

A Fault Tolerant Abstraction for Transparent Distributed Programming*

Donatien Grolaux¹, Kevin Glynn², and Peter Van Roy²

¹ CETIC asbl, Rue Clément Ader 8,
B-6041 Charleroi, Belgium
dg@cetic.be

² Université catholique de Louvain,
Département d'Ingénierie Informatique,
B-1348 Louvain-la-Neuve, Belgium
{glynn, pvr}@info.ucl.ac.be

Abstract. This paper introduces a network fault model for distributed applications developed with the Mozart programming platform. First, it describes the fault model currently offered by Mozart, and the issues that make this model inconvenient for building fault-tolerant applications. Second, it introduces a novel fault model that addresses these issues. This model is based on a localization operation for distributed entities, and on an event-based mechanism to manage network faults. We claim that this model 1) is much better than the current one in all aspects, and 2) simplifies the development of fault-tolerant distributed applications by making the fault-tolerant aspect (largely) separate from the application logic. A prototype of this model has been developed on the existing Mozart platform. This prototype has been used on real applications to validate the aforementioned claims.

1 Introduction

With the Internet distributed applications have become commonplace, and software environments have adapted to offer adequate programming support. At first, TCP/IP offered a reliable communication channel between two processes on remote computers to exchange information in byte form. Then, different schemes were invented to further abstract from the network layer: remote procedure calls (or method invocations), message based and event driven communication mechanisms (examples are Java's Remote Method Invocation [1], Web Services, and Erlang [2]), peer to peer communication patterns (e.g., JXTA [3]), and the transparent distribution of language entities (e.g., Mozart [4]). Since the Internet is not a reliable environment where all components are constantly available, these

* The first author was funded at CETIC by the Walloon Region (DGTRE) and the E.U. (ERDF and ESF). The second author was funded by European project PEPITO IST-2001-33234.

approaches also provide mechanisms for dealing with network faults. This paper focuses on the Mozart system, an implementation of the Oz language, which provides transparent distribution of language entities. We assume the reader is familiar with the Oz language (tutorials are available from the Mozart web site [5], or an overview is provided in Chapter 1 of Van Roy and Haridi's text book [6]).

Section 2 introduces Mozart's distributed implementation of Oz. In Section 3 the current fault model of Mozart, which is based on the operations on language entities, is described and criticised. In Section 4 we propose a new approach based on the distributed entities, rather than the operations, and demonstrate its advantages. Finally, in Section 5 we conclude by describing a prototype version of this new model, a fault tolerant application written using it, and preliminary conclusions we can draw from this experience.

2 Transparent Distribution of Oz

The Mozart implementation of Oz offers distributed programming by attaching distributed protocols to the language entities [7]. In essence, a single Oz store is shared between the different inter-connected Oz processes. There is no explicit notion of communication channel between the processes at the language level. From the Mozart programmer's point of view, as long as there are no communication problems, there is no difference in operating on entities locally or remotely¹. Similarly, a thread running in a remote process is equivalent to a local thread.

In practice, two or more processes have to share at least one common reference if they want to communicate. Once an entity is shared it can be used to introduce more shared entities. A typical example is a shared port: the output stream of the port stays local to the site that created the port and a reference to the port is shared with other processes. They can use the port reference to send information to the output stream of the port, and this information can include references to free variables, other ports, cells, locks, and so on. When exchanging a non-scalar reference a distributed protocol is transparently attached to the local entity (if not already done) to turn it into a global entity.

Of course, this mechanism requires a first reference to be exchanged between the different processes as a bootstrap mechanism. A built-in Mozart operation maps a local reference into a global identity, accessible by a unique, universal, human-readable name. Any process that knows this name can use it to create a reference to the original entity. The name can be transferred by any means, including voice call, email, web pages, and so on.

