may exist in many threads at the same time, and those threads must be coordinated to recover from the failure.

All this conspires to make fault tolerance complicated to program if based on synchronous failure handling. This mechanism was in fact never used by Oz programmers developing robust distributed applications [5]. Instead, programmers relied on the asynchronous handler mechanism to implement fault-tolerant abstractions. One such abstraction is the "GlobalStore", a fault-tolerant transactional replicated object store designed and implemented by Iliès Alouini and Mostafa Al-Metwally [6].

1.3 Structure of the Paper

The present paper introduces a model based on asynchronous failure handling and shows how programming fault tolerance is simplified with this model. Section 2 explains the distributed programming model of Oz. Section 3 presents the design of a new fault module that takes our experience building fault-tolerance abstractions into account. Section 4 gives a detailed example of a group communication abstraction that shows minimal interaction between the failure handling and the abstraction's main functionality. Section 5 describes the implementation of the fault module. Finally, Sect. 6 explains the lessons we learned and Sect. 7 compares with related work.

2 The Programming Model of Oz

This section gives an overview of Oz as a programming language and its extension to distributed programming. We discuss an important property of the latter extension, namely the *network transparency*, which is convenient for separating distribution concerns from functional concerns of a program [3].

Oz is a high-level general-purpose programming language that supports declarative programming, object-oriented programming, and fine-grained concurrency as part of a coherent whole. It is dynamically typed and supports *multiparadigm* programming in a natural way. The Mozart Programming System implements the language and provides the support for its distributed implementation [4].

```
local X Y in
   thread X={Pow 2 100} end      % computes 2^100
   thread Y={Pow 5 10} end       % computes 5^10
   {Show X+Y}                    % blocks until X and Y are known
end
```

Fig. 1. An example of dataflow synchronization

To understand Oz and its distribution model, it is important to keep in mind that entities in Oz are classified into three kinds: stateless (including numbers, records, and procedures), stateful (cells and ports, see below), and single assignment (dataflow logic variables, see below).

2.1 Dataflow Concurrency

We briefly give the ideas underlying the Oz execution model. Oz can be defined by a process calculus based on concurrent constraints [7,8]. It provides lightweight threads and dataflow logic variables [9]. A logic variable is a placeholder for a value. Upon creation, the variable's value is unknown. The variable can be assigned at most once to a value, thanks to a unification mechanism. Note that unification is monotonic and there is no backtracking. A unification that fails simply raises an exception (see below).

A thread is created by an explicit statement **thread** S **end**, where S is the statement to be executed by the new thread. Threads communicate with each other by sharing logic variables, stateless entities (values), and stateful values (such as objects). A thread that attempts to use a variable's value automatically *blocks* if that variable is not bound yet. Once the variable is bound to a value, all threads blocking on that variable become runnable again. This synchronization mechanism is called *dataflow*. An example is shown in Fig. 1. We note that dataflow in Oz is monotonic (at most one token can appear on an input), whereas classic dataflow is nonmonotonic (new tokens can appear on an input).

2.2 Stateful Entities

Oz comes with a set of stateful entities that have a well-defined behavior in the presence of concurrency. The most primitive of them is the *cell*, which is a simple mutable pointer with an atomic value exchange operation. The first part of Fig. 2 shows the main operations on a cell. The exchange operation is provided as the multifix operator x=y:=z. In the example, the variable J is put in C, and the statement resumes by unifying I with the former contents of C. The new value of C is then determined. The last two lines of the example implement a thread-safe atomic increment of a counter.

Another important stateful entity is the *port*. The port defines a simple and efficient message-passing interface. The second part of Fig. 2 gives the typical use of a port. The *stream* S is a potentially infinite list that is built incrementally,

```
local C I J in
   C={NewCell 42}      % create a new cell C with contents 42
   {Show @C}           % print the contents of C (42)
   C:=7                % assign 7 to C
   I=C:=J              % assign J to C, and unify I with 7
   J=I+1               % bind J to I+1=8
end

local S P in
   P={NewPort S}       % create a port with stream S
   thread              % print every element appearing on S
      for X in S do {Show X} end
   end
   {Send P foo}        % send foo on P, S becomes foo|_
   {Send P bar}        % send bar on P, S becomes foo|bar|_
end
```

Fig. 2. Examples showing the cell and port entities

and whose elements are the messages sent to the port. In order to receive the messages, one simply has to read the list `S`. A list is either the empty list `nil`, or a pair $x|t$, where x is the head element, and the tail t is also a list. The dataflow synchronization automatically wakes up the threads reading the stream when new messages arrive.

