Modular Fault Tolerance in a Network-Transparent Programming Language

LADA Workshop ("Languages for Distributed Algorithms")
Philadelphia, PA

Jan. 23-24, 2012

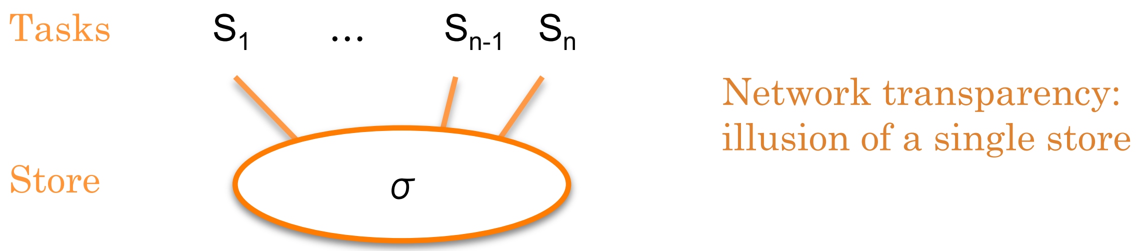
Peter Van Roy, Raphaël Collet, Sébastien Doeraene, and Géry Debongnie
ICTEAM Institute
Université catholique de Louvain
B-1348 Louvain-la-Neuve, Belgium

OVERVIEW

- Our goal is to build a programming language and system for distributed applications that removes all irrelevant complexity
  - Combine network transparency with network awareness in a language that supports the right concepts
  - Research implementation based on Oz language and Mozart Programming System
  - Evaluating the approach and the language
- Failure and modularity
  - The failure model and how fault streams expose it to the program
  - Blocking failure handling versus non-blocking failure handling
  - Why non-blocking failure handling is right for network transparency
- Theory and practice
  - Formal semantics of Distributed Oz
  - Some common fault stream programming patterns
- Conclusions and references

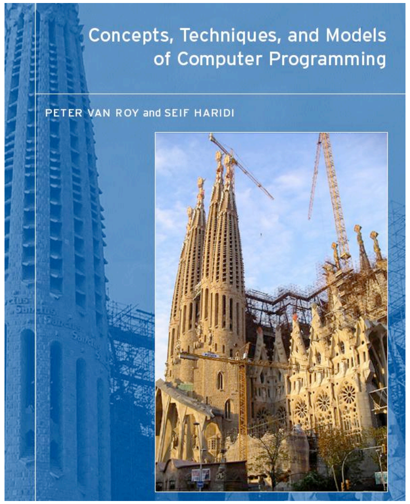
Network Transparency and Network Awareness

- We would like to remove irrelevant programming complexity
  - "The removal of much of the accidental complexity of programming means that the intrinsic complexity of the application is what's left" – Ross Anderson, in *Security Engineering* (2001)
  - Remove code for marshalling/unmarshalling, connecting/disconnecting, failure detection, data caching, globally unique identities, memory management



- Network transparency: execution obeys the same language semantics independent of physical distribution
  - A program executing on multiple nodes gives same result as on one node, if network delays are ignored
- Network awareness: sufficient system properties can be observed and controlled by program so that execution is efficient
  - Physical distribution, network behavior, partial failure

CONTEXT OF THE RESEARCH



- Oz multiparadigm language (Gert Smolka *et al*)
  - Contains many concepts, factored for simplicity
    - See textbook *Concepts, Techniques, and Models of Computer Programming*, MIT Press 2004
  - For distribution we especially appreciate:
    - Fine-grained concurrency & non-blocking channels
    - Distinction between stateless, single assignment (monotonic), stateful
    - Higher-order declarative dataflow subset with concurrency and laziness
- Mozart Programming System (www.mozart-oz.org)
  - Open-source, many developers, first public release in 1991
  - Network-transparent distribution in 1999
  - Improved distribution architecture in 2005 (Erik Klintskog)
  - Modular fault tolerance in 2007 (Raphaël Collet)
  - Distributed applications and scalability ongoing since 2008