Figure 1 shows this mechanism being used to share variables between two Mozart processes. Process 1 exports a reference to the variable `A` as a text string. This string is given (by mail, phone, web interface, and so on) to process 2 which imports the variable as `B`. Now both sites have a shared virtual store. Process 2

¹ Assuming the entity does not refer to localised resources (local file system, keyboard or display, for example). We assume this to be the case in this paper.

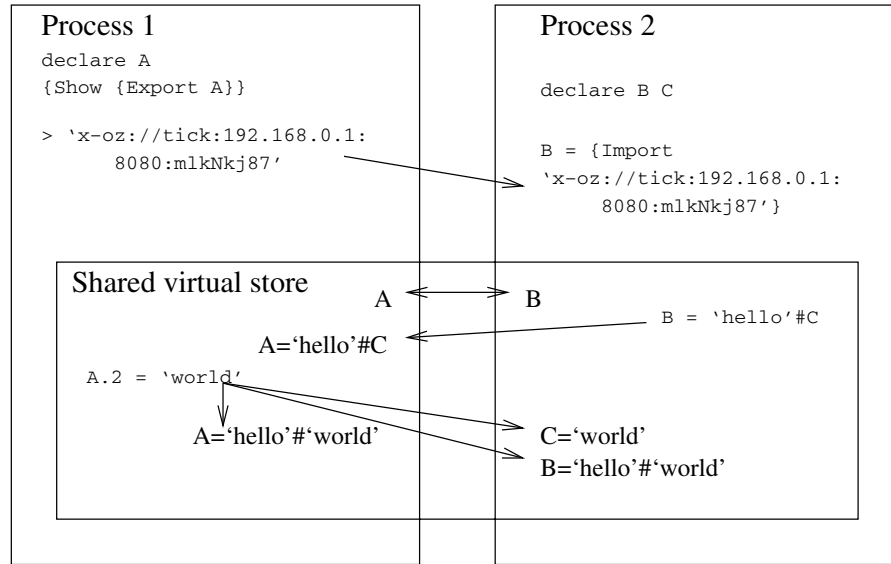


Fig. 1. Transparent Distribution in Oz

assigns a value to B containing a reference to C ($B = \text{'hello'}\#C$); the shared virtual store reflects this assignation in Process 1 too. Now C is shared by both sites as they both have a reference to it. Process 1 indirectly assigns a value to C ($A.2 = \text{'world'}$) and this is reflected back to Process 2.

3 Fault Tolerance Management

As already explained, a remote entity is equivalent to a local entity as long as the communication between the different processes is never interrupted. This condition is not respected on real networks like the Internet. To achieve fault tolerance at the application level the network problems must be made visible at the language level. This breaks the transparency of the distribution: an operation that cannot fail on a local entity can fail on a remote one because of network problems. Breaking the transparency is unfortunate, but unavoidable². The goal of our proposal is to minimize the cost of adding fault management to real distributed applications.

3.1 Fault Tolerance Management in Mozart

First, we take a look at the current Mozart implementation and the mechanisms offered for managing network faults. There are two aspects involved:

² In the context of network transparency as provided by Oz. Other schemes use dedicated data structures and operations for remote entities where failure of communication is supported by the data type (for example, remote procedure calls).

1. The health of a distributed entity can be `ok`, `permFail`, or `tempFail`:
 - `ok`: operations on the entity are applied successfully for now.
 - `permFail`: the entity is in a permanent state of failure; no operation on the entity can succeed, ever. A `permFail` is definitive. `permFails` are detected only when the remote computer can certify the death of the process. This is usually only possible on local area networks when a process crashes; if the whole computer crashes, or no further communication is possible then the `permFail` cannot be detected.
 - `tempFail`: the entity is in a temporary state of failure. For now, the system cannot successfully apply operations on the entity but that may be possible in the future. A `tempFail` may turn into a `permFail`, may disappear, or may stay forever. Most network faults over the Internet produce a never-ending `tempFail`, instead of a `permFail` because of the lack of information about the remote site.

An application can trigger a computation in a separate thread when an entity changes its state (these computations are called *watchers*)³.

