Improving transparency of a distributed programming system

Boris Mejías[1], Raphaël Collet[1], Konstantin Popov[2], and Peter Van Roy[1]

[1] Université catholique de Louvain, Louvain-la-Neuve, Belgium
{bmc, raph, pvr}@info.ucl.ac.be
[2] Swedish Institute of Computer Science, Stockholm, Sweden
kost@sics.se

Abstract. Since Grid computing aims at creating a single system image from a distributed system, transparent support for distributed programming is very important. Some programming systems have attempted to provide transparent support for distribution hiding from the programmer several concerns that deal with the network behaviour. Sometimes, this leads to restricted models that attach programmers to the decisions of language designers. This work propose language abstractions to annotate entities specifying their distributed behaviour while keeping the transparency of the distribution support, and a failure model that separates the application's logic from failure handling.

1 Introduction

It is well known that distributed programming is hard, and large research effort has been devoted to simplify it. One of the approach is transparent distribution, where the distributed system appears to the user as a single system image. In such environment, semantics of operations performed in a centralised system remain the same in a distributed one. Even though, distributed programming is plagued with network failures, lack of global time and other problems shown in [1] that make full transparency infeasible. However, transparent distribution can be achieved in several degrees.

In transparent distribution, the consistency of the system is maintained using predefined access architectures and protocols, which are designed depending on the semantics of each type of entity. These distribution strategies cannot be modified by the programmer, who is limited to use the default design. We propose a way to annotate language data structures, here referred as entities. These annotations will allow programmers to choose between several strategies for every entity according to the non-functional requirements of their programs.

Annotating entities in peer-to-peer networks becomes very helpful to keep the consistency of their states. Knowing that the peer that is currently hosting an entity can leave the network at any time, one may annotate it with a migratory strategy. As soon as another peer request access to it, the entity will migrate to the requesting peer, and the original host can safely leave. Regarding distributed garbage collection, a peer can chose a persistent strategy to keep an entity alive even when temporary there is no distributed reference to it.

The platform we are using to test our concepts is based in the Distribution SubSystem (DSS) [2], which is a language independent middleware for efficient distribution support of programming systems. Working in collaboration between SICS and UCL, we have integrated the DSS into the Mozart system[3, 4], which provides a state of the art in transparent distribution. Inspired by the work presented in [5], we also propose a new fault model to improve fault handling in a more factorised way. This fault model, together with the election of right distribution strategies, can strongly help in design and implementation of decentralised application, such as Peer-to-Peer networks and Grid-based applications, where network failures are very common.

Our proposal will be directly applied in the support of the Peer-to-Peer networking library, P2PS[6], but the results is not limited to that. Existing programming models for Grid at best support failure handling at the level of services or components, but not at the granularity level of individual operations and individual data items. Such fine-grained failure handling is important in particular for building high-performance Grid-based applications that must react to changes in the network promptly, without reconfiguring distributed Grid services involved in the application. We believe that this work helps in the achievement of that required granularity.

2 Entity Annotations

Let us consider the example of a cell with a state that can be updated. To maintain its state consistent over the distributed system, several protocols are provided such as "migratory" or "stationary". In the migratory protocol, the state migrates to the site that is performing the operation. In the stationary protocol, the operation moves to the site where the state resides, and only the result of the operation travels back. We may also take into account the algorithm that maintains the distributed references to the entity, to do a proper garbage collection. A programmer could ask for a reference counting or a time-lease based mechanism, or even force the entity to never become garbage. It is also possible to parametrise the communication architecture of the group of sites referring to a given entity.

Currently, there is no way to choose arbitrarily which protocol to use. The decision is made by language designers and programmers are limited to it. We provide a language abstraction to *annotate* the entity deciding its distribution strategy. The following formal semantics were presented in the Nordic Workshop of Programming Theory (NWPT'05) [7], and they represent a strong base for our development of applications. The operational semantics of the abstraction are expressed using reduction rules as

$$\frac{S \,\|\, S'}{\sigma \,\|\, \sigma'}\, C$$

where C is a boolean condition, *S* and σ are the statement and store before the reduction, and *S'* and σ' are the statement and store after the reduction.

