Communication Abstractions Through New Language Concepts

Jessie Dedecker[*]and Wolfgang De Meuter (jededeck—wdmeuter@vub.ac.be)
Tel: +32-2-629.35.30 - Fax: +32-2-629.35.25
Vrije Universiteit Brussel
PROG - Department of Informatics
Pleinlaan 2
1050 Brussels
Belgium

1 Introduction

In this paper we take the position that dedicated language concepts are to be considered as the solution for introducing commonly used communication abstractions into distributed programs. In our research we explicitly abandon middleware solutions, such as generation of stubs and skeletons. They do not give rise to the new ways of thinking that will be required for the construction of distributed and mobile systems in highly dynamic environments such as interconnected desktops, pda's and domotics. More specifically, we think that both the complexity and weakness of most middleware technology and the 'solutions' the spawn is due to the fact that the technology is statically typed and class-based. Indeed, the major raison d'etre for generated interfaces and stubs is to satisfy type checkers for static languages. Because of that, we are starting to investigate how we can put the properties of prototype-based languages to structure and simplify the development of mobile agent software and thus also distributed systems.

In this paper we introduce some preliminary resulting communication abstractions based on the delegation mechanism most prototype-based languages feature. As a concrete case we will discuss some coordination problems in the master-slave design pattern and do a few gedankenexperiments in language design in this context.

2 Master-Slave Pattern

The master-slave pattern [AL98] is used to partition a task into a set of subtasks to increase the reliability, performance, security and accuracy of the execution. In the context of mobile agents we have a master agent that spawns some slave agents that each perform a subtask and migrate to other computational environments in order to perform that task. After completion

[*]Research Assistant of the Fund for Scientific Research Flanders, Belgium (F.W.O.)

the slaves communicate the result back to the master. Important here is that there is often a need for coordination between the master and its slaves. For example, suppose that we have a task to search for the cheapest hotelroom available in a hotel. We could spawn several slaves that each visit a subset of hotels searching for a hotelroom. When a slave finds a suitable hotelroom it sends the hotel and its price back to the master that maintains the cheapest hotel. Agents will have to migrate often between the computational environment of the master and the computational environment that hosts the hotels when a lot of rooms are available. To optimize this the master should be able to communicate the best hotel that has been found so far to its slaves each time a cheaper room has been found. This example shows that some coordination mechanism is necessary between the master and its slaves when there are interdependencies between the subtasks each slave is responsible for.

3 Language Concepts to the Rescue

In this section we introduce some new language concepts that help to dissolve the master-slave pattern and the coordination communication mechanisms that are often needed between the master and slave agents.

3.1 Prototype Based Languages

Prototype-based objects [Lie86] are self-dependent, they carry their own behavior and do not not depend on a class. Most prototype-based languages make use of a sharing mechanism that is called delegation or object-inheritance. For example, if object A inherits from object B (notice we are not talking about classes) and a message msg is send to object A that is not implemented in that object, then that message is *delegated* to object B. If object B has an implementation for msg then it executes the method associated with message msg in the context of object A. Said in another way, the SELF is referring to object A. Such a mechanism supports the sharing of both behavior and data. In prototype-based languages new objects are created ex-nihilo or by cloning another object, rather than instantiating the object from a class.

3.2 Distributed Object Inheritance

Object inheritance allows a parent to share both its data and behavior with its children. We propose to use object inheritance to structure mobile agents in an application in such a way that they are still able to exchange messages to each other in a convenient way to enhance the coordination between distributed objects and mobile agents. In our language we use mixins to introduce object-inheritance [1]. When a mixin method is executed on an object A (mixins are activated by sending a message to the object), then a new object is returned with the set of method and data slots defined in the mixin and the parent link of the new object is pointing to the object A. Below is the code of a simple counter object with two regular methods incr and decr and one mixin method protect. The mixin method overrides both regular methods to prevent overflow and underflow.

```
counter(n):
```

[1]Mixin-based inheritance has several advantages over classical object-based inheritance (with the most important one the reduction of the encapsulation problem) as pointed out in [SDM95]

```
{ incr():: n:= n+1;
  decr():: n:=n-1;
  protect(limit)::
  { incr()::
      if(n=limit,
         error("overflow"),
         super.incr());
    decr()::
      if(n=-limit,
         error("underflow"),
         super.decr());
    capture()
  }
  capture()
}
```