2. The behaviour of operations on distributed entities when there is a network fault. There are three possibilities:
 - Suspend the operation. This ensures that a distributed application that suffers from network problems does not do anything unwanted; instead all distributed operations block. In the case of a `tempFail` that eventually resolves itself the suspended operations are automatically resumed.
 - Raise an exception.
 - Replace the operation (message send, variable binding, and so on) by a user defined one. For example, a client could be configured to automatically switch to another server and retry the failed operation.

An entity can be configured to have different behaviours for `tempFail` and `permFail` conditions. For example, operations may block during a `tempFail` and raise an exception during a `permFail`.

The idea behind this model is that Mozart programmers should first develop their applications centrally using threads to simulate the different processes. Once the application is locally complete and correct, the threads are taken away to become real processes on remote computers. At this stage, in the absence of network problems, the distributed application is already working correctly because of the network transparency. The designers of Mozart's distributed mechanisms assumed that fault tolerance could then be straightforwardly achieved by configuring each of the distributed entities to react correctly to failures.

In practice, this goal is almost impossible to achieve with Mozart's fault model for several reasons:

Network Transparency Conflicts With Modularity. With Mozart's current fault model entities change their semantics upon network problems. De-

³ Oddly this is not possible for the transition from `tempFail` to `ok` in the current Mozart implementation; so this mechanism cannot be used to trigger a computation when an entity is working again.

pending how the entity's fault management is configured, an operation on an entity might raise an exception, block, or execute an arbitrary piece of code, none of which it would ever do in the centralized case. This is a deep change of the entity's semantics that breaks the modularity of the language. A module is composed of a public interface and a private implementation; a module's user should only have to know the public interface. If a module is written for localized entities, it may not work in a distributed environment.

In practice, it is often necessary to understand the inner workings of code which we want to reuse for distributed entities and it is depressingly common that we must rewrite that code to make it fault-tolerant.

We demonstrate the difficulty of writing modular, transparent, fault-tolerant applications in the case study below.

Misleading Feedback from Asynchronous Operations. Probably the most heavily used communication scheme used by Mozart applications is to make remote procedure calls through an Oz port, using a free variable to hold the response from the other site. The fault detection provided by Mozart when sending a message to a Port is inadequate: the operation is intrinsically asynchronous but the fault detection mechanisms of Mozart are synchronous. Consequently, a `Port.send` operation might act as if it were successful when, in fact, the link is already down but not yet detected by Mozart. Similarly, a `Port.send` operation might fail when, in fact, the link is back up again but not yet detected by Mozart. As a general rule, the only way to be sure an asynchronous message was sent successfully is to have an acknowledgement protocol, i.e., to introduce some synchronization. Since the application must handle this anyway it is preferable that the `Port.send` operation should always succeed (as in Erlang), regardless of the current state of the network.

Lack of Control at the Application Level. Distributed applications can make good use of knowledge about the status of network connections. For example, to give feedback to the user, to allow the user to cancel a computation that is suspended due to a network problem, and so on. This is currently difficult to achieve.

Additionally, operations that are automatically retried by Mozart cannot be cancelled, they will be retried forever (until the whole process is killed). Unfortunately, it is not possible to completely avoid situations where that might happen.

This does not mean that it is impossible to write well-behaved, fault-tolerant Mozart applications. Several successful applications have been written using this model. However, in our experience, they circumvent the difficulties by hiding them in abstractions that offer a limited communication channel, and poor transparency, in exchange for nicer fault awareness and management. As a consequence, the transparency distribution of language entities is not directly used and it completely nullifies the benefits of transparent distribution in Mozart.

3.2 Case Study: A Simple Problem Requiring a Complex Solution

In this section we describe a simple distributed client server application. The server makes use of a procedure that was originally written for a centralized environment. Ideally, we would like to make the client-server application fault-tolerant without rewriting the given procedure, and without knowing how it is implemented. We show that this is not possible with Mozart's fault model.