Rule (1) defines the semantics to annotate an entity E with the annotation A. The annotation is a keyword with the name of the distribution strategy, such as "migratory", "stationary", "read_write_invalitation", etc. It can also be express as a record of the

form *a(prot:*P *arch:*A *gc:*G*)*, where P represents the protocol, A the communication architecture, and G the garbage collection algorithm.

$$\frac{\{\texttt{Annotate}\ E\ A\}}{\sigma} \Big\| \frac{\texttt{skip}}{annot(E,A) \wedge \sigma} \text{ if } \forall B : \sigma \models annot(E,B) \Rightarrow compat(A,B) \tag{1}$$

The rule allows incremental annotations over an entity as long as they define a relation of *compatibility*. Two annotations are compatible if they do not implies contradiction in the behaviour of each part of the distribution strategy. Then, an entity could be annotated it to have a migratory state and a reference counting algorithm for garbage collection, but, having a migratory state and a stationary state is of course not allowed. The system also provides default annotations to be used when the programmer does not specify any.

3 Operations over an annotated entity

Let us consider now the `Exchange` operator to read and write the state of a cell in only one step. In a local computation, the operation follows the semantic of rule 2, where `E` represents the entity, `Y` is the variable to be unified with the old value of the entity, and `Z` corresponds to the new value. The entity E is bound to the mutable pointer e, having its state represented with the statement $e{:}W$, where W corresponds to the current value. The operation is reduce to the the binding of Y with the old value W, where Z becomes the new current value of the cell.

$$\frac{\{Exchange\ E\ Y\ Z\}}{E{=}e \wedge e{:}W \wedge \sigma} \Big\| \frac{Y{=}W}{E{=}e \wedge e{:}Z \wedge \sigma} \tag{2}$$

Now we present the consecuence of annotating an entity. We are considering one single store for all the nodes involved in the network. We introduce indexes to specify the location of a particular statement, then, $(X{=}Y)_i$ describes a unification between entities X and Y that takes place in node i. The following rules define the distributed behaviour that implements the chosen protocol. Note that if indexes are removed, semantics remain the same, keeping distribution support transparent.

$$\frac{\{Exchange\ E\ Y\ Z\}_i}{annot(E,A) \wedge E{=}e \wedge (e{:}W)_j \wedge \sigma} \Big\| \frac{(Y{=}W)_i}{annot(E,A) \wedge E{=}e \wedge (e{:}Z)_i \wedge \sigma} \tag{3}$$

$$\text{if } A{=}migratory$$

$$\frac{\{Exchange\ E\ Y\ Z\}_i}{annot(E,A) \wedge E{=}e \wedge (e{:}W)_j \wedge \sigma} \Big\| \frac{(Y{=}W)_i}{annot(E,A) \wedge E{=}e \wedge (e{:}Z)_j \wedge \sigma} \tag{4}$$

$$\text{if } A{=}stationary$$

In rule 3, the entity has been previously annotated as “migratory”. Originally, the state $e{:}W$ resides in node j, and `Exchange` is invoked at node i. The operation will reduce to the unification of Y with the old value W at the same node where the operation was invoked. Due to the migratory annotation, the new state $e{:}Z$ will also move to node

i. Rule 4 is the equivalent operation having a "stationary" annotation. In this case, the result of the operation will travels back to node i, but the new state will remain in the original note j, because of its stationary strategy.

Note that in both rules, the reduction is a new operation at the invoking site, because the result travels back to the invoker, no matter where the operation of writing the new state is performed. The reduction of a unification will be entailed by the store as a global information, which is consistent with the statement $E{=}e$ of the previous rules.

$$\frac{(Y{=}W)_i \,\Big\|\, \sigma}{skip \,\Big\|\, Y{=}W \wedge \sigma} \text{ if } \sigma \models Y{=}W \tag{5}$$

4 Fault Model

We also propose improvements on the fault model of Mozart presented in [8]. Currently, if the distribution support of a certain entity presents a connection failure, an exception will be raised when an operation is performed and therefore, transparency will be unexpectedly broken. Let us consider the case of a piece of code that was written for a local execution that is used in a distributed program. This code reusing is perfectly reasonable because the system aims at transparency. The problem appears when a network failure is detected. The piece of code triggers a totally unexpected exception, making fault handling very hard to program.

We propose that operations silently block in case of failure. The operation is able to resume if the system recovers from the failure, or it might block forever. Fault handling is done in a concurrent thread which monitors the fault state of the entity, taking the correspondent actions. The advantage of that design is that pieces of local code will not modify their semantics when they are executed in a distributed environment. This improves the factorisation of fault handling. There is no full transparency at this level, but a higher degree is achieved.

Each distributed entity can be in one of three fault states: *ok* (no failure), *tempFail* (temporary failure), and *permFail* (permanent failure). The latter state means that the entity will never recover. Valid transitions between those states are defined by the automaton in Figure 1.