Mixins reduce the encapsulation problems, because it is the parent object that defines the mixin method and thus decides what child objects are created. We propose to extend mixins to the context of mobile agents. When a *netmixin* is executed on an agent A then a new agent B is spawned (new object and execution thread) with its parent pointing to agent A. When agent B migrates to another computational environment its parent link keeps pointing to that of agent A, but agent A does not necessarily migrate with agent B. Below is the code of an agent that roams over the network to find suitable hotels and communicates the result back to the master agent. The *hotelSearchAgent* (the master) has one netmixin method that spawns a new agent that travels to a set of hotels and checks if they meet the criteria. If they do then the price is sent together with the name of the hotel to the master agent.

```
hotelSearchAgent():
{ bestHotel::void;
  bestPrice::void;

  newHotel(aHotelName, aPrice)::
   if(aPrice < bestPrice,
      {
        bestHotelName:=aHotelName;
        bestPrice:=aPrice
      });

  spawnSlave(hotels,query)::netMixin(
  {
   index::1;
   run(aHotel)::
   {
    migrate(aHotel);
    if(aHotel.meetsCriteria(query),
       super.newHotel(aHotel.name,
  aHotel.price));
    if(index <= size(hotels)),
   index:=index + 1,
   stop());
    run(hotels[index])
   }
   run(hotels[index])
  }
  clone()
})
```

The code below shows an example use of the hotelSearchAgent object. The organisational hierarchy of the objects created by running this code is shown in figure 1.

```
a: hotelSearchAgent();
b: a.spawnSlave([hotel1], myCriteria);
c: a.spawnSlave([hotel2], myCriteria);
d: a.spawnSlave([hotel3], myCriteria);
```

One form of communication between the master and slave agent is through the use of the super and self keyword. For example, a slave agent can communicate with the master agent

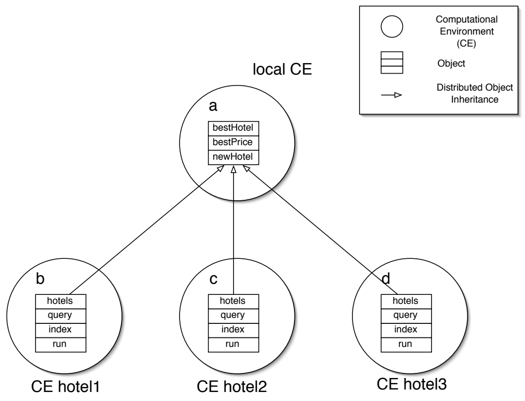

Figure 1: Physical Location of Objects

by sending it a message using the super keyword. The master agent can then send a result back to the slave as a return value to the super request - or - in the case of more complex interactions the master agent can continue to communicate with the slave by sending it messages through the use of the self keyword. Indeed, when the slave sends a message to the master through the use of the super keyword then the master can send a reply to the slave that send that message using the self keyword, because the SELF is referring to that slave. This example shows us how the communication between the master and a slave melts away in the structure of the program code. Without such a mechanism both the master and slave are responsible for maintaining references to each other. These references have to be passed with each communication or stored in a variable making the code more complex. If we look how we can implement the optimisations we described in section 2 then we come to the conclusion that it is impossible using the super and self keywords, because we cannot reference the other slaves.

3.3 Communication Patterns

3.3.1 One-to-many Communication

Sometimes we want to send a message from the master agent to all the other slaves. The keyword self and super do not allow us to send such messages, this is why we introduce the new keyword we. If we send a message to we, then we are sending a message to all the children. The return value of a method call to we always returns void. To make the optimisation such that the master notifies the best hotel that was found so far to its slaves we can conveniently change the code to the following:

```
  if(aPrice < bestPrice,
     {
       bestHotelName:=aHotelName;
       bestPrice:=aPrice;
       we.bestSoFar(bestPrice)
     });
```

We further need a local variable localBestPrice and code in the slave mixin to check if the result that the slave found is better than the best local result. Notice that it is important to keep a local variable in the slave, because otherwise we have to communicate with the master each time we need to check the new price that we found. We can modify the slave mixin to the following:

```
spawnSlave(hotels,query)::netMixin(
{
 index::1;
 localBestPrice::void
 bestSoFar(aPrice)::
 {
   localBestPrice:=aPrice;
 }
 run(aHotel)::
 {
    migrate(aHotel);
    if(and(aHotel.meetsCriteria(query)
           aHotel.price <
           localBestPrice),
       super.newHotel(aHotel.name,
  aHotel.price));
.
.
.
```

Note that we could also implement similar behavior with the observer pattern [GHJV94], but this would require us to implement the pattern manually in the master and slave. Another disadvantage of the observer pattern is that the developer needs to discover the pattern in the code, there are no explicit keywords for it. Sending more refined notify messages to the slaves comes for free with the we keyword, while in the observer pattern has to be adapted for this to a system that is similar to the java listener framework.

