Robust distributed programming in the Mozart platform: the importance of language design and distributed algorithms

La programmation répartie robuste dans la plate-forme Mozart : le rôle du langage et de l’algorithmique répartie

Langages et Modèles à Objets (LMO’2002)

Peter Van Roy
Département d’Ingénierie Informatique
Université catholique de Louvain
25 janvier 2002
Overview

• Designing a platform for robust distributed programming requires thinking about both language design and distributed algorithms
  – Distribution and state do not mix well (global coherence); the language should help (weaker forms of state, different levels of coherence)
• We present one example design, the Mozart Programming System
  – Mozart implements efficient network-transparent distribution, refining language semantics with distribution
• We give an overview of the language design and of the distributed algorithms used in the implementation
  – It is the combination of the two that makes distributed programming simple in Mozart
• Conclusions and ongoing work
  – Projects starting in high availability, security, peer-to-peer
Mozart at a Glance

• **Oz Language**
  – A concurrent, compositional, object-oriented language that is state-aware and has dataflow synchronization
  – Simple formal semantics and efficient implementation

• **Strengths**
  – **Concurrency**: ultralightweight threads, dataflow
  – **Distribution**: network transparent, network aware, open
  – **Inferencing**: constraint, logic, and symbolic programming
  – **Flexibility**: dynamic, no limits, first-class compiler

• **Mozart System**
  – Under development since 1991 (distribution since 1995), 10-20 people for 10 years
  – Mozart Consortium: Universität des Saarlandes (Germany), Swedish Institute of Computer Science (Sweden), Université catholique de Louvain (Belgium)
  – Releases for many Unix/Windows flavors; free software (X11-style open source license); maintenance; user group; technical support ([http://www.mozart-oz.org](http://www.mozart-oz.org))

• **Research and applications**
  – Research in distribution, fault tolerance, resource managements, constraint programming, language design and implementation
  – Applications in multi-agent systems, “symbol crunching”, collaborative work, discrete optimization (e.g., tournament planning)
Basic principles

- *Refine* language semantics with a distributed semantics
  - Separates *functionality* from *distribution structure* (network behavior, resource localization)

- Three properties are crucial:
  - **Transparency**
    - Language semantics *identical* independent of distributed setting
    - Controversial, but let’s see how far we can push it, *if* we can also think about language issues
  
  - **Awareness**
    - Well-defined distribution behavior for each language entity: simple and predictable
  
  - **Control**
    - Can give different distribution behaviors for a given language entity
    - Example: objects are stationary, cached (mobile), asynchronous, or invalidation-based, with same language semantics
Mozart today
Language design

- Language has a **layered structure** with three layers:
  - **Strict functional core** (stateless): exploit the power of lexically-scoped closures (“call backs done right”)
  - **Single-assignment extension** (dataflow variables + concurrency + laziness): provides the power of concurrency in a simple way (“declarative concurrency”)
  - **State extension** (mutable pointers / communication channels): provides the advantages of state for modularity (object-oriented programming, many-to-one communication and active objects, transactions)
- **Dataflow extension is well-integrated with state**: to a first approximation, it can be ignored by the programmer (it is not observable whether a thread temporarily blocks while waiting for a variable’s value to arrive).
- **Layered structure is well-adapted for distributed programming**
  - This was a serendipitous discovery that led to the work on distributing Oz
- **Layered structure is not new**: see, e.g., Smalltalk (blocks), Erlang (active objects with functional core), pH (Haskell + I-structures + M-structures), even Java (support for immutable objects)

Adding distribution

- Each language entity is implemented with one or more distributed algorithms. The choice of distributed algorithm allows tuning of network performance.
- Simple programmer interface: there is just one basic operation, passing a language reference from one process (called “site”) to another. This conceptually causes the processes to form one large store.
- How do we pass a language reference? We provide an ASCII representation of language references, which allows passing references through any medium that accepts ASCII (Web, email, files, phone conversations, …)
- How do we do fault tolerance? We will see later in the talk…
Example: sharing an object (1)

```plaintext
class Coder
    attr seed
    meth init(S) seed<-S end
    meth get(X)
        X=@seed
        seed<-(@seed*23+49)mod 1001
    end
end

C={New Coder init(100)}

T={Connection.offer C}
```