2.3 Exceptions and Failed Values

The language provides a classical, thread-based exception mechanism. The block construct **try**...**catch**...**end** works as in most languages. When an exception is raised, all following statements are skipped until the closest **catch** delimiter. The exception is then handled by the code that follows the matching **catch**. Threads have no default handler, so uncaught exceptions are programming errors, which make the whole program fail.

An example is shown in Fig. 3. The keyword **fun** defines a new function. Notice that the exception value is the *record* `divisionByZero(Y)`, and that the **catch** construct supports pattern matching.

Threads are independent of each other, and an exception can only be caught *within* the thread where it was raised. In order to propagate exceptions from thread to thread, we extend the basic model with *failed values*. A failed value is a special value that encapsulates an exception. A thread that attempts to use that value automatically raises the exception.

This model fits well with functional style programming. Suppose that a thread T computes some value, and binds a variable `X` to that result. That variable may be shared by other threads that are interested in the result. If the computation in T raises an exception `E`, the latter is caught, and `X` is bound to a failed value

```
fun {Divide X Y}
   if Y==0
   then raise divisionByZero(Y) end      % throw exception
   else X div Y                          % return result
   end
end

try
   {Show {Divide 42 0}}
catch divisionByZero(Y) then             % match exception
   {ShowError "error: division by zero"}
end
```

Fig. 3. An example of exception handling

```
fun {ConcurrentDivide X Y}
   thread % return either the result, or a failed value
      try {Divide X Y} catch E then {FailedValue E} end
   end
end

try
   I={ConcurrentDivide 42 J}   % (1)
   J={ConcurrentDivide 7 13}   % (2)
in
   {Show I+1}       % blocks on I, which will raise an exception
catch E then
   {ShowError E}
end
```

Fig. 4. An example of exception passing between threads

containing `E`. Other threads trying to use `X` automatically raise the exception. An example is shown in Fig. 4. The statement `(1)` spawns a thread that blocks until `J` is known. Statement `(2)` spawns a thread that computes `J`, which is determined to be zero. Once this is known, `I` is bound to a failed value. The expression `I+1` then raises the exception contained in `I`, which is caught in the main thread.

Failed values are strongly motivated by lazy computations in Oz. We model a lazy computation as a thread that waits until a result variable becomes *needed*. Another thread that blocks on the variable automatically makes it needed, which wakes up the lazy thread. A failed value allows to propagate the exception without forcing the user to add extra tests on the result.

2.4 Distribution Model

The distribution model of Oz allows several *sites* to share language entities. A site is simply a system process, and sites can be spread among a network of computers. The model gives all sites the illusion of a shared memory, with reference integrity. Well-chosen protocols implement the semantics of entity operations [10,11,1]. The power of the model is that it clearly distinguishes between the protocols for *stateless*, *stateful*, and *single-assignment* entities. The distribution strategies and implementation are chosen to be appropriate for each category.

Stateless entities, e.g., atomic values (numbers, literals), records, and procedures, are copied between sites that share them. Entities whose equality is referential (e.g., code) are given a globally unique identity, which ensures their referential integrity. Entities with structural equality (like integers and records) can be copied at will.

Stateful entities are given a global identity and use specific protocols to ensure the consistency of their state. For instance, ports use a stationary state. A stationary state requires each read/write operation to send a message to the state's home site. On the other hand, objects can use a mobile state [11]. The migratory protocol ensures that the state migrates where the operations are attempted. Once the state arrives on a site, a batch of operations can be performed locally without extra overhead. The state behaves like a cache. Protocols for a replicated state are also provided.

Single-assignment entities, i.e., logic variables, are implemented by a distributed unification algorithm [10]. Among the sites sharing a given variable, one of them is responsible for determining the final binding of the variable. Other sites that want to bind it send a message to that site, which propagates the binding to the other sites. The algorithm ensures the unicity of the binding, and the absence of cycles (when several variables are bound to each other).

In general, distributed entity references are acquired by transitivity. A bootstrapping mechanism allows a site to create a *ticket*, which is a public reference to an entity. The ticket is a character string that has the syntax of a URL, and can be transmitted by any other means (web page, email). The receiver program uses that ticket to retrieve the entity, and share it with its provider.

2.5 Network Transparency and Network Awareness

The distribution model has two important aspects: it is both *transparent* and *aware* with respect to the network. While these may look contradictory, they are in fact complementary. Let us give a definition for each one.

