Facilitating agent development in open distributed systems

Mauro Gaspari[1] and Davide Guidi[2]

[1] Dipartimento di Scienze dell'Informazione, University of Bologna, Italy
[2] Knowledge Media Institute, The Open University, United Kingdom

Abstract. One of the main reasons about the success of the Web is that many "regular users" are able to create Web pages that, using hyperlinks, incrementally extend both the size and the complexity of the Web itself. The development of agents in the Web infrastructure should ideally be driven by the same paradigm: users writing simple or advanced agents. These agents will then provide capabilities using a set of resources, such as standard Web pages, Web services and, of course, other agents. At the moment, however, agents providing advanced services will never be developed as Web pages have been created in the past. In fact programming agents is a complex task which needs adequate skills and tools to be carried out successfully. As a consequence, only few people are currently able to contribute to their development. Anyway, will it be possible to reduce this gap in the future? In this paper we answer this question presenting NOWHERE, an open agent communication infrastructure which facilitates the programming task in open distributed multi-agent systems.

1 Introduction

Agent platforms usually provide a programming environment and common services to applications developed as agents. These environments can include high-level programming tools for the development of intelligent agents capable of reasoning, planning, and acting in a changing environment, together with communication mechanisms supporting agent interaction. This paper focuses just on communication facilities, which have a fundamental role to increase the power of agents in open distributed Multi-Agent Systems (MAS). Agent platforms embed specific tools to support inter-agent communication. Many of them are based on the speech act theory, which is also the approach followed by the current standard, the FIPA ACL [6]. Jade [1], for example, is one of the most used agent platforms both in academia and in industry, and uses FIPA ACL to provide communication facilities. While FIPA ACL includes human like high-level primitives, it does not have specific features for geographically distributed MAS where agents may crash or simply become unreachable for a while. In fact, if we aim to develop robust implementations of agents in these systems, we have to consider agent failures, and a number of extra speech act primitives should be added to the agent code. Additionally, several low-level issues should be considered, such

as detecting failures, establishing correct timeouts, establishing correct actions to handle failures and so on. As a result of having to deal with these issues the programming task is more difficult and the high-level programming style of the speech act based approach is partially lost.

In this paper, we try to tackle this problem presenting NOWHERE, a modular and open agent communication infrastructure, which has been designed to facilitate the programming task in open, geographically distributed, multi-agent systems. NOWHERE supports a simple programming model which facilitates the development of agents in open distributed systems and works in a reasonable class of application domains. This model supports communication among Knowledge-Level (KL) agents [8], which are agents only concerned with the use, request and supply of knowledge, exploiting an advanced ACL ($\mathbb{FT}$-$\mathbb{ACL}$) [3]) including high-level primitives to deal with failures of agents. Using Knowledge-level agents, the available communication primitives are those of the ACL, and the programmer does not have to explicitly handle many low-level issues, such as network, timeout and concurrency related problems. Thus the programming task becomes simpler.

This paper is organized as follows. In Section 2 we give a sketch of the NOWHERE platform architecture and we present how the $\mathbb{FT}$-$\mathbb{ACL}$ primitives have been realized in a real programming language (Java). Then we compare our approach with a state-of-the-art agent architecture, presenting some details of the FIPA contract-net specification implemented using Jade and NOWHERE. We conclude the paper with a few remarks.

2 The NOWHERE platform

In the NOWHERE platform each agent consists of two main components: a *Dispatcher*, that provides the Knowledge Level layer, and a *Facilitator*, a separate component that deals with low-level aspects, such as sending and retrieving messages providing fault tolerance. These two components communicate together by means of the TCP protocol, using a simple *Connector* interface. While the *Dispatcher* is a relatively simple component that mainly provides ACL primitives to a specific programming language, the *Facilitator* hides the whole complexity of the platform, as shown in Figure 1.