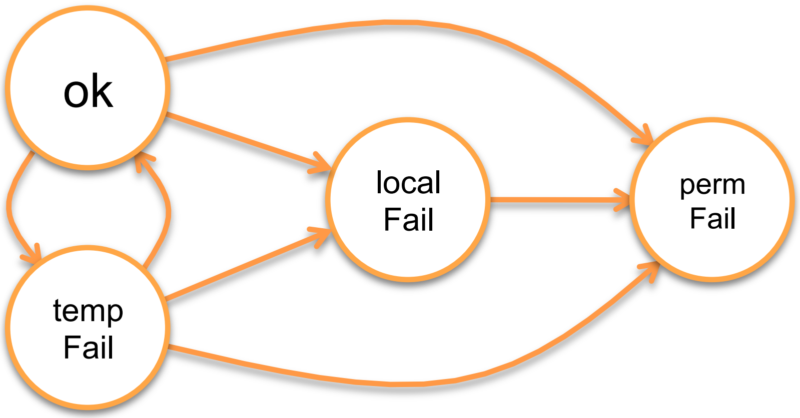
THE GOOD, THE BAD, AND THE UGLY

- Network transparency is implementable and practical. We have built large nontrivial applications that are of intrinsic interest, for example:
  - TransDraw (Donatien Grolaux): multi-user graphic editor that uses distributed transactions combined with human-computer interface design to overcome network delays (instantaneous reactivity combined with global coherence)
  - Beernet (Boriss Mejías): scalable transactional store based on self-organizing peer-to-peer network, replication, and Paxos uniform consensus for atomic commit
- Network awareness is more difficult
  - Partial failure cannot be hidden: focus of this talk
  - Ability to have efficient network behavior for natural programs
  - Ability to define fault-tolerance abstractions within the system
- Caveats for implementation and usability
  - Implementing this distribution model is a lot of work: several Ph.D. theses (Per Brand, Erik Klintskog, Raphaël Collet)
  - Usability depends on fault model: so far, we have a model that fits most Internet failures
  - The current system does not support security or long-term resource management defined inside the system (i.e., formal software rejuvenation)

THE MODEL IN A NUTSHELL

- Each language entity (mutable variable, channel, dataflow variable, thread, closure, record, number, name) can be distributed
  - Operations on the entity are implemented using distributed algorithms to keep the same semantics as a centralized system
- Each node maintains a local fault state for each distributed entity: ok, tempFail, localFail, and permFail
  - Failure model designed for Internet with TCP/IP: crash-stop processes and FIFO message delivery between processes with arbitrary message delay or loss
- The fault state is reified in the language as a fault stream, i.e., a monotonically growing list of fault states, a new state added for each transition
  - This fault state can be monitored by a separate thread
- Any operation on a failed entity blocks until the fault state is ok, or forever if it is permFail or localFail
  - Failure causes no behavior that would be incorrect for no failure

Local Fault State of an Entity



- The fault stream combines information from three failure detectors
  - tempFail detector: suspect/resume events, eventually perfect and adaptive to round-trip time variations
  - permFail detector: accurate but incomplete, for example process failure inside a host that does not fail
  - localFail detector: perfect, but handles local failures only
- After experience with more complex detectors, we find that giving the simplest useful information is best

Fault-Tolerant Compute Server: Blocking Version

```
proc {RemoteCompute Comp ?Res}
  Node = {GetNodeFromPool}
  ResFromNode
in
  try
    ResFromNode = {Send Node comp(Comp $)} /* send computation to node */
    {Wait ResFromNode} /* wait for result (dataflow synchronization) */
    Res=ResFromNode
  catch remoteException(Why) then /* retry if failure */
    {RemoteCompute Comp Res}
  end
end
```