- *Network Transparency.* This property states that the semantics of a distributed entity is the same as if it were purely local. Primitive operations on that entity return the same results as if the whole computation was in the same address space. In other words, a programmer may reason about the functionality of a program without taking distribution into account.

– *Network Awareness.* This aspect gives the programmer some control over the non-functional behavior of the entity. For instance, different strategies for distributing a stateful entity will have different performances and robustness, depending on how they are used by the application.

A programmer cannot write a program without *ever* taking distribution into account. He or she should decide at some point where the various pieces of the code will be run, and which entities will be distributed among sites. But the network transparency will favor a separation of concerns, where the application's functionality is as independent as possible from its non-functional properties.

3 The Fault Model

This section proposes a language-level fault model that is compatible with network transparency. The model defines how site and network failures are reflected in the language. Because a failure may affect the proper functioning of a distributed entity, failures are reflected at the level of entities. Here are the principles defining our model, each being described in the corresponding subsection below.

1. Each site assigns a local *fault state* to each entity, which reflects the site's knowledge about the entity.
2. There is no synchronous failure handler. A thread attempting to use a failed entity blocks until the failure possibly goes away. In particular, no exception is raised because of the failure.
3. Each site provides a *fault stream* for each entity, which reifies the history of fault states of that entity. Asynchronous failure handlers are programmed with this stream.
4. Some fault states can be enforced by the user. In particular, a program may provoke a global failure for an entity.

This fault model is an evolution of the first fault model of Oz, and integrates parts of another proposal [5]. A comparison between the latter and this proposal is given in Sect. 6.

3.1 Fault States

First, each site defines a current *fault state* for each entity, which reflects the local knowledge of the system about the entity's global state. This implies that a given entity may have different fault states on different sites. We define four different fault states: `ok`, `tempFail`, `localFail`, and `permFail`. Their semantics are given below, and the arrows on the left show the valid state transitions.

`ok`	The entity behaves normally.
↓↑	
`tempFail`	The entity is currently unavailable locally. This is typically triggered by a communication timeout. It can be temporary (hence the name), or might hide a permanent failure. Basically the actual status of the entity is unknown.
↓	
`localFail`	The entity is permanently unavailable locally. This state reflects the fact that the site considers the entity to be failed, whatever its actual global status is. This state can always be enforced by the program.
↓	
`permFail`	The entity is permanently failed on a global scale. This final state comes with a strict guarantee: the entity will never recover. Usually this kind of diagnosis can only be performed on a LAN. We have extended the use of `permFail` by allowing an application to explicitly cause an entity to fail permanently. This allows an application to use `permFail` as a way to communicate between different parts of itself.

The absence of some state transitions is intensional. An entity going from `ok` to `permFail` will have to step through states `tempFail` and `localFail` before entering `permFail`. This simplifies the monitoring of the fault state of an entity, since observing the state `localFail` means that the state has reached `localFail` at least. This will become clear with the fault stream below.

Our experience shows that this simple model is in fact sufficient in practice. One can program abstractions in the language to improve the failure detectors by using local observations in a global consensus algorithm, for instance.

As the reader may guess, fault states are related to how the distribution of an entity is implemented. The more sophisticated the distribution's implementation, the more complicated the fault model. In this paper, we favor a simple fault model, therefore keeping the implementation simple. The programmer should be able to reason easily about the properties of the distribution. Complex fault-tolerant abstractions should be built at the higher user level, not at the low level.

The original fault model of Mozart was much more complex. It tried to extract the maximum information that could be deduced efficiently about failed entities [12]. Our experience with the original model showed that this extra information was in fact never used. We conclude that a simple model (such as defined by the present paper) is sufficient in practice.

3.2 No Synchronous Failure Handler

When the fault state of a given entity is not `ok`, operations on that entity have few chances to succeed. Raising an exception in that case might look reasonable, but our experience suggested that it is not. The main reason is that it breaks the transparency: such an exception would never occur if the application was

not distributed. This is even worse for asynchronous operations, like sending a message on a port. Such operations should succeed immediately. With exceptions, the programmer cannot write a program without taking distribution into account from the start.

Another reason that exceptions are inadequate is because the exception mechanism assumes that you can *confine* the error. A partial failure in a distributed system can hardly be kept confined. Handling a distributed failure often requires some global action in the program. Moreover, because of the highly concurrent nature of Oz, the failure may affect many threads on a single site. Having many failure handlers for a single entity on a given site introduces too much complexity in the program.