For our case study we assume that we have a pair of Oz procedures, `Add` and `GetValue`. `Add` takes an integer as argument and adds it to the contents of an internal mutable cell shared only by `Add` and `GetValue`. A call to `GetValue` binds its argument to the current value of the shared cell.

If the argument to `Add` is an unbound variable then `Add` creates a thread to add the argument once it is bound. This thread only waits 3 seconds, if the variable is not bound in this time then it is bound to the atom `'ignored'` so that the caller can see that it was unsuccessful. In all cases, a call to `Add` returns without waiting or blocking.

It is simple to turn `Add` into a distributed server; we just create a port and call `Add` for all the entities received on the port's stream:

```
P={NewPort S}
thread {ForAll S Add} end
```

Here is a possible implementation of the `Add` and `GetValue` procedures:

```
local Counter={NewCell 0} in
  proc {Add V}
    if {IsDet V} then Old New in % do addition immediately
      {Exchange Counter Old New}
      New=Old+V
    else TimeOut in
      % run a thread to timeout the wait
      thread {Delay 3000} TimeOut=unit end
      % run a thread to wait for timeout or the value
      thread
        {WaitOr V TimeOut}
        % might have been already bound
        try V=ignored catch _ then skip end
        if V\=ignored then Old New in
          {Exchange Counter Old New}
          New=Old+V
        end
      end
    end
  end
  proc {GetValue V} V = @Counter end
end
```

If the entity sent to the server is already determined then it is immediately added to the internal `Counter`. If an unbound variable is sent then we create two threads and return. The first thread waits 3 seconds and binds the `TimeOut`

variable. The second thread waits until either `TimeOut` or the variable are bound, then it binds `v` to `ignored` inside a `try ... catch` statement (if `v` was already bound then this statement has no effect, the unification error exception will be thrown away by the `try .. catch`). Finally, if `v` is not bound to `ignored` we add it to the internal counter. The additions to the counter are performed by `Exchange` which is atomic with respect to other threads attempting to update the counter.

In the absence of communication errors, this implementation is correct. Here are some possible calls from clients:

```
% Immediately adds 10 to the internal cell
{Port.send P 10}

% waits 2 seconds then adds 10 to the internal cell
local X in {Port.send P X} {Delay 2000} X=10 end

% waits 10 seconds, there will be a unification error
% as X will have been bound to 'ignored' already
local X in {Port.send P X} {Delay 10000} X=10 end
```

Now, consider making this application fault tolerant. Suppose a remote client sends a variable `x` to the server's port, as we have described previously we have a number of options:

We can make operations on `x` block. If the client has a `tempFail` or `permFail` condition then we might block in `Add` on the `IsDet` operation. In the case of a `permFail` we will block for ever, in the case of `tempFail` we may or may not eventually proceed. While `Add` is blocked the server is unable to service other clients. This is unacceptable, we would like the server to ignore this client and continue servicing other requests.

We can make operations on `x` raise an exception. The application may raise an exception when performing the `v\=ignored` test. This exception is not caught by the thread doing the test. In Oz, exceptions that are not caught by a thread terminate the whole application, so again this is unacceptable.

Finally, we can replace the operations on `x`. This mechanism replaces low-level, primitive operations on a distributed entity. For a variable the primitive operations are `bind`, `wait`, and `isDet`. For our application the only sensible replacement action would be to throw an exception, which we have already discussed. But even if we could modify this mechanism to give more control over the operations replaced, we can easily get problems. Suppose, for `tempFail` we replace the binding `v=ignored` with a `skip` operation. If the `tempFail` then immediately disappears we will block forever when testing if `v\=ignored` because `v` will be unbound!

In summary, it is impossible to make `Add` fully fault-tolerant without modifying it. Further, it is often necessary to understand the implementation of code in order to make it successfully tolerant to network failures.

The issues raised by this `Add` example (and more!) appear often when building real, distributed, fault-tolerant applications. In practice Mozart's current fault model is often difficult, and sometimes impossible, to use successfully.