- Define a simple random number class, Coder
- Create one instance, C
- Create a ticket for the instance, T
- The ticket is an ASCII representation of the object reference
Example: sharing an object (2)

C2={Connection.take T}

\[\text{local } X \text{ in}\]
\[\{\text{C2 get}(X)\}\]
\[\% \text{ Do calculation with } X\]
\[\ldots\]
\[\text{end}\]

- Let us use the object C on a second site
- The second site gets the value of T (through the Web or a file, etc.)
- We convert T back to an object reference, C2
- C2 and C are references to the same object

*What distributed algorithm is used to implement the object?*
Example: sharing an object (3)

- C and C2 are the same object: there is a distributed algorithm guaranteeing coherence.
- Many distributed algorithms are possible, as long as the language semantics are respected.
- By default, Mozart uses a cached object: the object state synchronously moves to the invoking site. This makes the semantics easy, since all object execution is local (e.g., exceptions raised in local threads). A cached object is a kind of mobile object.
- Other possibilities are a stationary object (behaves like a server), an invalidation-based object, etc.
Example: sharing an object (4)

- **Cached objects:**
  - The object state is mobile; to be precise, the right to *update the object state* is mobile, moving synchronously to the invoking site.
  - The object class is stateless (a record with method definitions); it therefore has its own distributed algorithm: it is copied once to each process referencing the object.
  - We will see the protocol of cached objects later in the talk, together with its fault behavior. The mobility of a cached object is lightweight (maximum of three messages for each move).
Language entities and their distribution protocols

- **Stateless** (records, closures, classes, software components)
  - Coherence assured by **copying** (eager immediate, eager, lazy)
- **Single-assignment** (dataflow variables)
  - Allows to decouple communications from object programming
  - To first approximation: can be **completely ignored**
  - Uses distributed binding algorithm (in between stateless and stateful!)
- **Stateful** (objects, communication channels, component instances)
  - Synchronous: stationary, cached (mobile), invalidation protocols
  - Asynchronous FIFO: channels, asynchronous object calls
The path to true distributed object-oriented programming

• Simplest case
  – Stationary object: synchronous, similar to Java RMI but fully transparent, i.e., automatic conversion local↔distributed

• Tune distribution behavior without changing language semantics
  – Use different distributed algorithms depending on usage patterns, but language semantics unchanged
  – Cached (« mobile ») object: synchronous, moved to requesting site before each operation → for shared objects in collaborative applications
  – Invalidation-based object: synchronous, requires invalidation phase → for shared objects that are mostly read

• Tune distribution behavior with possible changes to language semantics
  – Sometimes changes are unavoidable, e.g., to overcome large network latencies or to do replication-based fault tolerance (more than just fault detection)
  – Asynchronous stationary object: send messages to it without waiting for reply; synchronize on reply or remote exception
  – Transactional object: set of objects in a « transactional store », allows local changes without waiting for network (optimistic or pessimistic strategies)
Stationary object

- Each object invocation sends a message to the object and waits for a reply (2 network hops)
- Creation syntax in Mozart:
  - Obj = {NewStat Cls Init}
- Concurrent object invocations stay concurrent at home site
- Exceptions are correctly passed back to invoking site
- Object references in messages automatically become remote references
Comparison with Java RMI

• Lack of transparency
  – Java with RMI is only network transparent when parameters and return values are stateless objects (i.e., immutable) or remote objects themselves
    • otherwise changed semantics
  – Consequence
    • difficult to take a multi-threaded centralized application and distribute it.
    • difficult to take a distributed application and to change the distribution structure.