In order to keep the network transparency, an operation on a failed entity simply blocks until the entity's fault state becomes `ok` again. The operation naturally resumes if the failure proves to be temporary. It will suspend forever if the failure is permanent (`localFail` or `permFail`).

3.3 Fault Stream

We propose a simple mechanism to monitor an entity's fault state and take action upon a state change. On each site, each entity is associated with a *fault stream*[1], which reflects the history of the fault states of the entity. The system maintains the *current* fault stream, which is a list $fs|s$, where fs is the current fault state, and s is an unbound variable. The semantic rule

$$\frac{}{\sigma \wedge \mathsf{fstream}(x){=}fs|s \,\|\, \sigma \wedge s{=}fs'|s' \wedge \mathsf{fstream}(x){=}fs'|s'} \quad \text{if} \quad fs \to fs' \text{ is valid} \quad (1)$$

reflects how the system updates the fault state to fs'. The dataflow synchronization mechanism wakes up every thread blocked on s, which is bound to $fs'|s'$. An asynchronous handler can thus observe the new fault state.

The fault stream of an entity reifies the history of fault states of that entity. Moreover it transforms the nonmonotonic changes of a fault state into monotonic changes in a stream. It provides an almost declarative interface to the fault state maintained by the system.

To get access to the fault stream of an entity x, a thread simply calls the function `GetFaultStream` with x, which returns the current fault stream. A formal definition is given below. To read the current fault state, one simply takes the first element of the returned list.

$$\frac{y{=}\{\texttt{GetFaultStream}\ x\}}{\sigma} \,\Big\|\, \frac{y{=}fs|s}{\sigma} \quad \text{if} \quad \sigma \models \mathsf{fstream}(x){=}fs|s \qquad (2)$$

Figure 5 shows an example of how an entity's fault stream may evolve over time. The stream is a partially known list, and the underscore "_" denotes an

[1] The fault stream is just like a port's stream.

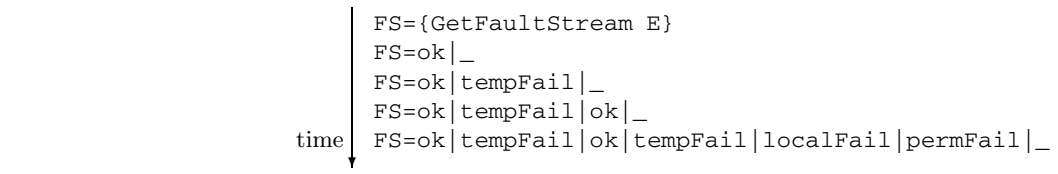

Fig. 5. An example of a fault stream evolving over time

```
thread
   for S in {GetFaultStream E} do
      T = case S                          % pattern matching on S
          of ok        then "entity is fine"
          [] tempFail  then "some problem, don´t know"
          [] localFail then "no longer usable locally"
          [] permFail  then "no longer usable globally"
          end
   in
      {Show T}
   end
end
```

Fig. 6. A thread that prints messages when entity `E`'s fault state changes

anonymous logic variable. Figure 6 shows a thread monitoring an entity `E`, and printing a message for each fault state appearing on the stream. The printed message is chosen by pattern matching. The thread is woken up each time the stream is extended with a new state.

3.4 Enforced Failure

Sometimes the system is unable to diagnose a distribution problem. And often, the actual impact of a distribution problem on a whole application is not reflected in the fault states. It is sometimes simpler to force a part of the application to fail, which causes it to launch a recovery mechanism.

We propose two operations to force an entity to fail, called `KillLocal` and `Kill`. The statement `{KillLocal E}` has a pure local effect. It forces the fault state of `E` to be at least `localFail`. The statement `{Kill E}` attempts to make the entity permanently failed. Execution of `{Kill E}` is asynchronous, i.e., it returns immediately. It initiates a protocol that attempts to make the entity globally failed. To succeed, this may require some of the other sites sharing the entity to be reachable at some point in the future. The local fault state of the entity becomes `permFail` upon confirmation of the failure. The next section gives an example that uses `Kill`.

4 An Example: A Robust Forwarder Tree

This section gives an example showing how to use the fault model defined in the preceding section. We present a simple and flexible group communication abstraction. The abstraction was tested on Mozart using the DSS implementation described in Sect. 5. The abstraction maintains a distributed tree whose nodes forward messages from the root to the leaves. Useful components are inserted as leaves in the tree. Depending on how the forwarding is defined at each internal node, one can broadcast messages or balance messages between components. In the latter case, each node forwards to one of its children only. With a small modification, one can also forward messages from the leaves to the root.