4 A New Approach to Fault Tolerance Management

We have observed that fault management implies a change in the semantics of shared entities. The existing model spreads this semantic change to every operation on the entities in a very drastic way: temporary or infinite suspension of threads; exceptions where none was possible before; a completely new definition of the operation. To write an application that automatically recovers from network faults these semantic changes spread like a plague: every single distributed operation has to be taken care of, and asynchronous operations require complex workarounds.

We propose a model that minimizes the semantic impact of network faults on shared entities. We consider that shared entities are local entities synchronized with compatible local entities in remote processes. Entities act as a single global entity when there are no network problems. If this synchronisation fails then the synchronisation is dropped and the entity becomes an unsynchronised local entity. It is also possible for an application to explicitly remove the synchronization from an entity with the same result. If it is not possible to determine if the synchronisation is currently working (for example, because of a `tempFail`) then operations on the synchronised entity are suspended until it is working again, it fails, or the synchronization is explicitly dropped. In essence, the operations on the entities always keep their local semantics: if possible, operations are applied globally, otherwise they are applied only locally. If it is not possible to determine between these two then they are suspended until this determination is achieved.

This approach removes the need to define replacement operations for distributed operations, but it is not enough to achieve fault-tolerance. Applications need awareness about the synchronisation state of an entity in order to manage failures. In our approach we associate a possibly infinite list of states with each entity. `DistStates={Watch X}` assigns to the variable `DistStates` the list of states of entity `x`. The possible states when there are no network problems are:

`'local'` : the entity is not synchronized with any other entity.
`shared` : the entity is synchronized with at least one other entity in a remote process.

Figure 2 demonstrates the progression of distribution states with an example. Reading clockwise from the top left hand corner, the variable `v` is created in process 1 which then shares it with process 2. Process 2 terminates, and `v` is now local to process 1 again. Another process, process 3, shares `v` with process 1. `v` is assigned the atom `foo`. As scalar values are copied and do not need to be shared any more (scalar values are invariant over time), `v` becomes local for both process 1 and process 2 and will stay local forever.

By observing the change between these two states, an application can guess if a communication with a remote site has succeeded or not (in the above example, process 2 terminated itself and `v` became local again). However, it is not very easy to manage faults based on only this information:

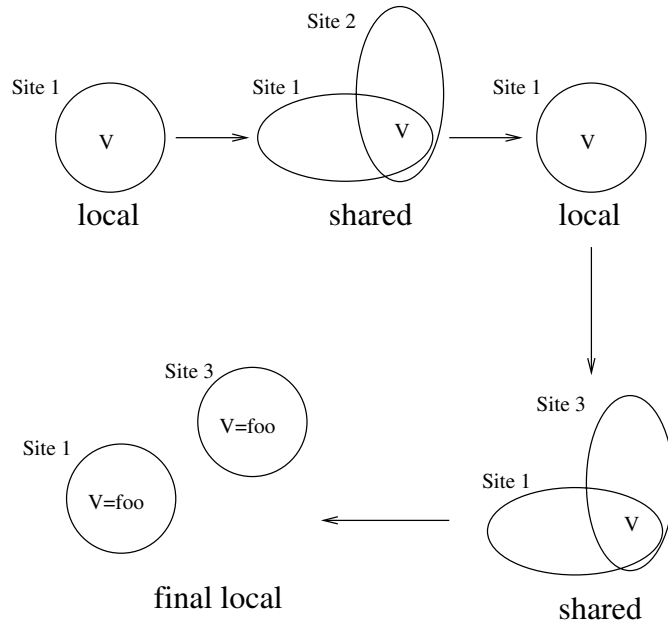


Fig. 2. Distribution States

1. The state stream is concurrent to the execution of the application; consequently, when an operation is applied on an entity that changes its state from `shared` to `local` it is not possible to know whether the operation was applied globally (the change occurred right after the operation) or only locally (the change occurred right before the operation).
2. It is important that the transparency property should go both ways: as well as turning a local application into a distributed one, we should also be able to turn a distributed application (or parts thereof) back into a local one. In particular, the modifications introduced to support fault tolerance in the distributed case should not break the application when it is run centralized. If the fault detection is based on the local/shared state then when run locally the application may think that the entity constantly has network problems.