NOWHERE differs from other agent architectures in the way it manages faults. The `Facilitator` component contains a failure handler mechanism that is able to discover crashes of agents. It is based on a set of *transparent* timeouts, that are automatically managed by the architecture.

2.1 NOWHERE's Agent Communication Language

One of the main features of $\mathbb{FT}$-$\mathbb{ACL}$ is the ability to deal with crash failures of agents. Following a well known classification of process failures in distributed systems [10], we say that an agent is *faulty* in an execution if its behaviour deviates from that coded in the algorithm that it is running; otherwise, it is

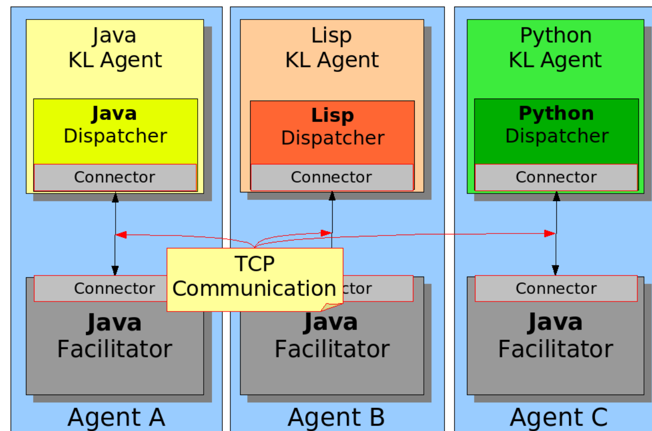

Fig. 1. The NOWHERE architecture

correct. A faulty agent *crashes* if it stops prematurely and does nothing from that point on. 𝔽𝕋-𝔸ℂ𝕃 manages faults considering only crash failures. This is a common fault assumption in distributed systems, since several mechanisms can be used to detect more severe failures and to force a crash in case of detection. 𝔽𝕋-𝔸ℂ𝕃 deals with crash failures of agents allowing the programmer to choose what actions to invoke for each interaction they perform in the MAS, using a continuation based mechanism.

The following Java-like pseudo code describes a sample `genericRequest` primitive, like for example `askOne`, illustrating how 𝔽𝕋-𝔸ℂ𝕃 continuations works.

```
1   public static void main(String args[]) {
2        ... some code ...
3        genericRequest(recipientAgent, content, onAnswer, onFail);
4        ... other code ...
5   }
6   public void onAnswer(Message replyMessage){
7        // Here we handle the success continuation
8        // of the genericRequest primitive
9   }
10  public void onFail(){
11       // Here we handle the failure continuation
12       // of the genericRequest primitive
13  }
```

In the code presented above there is a main function (`mainCode`, lines 1-5) that at some point sends a message to the agent `recipientAgent` using the performative `genericRequest` (line 3). A typical request primitive is usually realised using only two arguments: the recipient (`recipientAgent`) and the content of the request that must be sent (`content`). Instead, using the 𝔽𝕋-𝔸ℂ𝕃

style, a communication primitive also includes a success and a failure continuation, `onAnswer` and `onFail` respectively. These parameters are functions that allow the programmer to specify the success and the failure continuation associated to `genericRequest`. Due to the fact that the language uses non-blocking primitives, after the execution of `genericRequest`, the control flow immediately passes to the next instructions, contained in the "`... other code ...`" block, line 4. When the reply message is received, the success continuation `onAnswer` (lines 6-8) is executed, with the parameter `replyMessage` instantiated with the reply. Otherwise if a communication error arises, then the failure continuation `onFail` (lines 10-13) will be executed. Note that the behaviour of the success continuation `onAnswer` is specific for a request performative. If we consider other types of performative the role of the success continuation can be different. For example the success continuation can be activated when a message is received by the recipient agent to acknowledge that an inform performative is successfully executed. The interaction patterns supported by $\mathbb{FT}$-$\mathbb{ACL}$ for different classes of performatives are described in details in [4].