4.1 Architecture

Figure 7 depicts the architecture of the various components of the tree. The tree's nodes, shown as white circles in the figure, appear as Oz ports. They also use other entities (cells, and other ports) that are not visible outside the abstraction. The latter entities are not distributed, because they are never shared outside the site where they were created. This is implied by the fact that threads do not migrate by default.

We assume that the components are given as Oz ports, too. For each component, we create a leaf node, which is inserted in the tree by the root. For the sake of simplicity, the tree is built top-down. Every leaf first becomes a child of the root. When the root has six children, it groups them into two subtrees with three children each. The nodes of the tree can fail at any time, and the failure may be detected by the system or provoked by the program.

On the right of Fig. 7 we have illustrated some rules to follow when nodes fail. The main idea is that an internal node with less than two children makes itself fail. The failure will be propagated down the tree, and eventually forces new leaves to be created and inserted back in the tree. This avoids keeping "skinny" branches (linear chains) in the tree. The root of the tree is the only weak point in the architecture. Other algorithms could be used to make it robust.

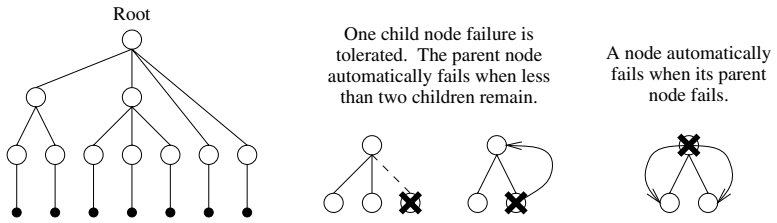

Fig. 7. Architecture of the forwarding tree

```
% create a leaf node
proc {MakeLeaf Root Component}
   Ms Leaf={NewNodeWithParent Ms}
in
   {Send Root insert(Leaf)}
   thread % forwarding messages to Component
      for M in Ms do {Send Component M} end
   end
   thread % handling failures
      case {WaitTwo {WhenFailed Component} {WhenFailed Leaf}}
      of 1 then {Kill Leaf}
      [] 2 then {MakeLeaf Root Component}
      end
   end
end

% return a variable that is bound to true when E fails
fun {WhenFailed E}
   thread {Member permFail {GetFaultStream E}} end
end
```

Fig. 8. Creation of a leaf node in the tree

4.2 Leaf Nodes

Figure 8 shows a procedure that creates a leaf in the tree (identified with its root) for a given component. The code inside the box is the part that handles failures. It can be removed for a non-robust version.

As a first approximation, consider that `NewNodeWithParent` is equivalent to `NewPort`. The returned `Leaf` is effectively a port, and `Ms` is its stream of incoming messages. We will see more in detail how it works below. The code outside the box sends a message to the root node to insert the leaf in the tree, then a thread forwards all incoming messages to the component.

The boxed code handles failures from the leaf node and the component itself. If the component fails, the leaf is forced to fail, too. That leaf will be removed from the tree. If the leaf fails before the component, we simply recreate another leaf. The function `WhenFailed` returns a variable that is bound only when its argument has reached the fault state `permFail`. The function `Member` tests whether a value is an element of a list. The function `WaitTwo` is nondeterministic; it can return 1 if its first argument is bound and 2 if its second argument is bound.

4.3 Nodes Monitoring Their Parent

As we stated before, when a node's parent fails, the node itself must fail. The leaves should follow that rule, too. Figure 9 shows how to implement such a node, in a generic way. As the node must know its parent, we assume that every

```
% create a port node that kills itself when its parent fails
proc {NewNodeWithParent Ms Node}
   Strm
   DependOn = {MakeDependency Node}
   fun {CatchParent M|Ms}
      case M of parent(P) then {DependOn P} {CatchParent Ms}
      else M|{CatchParent Ms}
      end
   end
in
   Ms = thread {CatchParent Strm} end
   Node = {NewPort Strm}
end

% return a procedure that establishes a dependency for E
fun {MakeDependency E}
   CurrentD={NewCell none}
   proc {DependOn D}
      CurrentD := D
      thread
         {Wait {WhenFailed D}}
         if D==@CurrentD then {Kill E} end
      end
   end
in
   DependOn
end
```

Fig. 9. Creation of an Oz port that is the basis of a node in the tree

node in the tree receives a message of the form `parent(P)` with its parent `P`. That message is sent to the node when its parent changes, too. This is the case when a child of the root is put under a new node.