Red code is for failure handling

An exception is raised when there is an attempt to use the result

Note: This example and the others in this talk are written in Oz. The syntax is designed to support the language's multiparadigm design. Three properties should suffice to understand the examples:

- *variable identifiers start with capital letters,*
- *procedure/function calls are enclosed in braces {…}, and*
- *'$' marks the return argument when a statement is used as an expression.*

Fault-Tolerant Compute Server: Non-Blocking Version

```
proc {RemoteCompute Comp ?Res}
  Node = {GetNodeFromPool}
  ResFromNode
in
  ResFromNode = {Send Node comp(Comp $)} /* send computation to node */
  {MonitorResult ResFromNode proc {$} {RemoteCompute Comp Result end}
  {Wait ResFromNode} /* wait for result */
  Res=ResFromNode
end
```

Red code is for failure handling

No exception is raised; instead the attempted use waits indefinitely and the fault stream is extended

```
proc {MonitorResult ?ResFromNode OnFail}
  proc {Loop Xs} /* loop over fault stream */
    case Xs of ok|Xr then {Loop Xr}
    [] tempFail|Xr then {Break ResFromNode} {Loop Xr} /* restart a slow computation */
    [] F|Xr andthen (F==localFail orelse F==permFail) then {OnFail} /* retry if failure */
    [] nil then skip
  end
in
  thread {Loop {GetFaultStream ResFromNode}} end
end
```

Why Non-Blocking Failure Handling is Better

- First problem with the blocking solution: it cannot handle tempFail efficiently since any failure will abort the remote computation and raise an exception
- But this is small fry compared to the real problem. In an network-transparent system, the invoking node of any remote operation is not aware that its result is coming from another node.
  - Because of dataflow, the result behaves like a promise: the invoking node can pass the result through the program, bind it to other dataflow variables, put it in data structures, and can potentially use it **at any arbitrary point** in the program (it will wait at that point until the remote computation returns its result)
  - If a failure occurs during the remote computation, then blocking failure handling will raise an exception at that point:
    - It couples the **entire application** to the distributed execution!
  - With blocking failure handling, exception handlers are needed everywhere to handle potential distribution failures!
  - With non-blocking failure handling, this problem does not occur. Program execution will wait without raising an exception. The failure can be handled in a separate thread that monitors the fault stream.

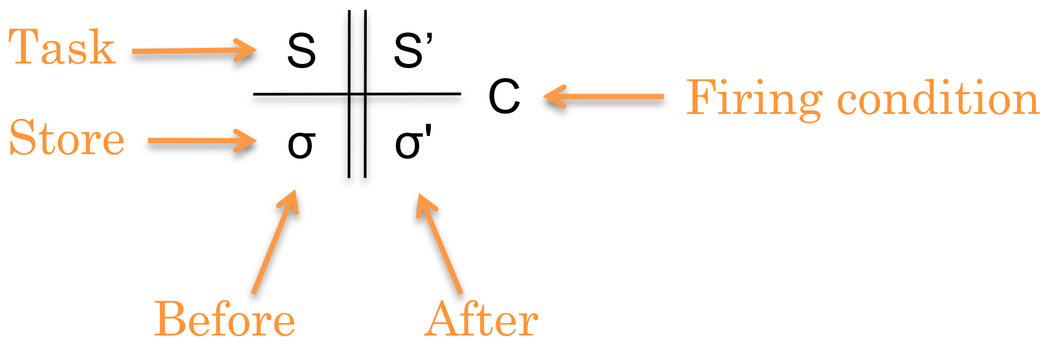
Some Related Work

- Non-blocking failure handling in Oz can be seen as a generalization of Erlang's failure handling:
  - "Let it fail" for both: use simple failure states
  - Ordered versus unordered sequence of fault states
  - Handling of temporary failures
  - Granularity is a language entity instead of process
  - The Erlang model can be implemented in just one page of code
- Network-transparent distribution was pioneered by Emerald
  - Emerald's abilities inspired the design of Distributed Oz
  - Emerald has powerful mobility primitives for threads and objects
  - Distributed Oz has a useful declarative concurrent subset that includes declarative dataflow and lazy evaluation as special cases
  - Distributed Oz emphasizes asynchronous programming (using non-blocking channels and fine-grain concurrency)

Formal Semantics of Distributed Oz

Operational Semantics in a Concurrent Constraint Model

- Small-step operational semantics



Note: other constraints possible (e.g., Bloom's sets!)