• Control
  – Compile-time decision (to distribute object)
  – Overhead on RMI to same machine
  – Object always stationary (for certain kinds of application - severe performance penalty)

• Ongoing work in Java Community
  – RMI semantics even on local machine
  – To fix other transparency deficiencies in RMI
  – Java Enterprise beans within a cluster
Notation for the distributed protocols

- We will use a graph notation to describe the distributed protocols.
- Each language entity (record, closure, dataflow variable, thread, mutable state pointer, class) is represented by a node.
- Distributed language entities are represented by two additional nodes, proxy and manager. The proxy is the local reference of a remote entity. The manager coordinates the distributed protocol in a way that depends on the language entity.
- For the protocols we will show, we have proven that the distributed protocol correctly implements the language semantics (see publications).
« Active » object

• Variant of stationary object where the home object always executes in one thread
• Concurrent object invocations are sequentialized
• Use is transparent: instead of creating with NewStat, create with NewActive:
  – Obj = {NewActiveSync Cls Init}
  – Obj = {NewActiveAsync Cls Init}
• Execution can be synchronous or asynchronous
  – In asynchronous case, any exception is swallowed; see later for correct error handling
Cached (« mobile ») object (1)

- For collaborative applications, e.g., graphical editor, stationary objects are not good enough.
- Performance suffers with the obligatory round-trip message latency
- A cached object moves to each site that uses it
  - A simple distributed algorithm (token passing) implements the atomic moves of the object state
  - The object class is copied on a site when object is first used; does not need to be copied subsequently
Cached (« mobile ») object (2)

- Heart of object mobility is the mobility of the object’s state
- Each site has a state proxy
- Object state moves atomically to each site that requests it
- Let’s see how the state moves
Another site requests the state
It sends a message to the manager, which serializes all such requests
The manager sends a forwarding request to the site that currently has the state

Cached (« mobile ») object (3)
Cached (« mobile ») object (4)

- Finally, the requestor receives the object state.
- All subsequent execution is local on that site (no more network operations).
- Concurrent requests for the state are sent to the manager, etc., which sequentializes them.
Cached (« mobile ») object (5)

- Let’s look at the complete object
- The complete object has a class as well as an internal state
- A class is a value
  - To be precise, each object has a closure that references both the class code and the state proxy
- Classes do not move; they are copied to each site upon first use of the object there
Invalidation-based object (1)

- An invalidation-based object is optimized for the case when object reads are needed everywhere and object writes are rare (e.g., virtual world updates).
- A state update operation is done in two phases:
  - Send an update to all sites
  - Receive acknowledgement from all sites
- Object invocation latency is 2 network hops, but depends on the slowest site.
A new site that wants to broadcast has first to invalidate the previous broadcaster.

If several sites want to broadcast concurrently, then there will be long waits for some of them.
Asynchronous FIFO
stationary object

• Synchronous object invocations are limited in performance by the network latency
  – Each object invocation has to wait for at least a round-trip before the next invocation

• To improve performance, it would be nice to be able to invoke an object asynchronously, i.e., without waiting for the result
  – Invocations from the same thread are done in same order (FIFO)
  – But this will still change the way we program with objects

• How can we make this as transparent as possible, i.e., change as little as possible how we program with objects?
  – Requires new language concept: dataflow variable
  – In many cases, performance can be improved with none or minor changes to an existing program
Dataflow variables (1)

• A dataflow variable is a single-assignment variable that can be in one of two states, unbound (the initial state) or bound (it has its value)

• Dataflow variables can be created and passed around (e.g., in object messages) before being bound

• Use of a dataflow variable is transparent: it can be used as if it were the value!
  – If the value is not yet available when it is needed, then the thread that needs it will simply suspend until the value arrives
  – This is transparent to the programmer
  – Example:


\[ \text{thread } X=100 \text{ end} \quad \text{Y} = X+100 \]

(binds X) \quad \text{(uses X)}

• A distributed protocol is used to implement this behavior in a distributed setting
Dataflow variables (2)

- Each dataflow variable has a distributed structure with proxy nodes and a manager node.
- Each site that references the variable has a proxy to the manager.
- The manager accepts the first bind request and forwards the result to the other sites.
- Dataflow variables passed to other sites are automatically registered with the manager.
- Execution is order-independent: same result whether bind or need comes first.

Bind request: \( X=100 \)

Needs variable: \( Y=X+100 \) (suspends)
Dataflow variables (3)

- When a site receives the binding, it wakes up any suspended threads.
- If the binding arrives before the thread needs it, then there is no suspension.