In `NewNodeWithParent`, a port is created for the node. Its stream `Strm` is filtered by the function `CatchParent`, which catches the parenthood message. Note that `CatchParent` is tail recursive, and its output is incremental. When a new parent is found on the stream, the loop calls the procedure `DependOn`, which establishes a link between the parent's failure, and the current node's failure. The procedure `DependOn` uses a hidden cell (`CurrentD`) to keep track of the current dependency. If a dependency `D` fails, and the dependency has not changed, the entity must fail. The procedure `Wait` blocks until its argument gets bound to a value.

4.4 Subtrees

The function `MakeTree` in Fig. 10 takes a list of nodes `Cs` in its argument, and returns a new node with `Cs` as children. That node must monitor its parent,

```
% create an internal node, initially with three children
fun {MakeTree Cs}
   Ms Node={NewNodeWithParent Ms}
   Children={NewCell Cs}
in
   thread % forwarding messages to children
      for M in Ms do {Forward Children M} end
   end
   for C in Cs do
      thread {Send C parent(Node)} end     % send parent message
      thread Cs Cs1 in     % handle child failures
         {Wait {WhenFailed C}}
         Cs = Children := Cs1
         Cs1 = {RemoveFromList C Cs}
         if {Length Cs1}<2 then {Kill Node} end
      end
   end
   Node
end

% forward message M to one or many of Children
proc {Forward Children M}
   for C in @Children do
      thread {Send C M} end
   end
end
```

Fig. 10. Creation of non-root internal nodes of the tree

so it is created by `NewNodeWithParent`. The cell `Children` contains the list of children of that node. The first thread created forwards messages to the children. The definition of `Forward` can be changed to forward to one child only, for instance. The procedure `Forward` is used by the root node as well.

For every child, a message `parent` is sent with the current node. The `Send` operation is performed in a separate thread, to make sure it does not block the main thread in case of a failure. The code that handles the failure is quite easy to read. It waits until the given child fails, then removes it from the children list. If the resulting list has less than two elements, the current node fails. This failure will be handled by `Node`'s parent and remaining children.

4.5 The Root Node

Figure 11 shows the function that makes a root node. That node must be given to the components in order to create the leaves. The root node is similar to the subtree nodes. The first difference is that it has to insert leaves, which may

```
% make the root node of a tree
fun {MakeRoot}
   Ms Root={NewPort Ms}
   Children={NewCell nil}
   proc {Adopt C}
      thread {Send C parent(Node)} end     % send parent message
      thread Cs Cs1 in     % handle child failures
         {Wait {WhenFailed C}}
         Cs = Children := Cs1
         Cs1 = {RemoveFromList C Cs}
      end
   end
in
   thread % handle insertions, and forward messages
      for M in Ms do
         case M of insert(C) then Cs Cs1 in
            Cs = Children := Cs1
            case Cs of [C1 C2 C3 C4 C5] then
               % make two subtrees of the 6 children
               Cs1 = [{MakeTree [C1 C2 C3]}
                      {MakeTree [C4 C5 C]} ]
               for T in Cs1 do {Adopt T} end
            else % simply add another child
               Cs1 = C|Cs
               {Adopt C}
            end
         else {Forward Children M} end
      end
   end
   Root
end
```

Fig. 11. Creation of the root of the tree

force it to create subtrees. The second difference is that it does not fail when its children fail. Failed children are simply removed from its list.

4.6 Discussion of the Example

The first thing to notice is that the code that handles failures has been written so that it interacts as little as possible with the functional part of the abstraction. Keeping the failure handlers in separate threads improves their modularity, and they are quite easy to reason about. A consequence is that it is pretty easy to extend the functional part. If no extra entity is distributed, the failure handlers will not need to be modified.

Another interesting point is that the fault model encourages the programmer to think in terms of events and reactions. A failure is an event on its own, and

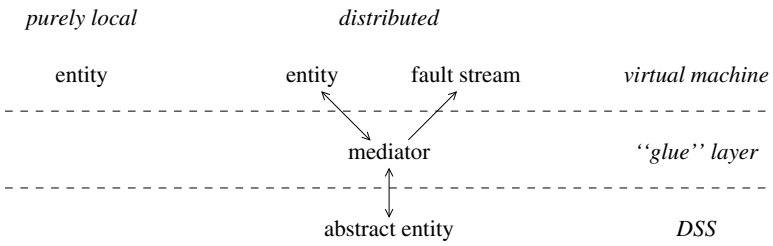

Fig. 12. The implementation's architecture

simple reactions are sometimes enough to make the program recover. The use of message passing also helps to simplify the reasoning.