- Concurrent constraint model
  - Store σ is a monotonic conjunction of constraints
  - $\sigma = \{ x, y, z=\mathrm{rec}(u), w=100, u=\mathrm{tup}(x\ y\ z), \ldots \}$
  - Variables x, y, z and single-assignment bindings
  - Firing condition C is decidable logical entailment: $\sigma \vDash c$

Example: If Statement

- Control flow is determined by a logical condition
  - If nothing is known about x, then execution waits
  - This implements declarative dataflow

$$\textsc{IfTrue}\quad \frac{\textbf{if } x \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}}{\sigma} \;\Bigg\|\; \frac{S_1}{\sigma} \quad \boxed{\sigma \models x\textbf{=true}}$$

Store knows
x is true

$$\textsc{IfFalse}\quad \frac{\textbf{if } x \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}}{\sigma} \;\Bigg\|\; \frac{S_2}{\sigma} \quad \sigma \models x\textbf{=false}$$

- Concurrent constraint model is expressive and concise
  - Complete semantics of Oz multiparadigm language
  - Used in many languages, e.g., E, Joule, AKL, Concurrent Prolog and its successors, constraint programming

MUTABLE STATE

- Mutable state (cell) is defined as a pair $x{:}y$ of a name x (bound to a constant ξ) and a content y

$$\textsc{Exchange}\quad \frac{\{\text{Exchange } x\ y_{old}\ y_{new}\}}{\sigma \wedge x{=}\xi \wedge x{:}y} \;\Bigg\|\; \frac{y_{old}{=}y}{\sigma \wedge x{=}\xi \wedge x{:}y_{new}}$$

Cell name   Pair of name and content

- Exchange operation atomically does a read and a write
  - Read operation: bind y_{old} and y
  - Write operation: replace old content by y_{new}
- Left side store notation $\sigma \wedge x{=}\xi$ is equivalent to condition $\sigma \models x{=}\xi$
- Objects are compound entities built with cells, closures, and records

DISTRIBUTED MUTABLE STATE

Exchange executes on node i

Result is a bind operation on node i

$$\textsc{Mobile Exchange}\quad \frac{\{\text{Exchange } x\ y_{old}\ y_{new}\}_i \quad \Big\| \quad (y_{old}=y)_i}{\sigma \wedge x=\xi \wedge (x{:}y)_j \quad \Big\| \quad \sigma \wedge x=\xi \wedge (x{:}y_{new})_i}$$

Name is known everywhere
(no node annotation)

New cell content is on node i

Old cell content is on node j

- We refine the semantics of cells and exchange to specify the nodes on which the tasks and store contents are located
  - Each operation and store item is annotated with a node
  - This rule defines a mobile cell implemented with a mobile state protocol; other rules correspond to other distributed behaviors (stationary/replicated cell, weaker consistency)

Fault Stream Operations

- Each language entity has one fault stream per node
- The fault stream is extended with each transition of the entity's fault state (interface to the failure detector):

$$\text{FSExtend} \quad \frac{}{\sigma \wedge (fstream_i(x)=f\,|s)_i} \;\Big\|\; \frac{}{\sigma \wedge (fstream_i(x)=s)_i \wedge (s=f'\,|s')_i} \quad f \rightarrow f'$$

- The program can access the fault stream of a language entity at node i with the operation GetFaultStream:

$$\text{FSAccess} \quad \frac{(y=\{\text{GetFaultStream } x\})_i}{\sigma} \;\Big\|\; \frac{(y=s)_i}{\sigma} \quad \sigma \models (fstream_i(x)=s)_i$$