Bind request:
X = 100

Needs variable:
Y = X + 100
(suspends)
Dataflow variables (4)

• The real protocol is slightly more complex than this (but not much more)
  – What happens when there are two binding attempts: if second attempt is erroneous (conflicting bindings), exception is raised on guilty site
  – What happens with value-value binding and variable-variable binding: bindings are done correctly (operation is called « unification »)

• Optimization for stream communication
  – If bound value itself contains variables, they are registered before being sent
  – This allows asynchronous stream communication (no waiting for registration messages)
Dataflow variable and object invocation (1)

• Similar to an active object
  – Return values are passed with dataflow variables:

    C={NewAsync Cls Init} (local)

    {C get(X1)}
    {C get(X2)}
    {C get(X3)} (remote)

• Can synchronize on error
  – Exception raised by object:
    {C get(X1) E} (synchronize on E)

  – Error due to system fault (crash or network problem):
    • Attempt to use return variable (X1 or E) will signal error (lazy detection)
    • Eager detection also possible
Dataflow variable and object invocation (2)

Need values

Call synchronously when needed

Need values

Call asynchronously when needed

Use values

Call asynchronously before needed

Use values

Copyright © 2002 by P. Van Roy. All rights reserved.
Transactional object

• Only makes sense for a set of objects (call it a «transactional store»), not for a single object
• Does both latency tolerance and fault tolerance
  – Separates distribution & fault tolerance concerns: the programmer sees a single set of objects with a transactional interface
• Transactions are atomic actions on sets of objects. They can commit or abort.
  – Possibility of abort requires handling speculative execution, i.e., care is needed to interface between a transactional store and its environment
• In Mozart, the GlobalStore library provides such a transactional store
Fault tolerance

- **Reflective** fault detection
  - Reflected into the language, at level of single language entities
  - For now: permanent process failure and temporary network failure
  - Both synchronous and asynchronous detection
    - Synchronous: exception when attempting language operation
    - Asynchronous: language operation blocks; user-defined operation started in new thread
      - Our experience: asynchronous is better for building abstractions

- **Fault tolerance**
  - Build abstractions using reflective fault detection
  - Example: *transactional store*
    - Set of objects, replicated and accessed by transactions
    - Provides both fault tolerance and network delay compensation
    - Lightweight: no persistence, no dependence on file system
Distributed garbage collection

- The centralized system provides automatic memory management with a garbage collector (dual-space copying algorithm).
- This is extended for the distributed setting:
  - First extension: weighted reference counting. Provides fast and scalable garbage collection if there are no failures.
  - Second extension: time-lease mechanism. Ensures that garbage will eventually be collected even if there are failures.
- These algorithms do not collect distributed stateful cycles, i.e., reference cycles that contain at least two stateful entities on different processes.
  - Algorithms for collecting these are complex.
  - So far, we find that programmer assistance is sufficient (e.g., dropping references from a server to a no-longer-connected client). This may change in the future as we write more extensive distributed applications.
Implementation status

• All described protocols are fully implemented and publicly released in the Mozart system
  – Including stationary, cached mobile, asynchronous, and transactional object
  – Except for the invalidation-based object, which is not yet implemented
Conclusion and ongoing work

• With proper language semantics, network transparency becomes practical
  – Separation of functionality, distribution, and fault tolerance
  – More fault tolerance abstractions are being developed (better separation of concerns)
  – Study fundamental limits of network-transparent distributed computing

• Ongoing work: simplifying building distributed applications
  – Hook distribution and fault tolerance into the user interface with distributed widgets
  – Just a few lines of code for many fault-tolerant distributed applications

• Ongoing work: improved network layer
  – Visualization tool for observing all network behavior at high level of abstraction
    (« Distribution Panel » in Mozart 1.2.0)
  – Fine-grained multi-channel transport protocol

• Ongoing work: security
  – Capability security at the language level, supported cryptographically by implementation
  – Related to work on E language and system (Mark Miller et al)

• Projects starting in high availability, security, and peer-to-peer computing
  – We are looking for good people to join our team