One issue that has not been mentioned is *where the entities are*. Where should we place the nodes of the tree? The answer is: this is mostly an orthogonal issue. In other words, put them where you want. This works because of network transparency. Only the leaves of the tree should be placed on the site of their component. This is because the thread that monitors the leaf should preferably be on the same site as the component. The root node could keep track of a pool of machines that may host the intermediate leaves, and creates them there. We have not included this part in the code to keep it simple. The Mozart system provides a simple abstraction to create a remote process and execute code on it.

5 Implementation

The Mozart system contains a virtual machine that executes Oz programs and implements the distribution of Oz entities [4]. The distribution part of Mozart is currently being reimplemented with the Distribution SubSystem (DSS) library. The DSS is completely separate from the virtual machine emulator, with a well-defined interface between the two. The DSS provides generic distributed entities [13,14]. There are three types of abstract entities, namely *mutable*, *monotonic*, and *immutable*, and each type comes with a small set of abstract operations. The DSS makes a clear separation between communication protocols and the entity's semantics. The protocols are used internally by the DSS to implement generic operations, a distributed garbage collector, and failure detectors.

The way Mozart uses the DSS is sketched in Fig. 12. A purely local entity is managed exclusively by the virtual machine, while a distributed entity is mapped to an abstract entity in the DSS, which provides the basic support for its distribution. An intermediate object, called the *mediator*, defines the mapping between the virtual machine entity and the DSS entity. The mediator maps entity operations on abstract entity operations, and makes the virtual machine's garbage collector collaborate with the distributed garbage collector. It also reflects the abstract entity's failure state in the virtual machine, by building the fault stream of the entity and resuming threads that suspend because of a failure. The fault stream only has a small overhead in practice, because it is created on demand. Only monitored entities update their fault streams.

6 Lessons from the Past

Our argument against the use of exceptions to handle distribution failures comes from the original fault model used in Oz. The original model overlaps with the new model proposed in Sect. 3. The original model provided much more fault information (most of which was not used in practice) and provided both synchronous and asynchronous handlers. The major difference was the ability to define synchronous failure handlers, i.e., handlers that are called when attempting an operation on a failed entity. The programmer could either ask for an exception or provide a handler procedure that replaces the operation. The failure handler was defined for a given entity and with certain conditions of activation.

Instead of the synchronous handlers, programmers favored an asynchronous handler, called a *watcher*. A watcher is a user procedure that is called in a new thread when a failure condition is fulfilled. The fault stream we propose in this paper simply factors out how the system informs the user program. It also avoids race conditions related to the watcher registry system, which could make one miss a fault state transition. And finally, a watcher could not be triggered by a transition to state `ok`.

The original model had one further deficiency. There was no way to force an entity to be considered failed locally. As a result, there was a lack of control in case of erratic entity behavior (e.g., many transitions between `ok` and `tempFail`).

The original model is criticized in [5], which proposes an alternative model. That paper proposes something similar to our fault stream and an operation to make an entity fail locally. In order to handle faults, it proposes to explicitly break the transparent distribution of a failed entity. The local representative of the failed entity is disconnected from its peers and is put in a fault state equivalent to `localFail`. Another operation replaces that entity by a fresh new entity. This model has the advantage to avoid blocking threads on failed entities, because you can replace a failed entity by a healthy one. But this replacement introduces inconsistencies in the application's shared memory. We were not able to give a satisfactory semantics that takes into account these inconsistencies.

7 Related Work

Most mainstream programming languages use exceptions to reflect failures due to distribution faults. Those systems often propose less ambitious models for distributed programs. They usually do not favor concurrency, the distribution is often explicit, and failure handling is often mixed with the functionality of the program. A typical representative of those systems is Java's Remote Method Invocation (RMI) system. The exceptions thrown because of network failures are visible in the methods' signatures. Moving from a centralized to a distributed application requires to change API's explicitly. Making robust distributed abstractions is not impossible, but it comes at the price of a huge complexity increase in the program.

An interesting question is: how to implement the forwarder tree with the RMI approach? The problem is that no message is sent "upwards" the tree, hence a node never calls its parent. In order to detect the failure of its parent, a node would need to make regular dummy calls to it. Another possibility is to make the communication channel explicit, and catch problems at the receiving side. But this breaks the abstraction provided by RMI. Moreover, extra messages are required to simulate node failures, if those failures are not caused by site failures.