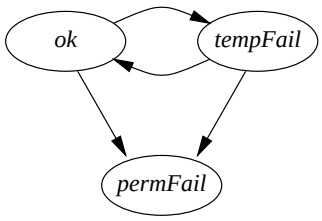

Fig. 1. Fault state transitions

In order to monitor fault states, each entity has a *fault stream*, which is a list of fault states. The system identifies failures in the communication between sites, and reports changes in an entity's fault state by extending its fault stream. Rule (6) models this, with $fs(E, s|sr)$ associating entity E with its fault stream $s|sr$. Rule (7) shows how to get access to the fault stream of an entity. The stream is always prefixed with the current fault state of the entity. Dataflow synchronisation automatically awaken a monitoring thread when the fault state of the monitored entity changes.

$$\frac{S \,\Big\|\, fs(E, s|sr) \wedge \sigma}{S \,\Big\|\, fs(E, s'|sr') \wedge sr{=}s'|sr' \wedge \sigma} \text{ if } s \rightarrow s' \text{ is valid transition} \tag{6}$$

$$\frac{\{\texttt{GetFaultStream}\ E\ S\}\ \Big\|\ S{=}s|sr}{fs(E,s|sr)\wedge\sigma\ \Big\|\ fs(E,s|sr)\wedge\sigma} \tag{7}$$

This model does not pretend to hide failures. Instead, it offers the possibility to treat them in a concurrent thread. The following code is an example of a thread monitoring a distributed entity. We first get the fault stream of the entity. Then, a thread is launched monitoring the fault stream, meanwhile another thread is performing operations over the entity.

```
FS = {GetFaultStream Entity}
thread {Monitor FS} end
thread <several operations over Entity> end
```

An example of a procedure that monitors the fault stream of any entity follows. The procedure receives the stream and try to do pattern matching with all possible states. If no failure is reported to the stream, the pattern matching will block. Once a fault state is matched, the procedure will take the correspondent action according to each case, and it will continue monitoring the rest of the stream. In the case of the example, temporal failure are treated with a time out.

```
declare
proc {Monitor Stream}
   case Stream of S|Sr then
      case S
      of ok then skip
      [] tempFail then
         {WaitOr Sr.1 TimeOut}
         <doSomething>
      [] permFail then
         <doSomething>
      end
   end
   {Monitor Sr}
end
```

5 Conclusions and Future Work

We have presented language abstractions that improve transparent distribution support. Entity annotation allows programmers to decide the distribution strategy that fits better to each entity, providing a granularity that cannot be achieved at the level of distributed systems based on components. These annotations do not break the transparency of the distribution support, helping to conceive the distributed system as a single image.

Being aware of network failures, which is the main limitation of distribution transparency, we proposed a new failure model to improve modularity and transparency. Failures are reported to a fault stream created per entity. This allows to monitor failures concurrently, avoiding unexpected exceptions in code extended to a distributed execution. This also provides granularity that is not presented in component based systems.

As future work, we will design guidelines to use the more convenient annotations according to each scenario, having first in mind peer-to-peer applications. We also need

to define proper actions in failure handling to keep consistency of the system. Implementation also need to be finished.

6 Acknowledgements

This research is partly supported by the projects CoreGRID (contract number: 004265) and EVERGROW (contract number:001935), funded by the European Commission in the 6^{th} Framework programme, and project MILOS, funded by the Walloon Region of Belgium, Convention 114856.

References

1. Waldo, J., Wyant, G., Wollrath, A., Kendall, S.: A note on distributed computing. In: Mobile Object Systems: Towards the Programmable Internet. Springer-Verlag: Heidelberg, Germany (1997) 49–64
2. Klintskog, E.: Generic Distribution Support for Programming Systems. PhD thesis, KTH Information and Communication Technology, Sweden (2005)
3. Consortium, M.: The mozart-oz programming system. *http://www.mozart-oz.org* (2006)
4. Van Roy, P., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. MIT Press (2004)
5. Grolaux, D., Glynn, K., Van Roy, P.: A fault tolerant abstraction for transparent distributed programming. [9] 149–160
6. Mesaros, V., Carton, B., Van Roy, P.: P2ps: Peer-to-peer development platform for mozart. [9] 125–136
7. Mejías, B., Collet, R., Van Roy, P.: It's such a fine line between transparent and non-transparent distribution. In: The 17th Nordic Workshop on Programming Theory. (2005) 94–96
8. Van Roy, P.: On the separation of concerns in distributed programming: Application to distribution structure and fault tolerance in mozart (1999)
9. Van Roy, P., ed.: Multiparadigm Programming in Mozart/Oz, Second International Conference, MOZ 2004, Charleroi, Belgium, October 7-8, 2004, Revised Selected and Invited Papers. In Van Roy, P., ed.: MOZ. Volume 3389 of Lecture Notes in Computer Science., Springer (2005)