Failure Handling in a Network-Transparent Distributed Programming Language

Raphaël Collet, Peter Van Roy, Boris Mejías, Yves Jaradin
Université catholique de Louvain
Belgium
Abstract

- We present a model for reflecting failures at the level of entities in a distributed programming language.

- The model favors *asynchronous* failure handling, where the failure is notified independently of the entity's usage (sort of failure *listener*).

- We think that this model is more expressive than exception-based failure reflection.
Plan

- Oz and Mozart
- Transparent distribution
- Tuning the distribution with annotations
- Reflecting and handling failures
- Implementation
The programming language Oz

- High-level programming language, with well-integrated declarative, stateful, and concurrent paradigms.
- Implemented by the platform Mozart.
- Provides support for transparent distribution.
  - in case of no failure, the semantics does not depend on how the program is distributed
- Now provides enhanced support for distribution awareness at the language level.
The platform Mozart

- Implements the language Oz.
  - First released in 1998
  - Provides support for distribution and fault handling

- Will be used for one of the project's implementations.

- The distribution support is reimplemented on top of the library DSS (distributed subsystem, see later).
Transparent distribution

- Language entities (logic variable, object, record) can be shared between sites (system processes).
- The system hides the distribution to the programmer. Entity operations on distributed entities are implemented by protocols.
- In case of no failure, the semantics of the program is the same as if the whole program was running on a single site.
Example : getting connected

- C={NewCell 42}
  T={Connection.offer C}
  - T is a string that can be transmitted to the other site by any means.

- D={Connection.take T}
  D:=43
  - C and D refer to the same shared entity. Both sites can use it. Other references are obtained by transitivity.
Example: cell with mobile state

C

C := 43

Migrate contents

C: 42

Update contents

C: 42

C: 43
Example: dataflow variables

Y = X + 3

“Home” of X

X = 42

Blocks on X

bind

Wakes up thread

Y = 45

X = 42

X = 42

X = 42
Tuning by annotation

• Distribution parameters can be defined for an entity before it is distributed. One can choose the most appropriate network behavior without changing the entity's semantics.

• Separation of concerns!

• Examples:
  
  {Annotate A stationary}
  {Annotate B migratory}
  {Annotate C gc(lease 30000)}
Fault model

- For every entity, each site maintains a fault state that reflects the entity's ability to function.
  - ok, tempFail, localFail, permFail
- An operation on a failed entity simply blocks until the entity (possibly never) works again.
- You never get unexpected behavior from a distributed program that experiences failures!
  - Exception are difficult to deal with in a highly concurrent programming language.
Fault model

• An entity's fault state can be monitored in the language by reading its fault stream, that reifies the history of fault states of the entity.

\[
\text{fstream}_i(x) = \text{fs} \land \text{fs} = \text{slfs}' \\
\text{fstream}_i(x) = \text{fs}' \land \text{fs} = \text{slfs}' \\
\quad \land \text{fs}' = \text{slfs}''
\]

The fault state of x on site i goes from s to s'

• \{\text{GetFaultStream E}\} returns the current fault stream of entity E (given by fstream()).
Printing an entity's states

- States are observed in a sequential way, and the observer is woken up by dataflow.

```plaintext
for S in \{GetFaultStream E\} do
    case S
        of ok then {Show 'fine'}
        [] tempFail then {Show 'check network!'}
        [] localFail then {Show 'unusable here'}
        [] permFail then {Show 'unusable'}
    end
end
```
Switching between servers upon failure

fun {MakeProxy Servers}
  Ms
  Unsent={NewCell Ms}
in
  thread
    for S in Servers do
      thread
        for M in @Unsent do
          {Send S M} Unsent:=@Unsent.2
        end
      end
    end
{Wait {Member permFail {GetFaultStream S}}}
end
end
{NewPort Ms}
end
Making an entity fail

- The programmer can manually make an entity fail. This is convenient for triggering a recovery mechanism.
- \{KillLocal \ E\} makes the entity \( E \) fail on the current site only.
- \{Kill \ E\} attempts to make the entity \( E \) fail on a global scale. This operation is asynchronous.
Propagating failure on a set of entities

fun {WhenFailed E}
    thread {Member permFail {GetFaultStream E}} end
end

proc {SyncFail Es}
    thread
        Trigger
    in
        for E in Es do Trigger={WhenFailed E} end
        {Wait Trigger}
        for E in Es do {Kill E} end
    end
end
Asynchronous failure handling

- This model improves the *modularity* of failure handling, because one can easily make it orthogonal to the “functional” code.
  - Handling rules “when failure do recover” are coded in a natural way.
  - Improves the separation of concerns.
- We believe this model is good for Selfman's purposes.
Implementation: Mozart/DSS

- The distribution support of the upcoming Mozart release is built on top of the DSS.
- DSS is a C++ library that provides generic distributed entities to support any programming system. Main design and implementation by Erik Klintskog at SICS.
  - Protocols to manage the entities' state
  - Protocols to garbage collect entities
Abstract entities

- The DSS defines three categories for abstract entities: immutable, monotonic and mutable. Each category defines a small set of abstract operations.

- Each category comes with a set of protocols that implement its abstract operations. The protocols are designed to give the best behavior for each kind of entity.
  - No “one size fits all”.

Coordination structure

coordinator

proxy

proxy

proxy

proxy
State protocols

- immutable: immediate or lazy copying

- monotonic:
  - single assignment protocol (with by-need sync),
  - stream communication

- mutable:
  - stationary state,
  - migratory state (2 protocols),
  - replicated state (2 invalidation protocols)
Single assignment protocol
Single assignment protocol
Distributed garbage collection

- Simple reference counting: the coordinator counts its proxies.
- Weighted reference counting: a proxy has an amount of credits. When a proxy send its reference to another site, it put some of its credits in the message. The coordinators simply counts how many credits are away.
- Time lease algorithm: proxies must regularly notify the coordinator.
Reflecting failures

- Remote site failures are notified to coordination and protocol objects. The programmer can help diagnosing site failures.

- Coordinators and protocols reflect the site failure in the entity's fault state, and notify the above layers when the fault state changes.

- Site dependencies vary from protocol to protocol. Each one has its own way to diagnose an entity failure.
Route reflector

- The user of the DSS must provide transport channels to the remote sites. Currently TCP connections are used.
- The idea behind the “route reflector” is to let the programmer provide its own channels. One can provide connections on top of an overlay network programmed in Oz.
  - TCP connections to neighbors
  - “Overlay channel” to other nodes.
Route reflector

{Send P2 s(1 3 M)}

{Tcp.write M'}

{Send P3 s(1 3 M)}

{S1.receive M}

{Tcp.write M''}

M

S3

S2

S1

DSS

S3

S1

DSS

S2

DSS

S3

DSS
Route refector

- The overlay network can provide more complex ways to diagnose a site failure (consensus).
- The overlay network is mostly invisible to the application. One can therefore make some self-* properties partly transparent too!
Conclusion

• We define a language model to reflect failures and to handle them. The model chooses asynchronous failure handling as the preferred way to make robust applications.

• Mozart/DSS is an improved implementation of the distribution of Oz.
  – Currently available in Mozart's CVS repository.
  – Release in a few months.