- Possible values of the fault state f depend on the failure model; for Mozart: $f \in$ {ok,tempFail,localFail,permFail}

The Complete Abstraction

- Three primitive operations are provided for programs to implement failure handling

1) Access an entity's fault stream at the current node:
   - S={GetFaultStream X}
2) Cause an entity to fail globally:
   - {Kill X}
   - If this succeeds, permFail will appear on the fault stream
   - If the entity is temporarily inaccessible (tempFail), this will wait until the entity is accessible and then cause it to fail
3) Cause an entity to fail on the current node:
   - {Break X}
   - This always succeeds; localFail will appear on the fault stream
   - Any attempt to use the entity on the current node will block forever; the entity is still operational on other nodes

Some Common Fault Stream Programming Patterns

Monitoring the Fault Stream

```
FS = {GetFaultStream E}
thread {Monitor FS} end
proc {Monitor S}
  case S of F|S2 then
     case F of ok then skip /* do nothing */
     [] tempFail then /* things are slowing down */
        <doSomething>
     [] localFail then /* local use no longer possible */
        <doSomething>
     [] permFail then /* it's dead everywhere, Jim */
        <doSomething>
     end
   end
   {Monitor S2}
end
```

If One Dies, Kill Them All (An Erlang Abstraction)

```
proc {SyncFail Es} /* argument is list of entities */
  Trig in
  for E in Es do /* set up a failure monitor for each entity */
    thread
      if {List.member permFail {GetFaultStream E}} then
      Trig=unit end
    end
  end
  thread /* if one is dead, kill them all (Erlang style) */
    {Wait Trig}
    for E in Es do {Kill E} end
  end
end
```

Post-mortem Finalization

- The fault stream is closed (terminated with nil) when the entity is garbage collected (known to be no longer accessible). This can be used to implement post-mortem finalization.

```
proc {Finalize E P} /* execute P after E is GC'ed */
  thread
    for X in {GetFaultStream E} do skip end
    {P} /* clean up after E */
  end
end
```

Conclusions and References

Conclusions

- Network-transparent distribution is a promising path for distributed application development
  - It increases the abstraction level of the programming language
- It still has many challenges:
  - Supporting more general fault models than just "Internet failures"
  - Efficient native code implementation (Mozart is emulated byte code)
  - Long-lived scalable applications should have all resource management "inside the language"
- Non-blocking failure handling is the right approach
  - It is natural for asynchronous programming with declarative dataflow
  - In a network-transparent system, it avoids the need to handle potential failure exceptions everywhere in the program
- Generalizations of Distributed Oz
  - Executable specifications for distributed algorithms
  - CALM and CRON generalize the declarative concurrent execution model of Distributed Oz

References

- Seif Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer, and Gert Smolka. *Efficient Logic Variables for Distributed Computing*. ACM TOPLAS, May 1999, pp. 569-626.
- Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile Objects in Distributed Oz. ACM TOPLAS, Sep. 1997, pp. 804-851.
- Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.
- Per Brand. The Design Philosophy of Distributed Programming Systems: the Mozart Experience. Ph.D. Dissertation, KTH, June 2005.
- Erik Klintskog. Generic Distribution Support for Programming Systems. Ph.D. Dissertation, KTH, April 2005.
- Raphaël Collet. The Limits of Network Transparency in a Distributed Programming Language. Ph.D. Dissertation, UCL, Dec. 2007.
- Donatien Grolaux. Transparent Migration and Adaptation in a Graphical User Interface Toolkit. Ph.D. Dissertation, UCL, Sep. 2007.
- Boris Mejías. Beernet: A Relaxed Approach to the Design of Scalable Systems with Self-Managing Behaviour and Transactional Robust Storage. Ph.D. Dissertation, UCL, Oct. 2010.