The Erlang programming language and system was designed at the Ericsson Computer Science Laboratory for building high availability telecommunication systems [15,16]. An Erlang program consists of a (possibly large) number of processes. An Erlang process is a lightweight thread with its own memory space. Processes are programmed with a strict functional language, and they communicate by asynchronous message passing.

Erlang provides asynchronous fault detection of permanent failures between processes. Two processes can be linked together. When one of them fails, the other one receives a message from the runtime system, provided it is declared as a supervisor. Erlang chooses to model all failures as permanent failures, in accordance with its philosophy of "Let it fail". That is, keeping the fault model simple allows the recovery algorithm to be simple as well. This simplicity is very important for correctness. We extend Erlang's model with temporary failures and with a fault stream. Furthermore, our model is designed for a richer language than Erlang, which only has stationary objects (in our terminology).

8 Conclusion

This paper proposes a simple fault model for the distributed execution of the Oz language. This distributed execution is network transparent, i.e., the semantics of a language entity does not depend on whether it is distributed or not. *Synchronous* failure handlers, like exceptions, break this transparency property. Moreover, they are no longer practical if the language is highly concurrent. We give evidence that *asynchronous* failure handlers are more adequate. They can be defined so that they do not break the network transparency of the language. In our design, each language entity produces a stream giving its fault state transitions. Monitoring an entity is done by reading the stream. One can also force a failure either locally or globally, which allows to implement simple abstractions for handling partial failure.

Acknowledgments

We thank our colleagues Kevin Glynn and Boris Mejías for fruitful discussions about the model and careful proofreading of the paper. This work is supported by the CoreGRID Network of Excellence (contract number 004265) and the EVERGROW Integrated Project (contract number 001935), both funded by the European Commission in the 6^{th} Framework program.

References

1. Haridi, S., Van Roy, P., Brand, P., Schulte, C.: Programming languages for distributed applications. New Generation Computing **16**(3) (1998) 223–261
2. Waldo, J., Wyant, G., Wollrath, A., Kendall, S.: A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Mountain View, CA (1994)
3. Van Roy, P.: On the separation of concerns in distributed programming: Application to distribution structure and fault tolerance in Mozart. In: International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA 99), Sendai, Japan (1999) 149–166
4. Mozart Consortium (DFKI, SICS, UCL, UdS): The Mozart programming system (Oz 3) (1999) `http://www.mozart-oz.org`.
5. Grolaux, D., Glynn, K., Van Roy, P.: A fault tolerant abstraction for transparent distributed programming. In: Second International Mozart/Oz Conference (MOZ 2004), Charleroi, Belgium (2004) Springer-Verlag LNCS volume 3389.
6. Al-Metwally, M.: Design and Implementation of a Fault-Tolerant Transactional Object Store. PhD thesis, Al-Azhar University, Cairo, Egypt (2003)
7. Saraswat, V.A.: Concurrent Constraint Programming. MIT Press (1993)
8. Van Roy, P., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. MIT Press, Cambridge, MA (2004)
9. Smolka, G.: The Oz programming model. In: Computer Science Today. Lecture Notes in Computer Science, vol. 1000. Springer-Verlag, Berlin (1995) 324–343
10. Haridi, S., Van Roy, P., Brand, P., Mehl, M., Scheidhauer, R., Smolka, G.: Efficient logic variables for distributed computing. ACM Transactions on Programming Languages and Systems **21**(3) (1999) 569–626
11. Van Roy, P., Haridi, S., Brand, P., Smolka, G., Mehl, M., Scheidhauer, R.: Mobile objects in Distributed Oz. ACM Transactions on Programming Languages and Systems **19**(5) (1997) 804–851
12. Van Roy, P., Haridi, S., Brand, P.: Distributed programming in Mozart – A tutorial introduction. Technical report (1999) In Mozart documentation, available at `http://www.mozart-oz.org`.
13. Klintskog, E., El Banna, Z., Brand, P.: A generic middleware for intra-language transparent distribution. Technical Report T2003:01, Swedish Institute of Computer Science (2003)
14. Klintskog, E., El Banna, Z., Brand, P., Haridi, S.: The DSS, a middleware library for efficient and transparent distribution of language entities. In: Proceedings of HICSS'37. (2004)
15. Armstrong, J., Williams, M., Wikström, C., Virding, R.: Concurrent Programming in Erlang. Prentice-Hall, Englewood Cliffs, N.J. (1996)
16. Armstrong, J.: Making Reliable Distributed Systems in the Presence of Software Errors. PhD thesis, Royal Institute of Technology (KTH), Kista, Sweden (2003)