3.3.2 Master Agent Synchronization

Another problem we encounter when implementing the example from above is that we need to synchronize the master and slave agents. For example, if we want to implement a method in the master agent that returns the result of the search then we need to wait until all slave agents have finished before returning a result. For this reason we introduce a delayed method. A delayed method only executes when all the children of an object have sent a message to the object implementing the delayed method associated with that message. When a message is send to a delayed method it is asynchronous. This means that the slave that sends the message does not block until the delayed method has been executed. Since messages send to a delayed method are asynchronous, no return value is send back to the slaves. The synchronization is added by adding the following code to the master agent.

```
ready::false;
stop()::delayedMethod(ready:=true);

getResult()::{
  waitFor(stop,ready=true);
  bestHotel.name
}
```

We further need to research what happens with the actual parameters in the case of delayed methods. They could be provided as an array with all the actual parameters, when all children have send a message. Another, perhaps more interesting option is to use them as a means for matching certain results. The delayed message would then only execute when the parameters send match each other (in the sense of prolog-unification). We have to further explore these and other possibilities.

3.3.3 Slave Agent Synchronization

Sometimes we want the slaves to wait for the master until some value has been send by all other slaves. Such a behavior is captured using join methods. Join methods can be seen as the synchronous variation of the delayed methods introduced above. The semantics are the same, but the sending slave agent becomes blocked until all other slaves have send the method. Once the join method has executed the return value is send to all slaves.

Suppose we want the slave agent that found the cheapest hotel to book the room. We can only determine that slave agent after all slaves have been send their results to the master agent.

```
cheapestAgentName::void;
book()::
  joinMethod(cheapestAgentName);
.
.
.
  if(index <= size(hotels)),
    index:=index + 1,
    if(book()=this.name,
      makeBooking()));
```

When all slave agents have send their results to the master agent they send a book message to the master agent. The master agent returns the slave agent that found the cheapest hotel to all slave agents, so that the one can make the booking.

4 Conclusion and Future Work

In this paper we introduced some preliminary communication abstractions that were based on the delegation mechanism most prototype-based languages feature.

The examples shown in section 3 strong our beliefs that new language concepts are a good way for introducing new communication abstractions in languages. Furthermore, these language concepts aid in imposing a new way of thinking about the structure of distributed and mobile programs.

We are currently integrating the new language concepts explained above into the prototype-based programming language Pic% [MDD03] and will do some experiments to gain more practical experience with the new language constructs. We will also investigate the interactions with metaprogramming, concurrency and partial failures. Finally, we will search for other patterns that can expressed with new language concepts using prototype-based language features.

5 Acknowledgements

Thanks to Thomas Cleenewerck for discussing some of the ideas and Dr. Tom Tourwe for proof-reading.

References

[AL98] Yariv Aridor and Danny B. Lange. Agent design patterns: elements of agent application design. In *Proceedings of the second international conference on Autonomous agents*, pages 108–115. ACM Press, 1998.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1994.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 214–223. ACM Press, 1986.

[MDD03] Wolfgang De Meuter, Theo D'Hondt, and Jessie Dedecker. Intersecting classes and prototypes, 2003. Accepted at Andrei Ershov Fifth International Conference "Perspectives of System Informatics".

[SDM95] Patrick Steyaert and Wolfgang De Meuter. A Marriage of Class- and Object-Based Inheritance Without Unwanted Children. In Walter Olthoff, editor, *ECOOP '95 - Object-Oriented Programming*, number 952 in Lecture Notes in Computer Science, pages 127–144. Springer-Verlag, 1995. Proceedings of the 9^{th} European Conference on Object-Oriented Programming. Aarhus, Denmark, August 1995.