Two more states are introduced to better manage network problems:

`shared(suspend)`: the entity is synchronized with at least one other entity in a remote process. However, due to a network problem, it cannot function globally for now. While this state lasts all operations on the entity block. When (if) the network problem disappears the blocked operations will be automatically resumed.

`dirty`: the entity used to be synchronized with at least one other entity in a remote process but a network problem, or an explicit operation, terminated that synchronization. Operations on the entity block while this state lasts.

Applications use the following operations to implement fault tolerance:

1. The `Break` operation takes an entity as parameter and puts it into the `dirty` state, whatever its original state. All remote synchronizations of the entity are dropped. By calling `Break` the application can force an entity from the `shared(suspend)` state.
2. The `Clean` operation takes an entity as parameter. If this entity is in the `dirty` state then it turns it into the `'local'` state, otherwise it does nothing. All operations that were blocked on the dirty entity resume on the local copy.
3. The `watch` function returns the list of states of an entity, starting with its current state; new elements are added at each change of state. This list remains open as long as the current state can change; it is closed when the current state is final (as in Fig.2 above).
4. The `waitCond` procedure blocks until the condition specified as parameter is fulfilled. The operation can specify complex conditions about the states of one or more entities. We do not provide further details here, but an example is shown below in the definition of `FaultTolerantAdd`.

Applications should be written to support the disconnection / unavailability of remote sites. This could happen at any time; threads working on entities shared with the remote site will block on them. Concurrently, watcher threads waiting (by using `waitCond`) on the `shared(suspend)` and/or `dirty` state changes will be scheduled. If appropriate, the watcher thread can take whatever action is needed to cleanly separate the entity or application from the dead site. It can then resume the blocked operations by localizing the entities (using `Clean`). The work being done by threads on these broken entities is probably lost as they are now working on dummy local entities instead of the real globalized ones, but importantly they usually do not require to be adapted from their centralized version.

Fault detection via `shared(suspend)` and `dirty` does not break backward transparency (centralizing a previously distributed operation): in the local case these states never occur and the recovery actions are never triggered.

A fault tolerant version of `Add` in the case study (Section 3.2) would be:

```

proc {FaultTolerantAdd V}
  thread
    if {WaitCond 'or'(dirty(V) suspend(V) det(V))}\=det(V)
      then
        {Break V} {Clean V}
        try V=0 catch _ then skip end
      end
    end
  {Add V}
end

```

The server should call this procedure instead of `Add`. The additional thread waits for `v` to be bound, or to go into a network problem state. If it detects that `v` has a problem (`dirty` or `suspend`), then it breaks and cleans `v`, to make it

‘local’, and binds it to 0 so that this call to `Add` will not change the internal cell. The binding is done in a `try ... catch` in case the `Add` procedure has already bound `v` to `ignored` itself.

4.1 Properties of This New Model

1. It is possible to prevent distributed operations from suspending forever. Suspension is useful when one wants the system to retry the operation for some time. However, the retrying can be stopped using `Break` and `Clean`.
2. No semantic change to the entities. Contrary to the current Mozart model, a distributed entity cannot raise additional exceptions or execute arbitrary code compared to its local version. It can cause a suspension, but that suspension can be resumed externally. Consequently, in most situations, code written for local entities will also run unchanged for distributed ones. When a network failure occurs it is possible to complete running the code on a local version of the entity (to keep the current flow of the application) then take specific actions to recover from the failure. When this model is used efficiently the management of the failure can be achieved once per distributed entity, in a dedicated thread orthogonal to the rest of the application.
3. Asynchronous operations always succeed: detection of network problems is orthogonal to these operations. (Although they may block: asynchronous operations will be queued until sent or discarded, eventually an operation may block due to the unavailability of local resources such as buffers).
4. This model makes a trade-off. It simplifies applications by retaining local semantics for distributed entities, but possibly performs unnecessary computations on dummy entities. It is a fair trade-off for many cases where the rate of network faults is small or if the unnecessary computations are too small to be of any importance. If large, unnecessary computations may occur, it may be necessary to modify the application so that network fault detection can prematurely terminate the thread(s) of the computation.
5. The application has complete control over its communication channels. In particular, it may stop working with other sites at any time by breaking the distributed entities they are sharing.
6. This model is independent of the protocols used to synchronize distributed entities. In particular, if a protocol requires one or more particular communication links to run correctly this requirement does not appear directly at the application level. Instead, the distributed entity will switch from `shared` to `shared(suspend)` or `dirty` depending on the capacity of the underlying protocol to maintain its synchronization. If the protocols are changed then the application will not have to be adapted to reflect these changes.
7. The principle of a local entity synchronized with other local entities is independent of the type of entity being shared. As a result, this approach can be applied to any of Mozart’s distributable entities, such as cells or locks. For example, in this model a cell shared among several sites can be broken then cleaned by any of the sharing sites. This site then has a local cell that contains the last locally known state of the distributed cell.

5 Conclusion and Future Work

A partial prototype version of this model, supporting ports and logic variables, has been implemented on top of Mozart's existing fault model. Due to restrictions in the current model it cannot directly use the usual Oz data types and operations because of the blocking requirement of the dirty state. Consequently, the prototype implements its own set of data types and associated operations.

This prototype was used to make a rather complex peer-to-peer algorithm fault tolerant. The algorithm was first written in a local setting, using threads to simulate nodes. It was extended to recover from the disappearance of any node in the network: this is the application-specific part which manages fault recovery. Then the node threads were turned into remote processes: this is straightforward thanks to the transparent distribution of Mozart. Finally, the algorithm was extended to add fault detection to the distributed entities. A watcher thread cleans faulty entities and tells the application to eject the corresponding node. This thread is a dozen lines of code, orthogonal to the remaining 5000 lines of code. The resulting application is fully fault-tolerant. This example validates the proposed approach, and shows it is possible to have a fault model that keeps both the modularity and transparency properties of distributed computing.

Due to space constraints this paper has omitted many details. A follow up paper will describe the implementation of the fault tolerant mechanisms for Mozart, we will address the problem of the identity of broken entities, and also we will describe how the different data types of Oz behave in case of faults.

We have described a novel fault tolerant abstraction for a language with transparent distribution and demonstrated that it solves fundamental problems that exist in Mozart's current fault tolerant abstraction.

References

1. Grosso, W.: Java RMI. O'Reilly (2001)
2. Armstrong, J., Viriding, R., Wikström, C., Williams, M.: Chapter 6, Chapter 8. In: Concurrent Programming in Erlang. 2 edn. Prentice Hall (1996)
3. Flenner, R., Abbott, M., Boubez, T., Boubez, T., Cohen, F., Krishnan, N., Moffet, A., Ramamurti, R., Siddiqui, B., Sommers, F.: Java P2P Unleashed: With JXTA, Web Services, XML, Jini, JavaSpaces, and J2EE. Sams Publishing (2002)
4. Haridi, S., Van Roy, P., Smolka, G.: An overview of the design of Distributed Oz. In: Proceedings of the Second International Symposium on Parallel Symbolic Computation (PASCO '97), Maui, Hawaii, USA, ACM Press (1997) 176–187
5. The Mozart Consortium: Mozart documentation (2004) Available at <http://www.mozart-oz.org/documentation/>.
6. Van Roy, P., Haridi, S.: Chapter 1. In: Concepts, Techniques, and Models of Computer Programming. The MIT Press (2004)
7. Van Roy, P., Haridi, S., Brand, P.: Distributed programming in Mozart – a tutorial introduction (2004) Available at <http://www.mozart-oz.org/documentation/>.