Agents written using $\mathbb{FT}$-$\mathbb{ACL}$ are also easy to program because these Knowledge Level properties hold ([3]):

(1) The programmer does not have to manage physical addresses of agents explicitly.
(2) The programmer does not have to handle communication faults explicitly.
(3) Communication is Starvation free.
(4) Communication is Deadlock free.

Altough NOWHERE communication primitives are deadlock free, it is not guaranteed that applications implemented in NOWHERE are deadlock free in general. For example, if an agent implements a shared resource using a wrong allocation policy, then a resource deadlock may occur.

2.2 Language Primitives

Language primitives support communication providing agents with the capability to exchange messages and invoke service. A `Message` object encapsulates the content of the communication in a language-independent way, so that agents written in different languages are able to exchange messages, for example with the `inform` primitive. Simple or complex capabilities can be shared among agents using a `Service` object. These services are described using a subset of WSDL [2], the standard XML format for Web services. Services differ from messages because they have a description that holds information about several aspects, including the name of the service, its parameters and the data types used. They are used with specific communication primitives, for instance `askOne`, `askEverybody` and `tell`. In the NOWHERE architecture, a service description is contained in a `Description` object. To invoke a service and to send a reply NOWHERE provides a `Request` and a `Response` object, that can be retrieved from the description object. Both the `Request` and the `Response` objects are templates

containing relevant information extracted from the service description, such as the name of the parameters of the service. In order to invoke a service (to provide a response), a `Request` (a `Response`) template must first be filled in with the correct information. Due to the fact that these templates contain part of the service description, they simplify the actions of invoking and replying to a service.

The communication primitives provided by NOWHERE are shown in Table 1. For space constraints we only present the details of those used in the subsequent case study. The interested reader will find a detailed description of all the implemented performative in [9].

One-to-one knowledge exchange
`inform(recipientAgent, message)`
`informACK(recipientAgent, message, onAnswer[, onFail])`
Using functions to manage specific messages
`handler(message, function)`
Managing Services
`Description loadDescription(WSDL_Description)`
`Description makeDescription(targetNS, operation, parameters, returnParameters)`
Using functions to manage specific services
`handler(request, function)`
Providing and Requesting services
`askOne(recipientAgent, request, onAnswer[, onFail])`
`tell(recipientAgent, response)`
Service publishing
`register(description)`
Anonymous service request
`askEverybody(request, onAnswer[, onFail])`
`allAnswers()`

Table 1. Language Primitives

One-to-one knowledge exchange.
Communication between two agents can be achieved using the `inform` primitive, the very basic communication method provided by NOWHERE. The syntax of this primitive is:

```
inform(recipientAgent, message)
```

where `recipientAgent` is the unique ID (identifier) of the recipient agent and `message` represents the message containing the information to be sent. The `inform` primitive is used to send a message to another agent, without any feedback about the delivery status. No actions are performed by the sender agent

if the recipient receives the message, as well as no actions are performed if the message is not delivered for some reason.

Request/Response Performatives.
In order to use services, a `Description` object (that stores the data about the service) must first be generated from a standard WSDL file. The `loadDescription` primitive is provided for this purpose. It parses a WSDL file either from a local resource or from the Web, returning a NOWHERE `Description` object. The `askOne` primitive must be used to invoke a service provided by another agent. The syntax is:

```
askOne(recipientAgent, Request, onAnswer[, onFail])
```

The `recipientAgent` parameter represents the target agent, while the invocation of the service, together with the parameters associated to the specific service, is contained in the `Request` object. The `onAnswer` and the `onFail` parameters are the associated continuations. They represent the names of the functions that will handle the answer and the failure, respectively. The NOWHERE architecture automatically invokes one of these two functions, depending on the result of the invocation.

Anonymous interaction mechanism.
NOWHERE provides support for invoking a service from a set of agents. This mechanism is also known as content-based request, because a service is invoked specifying its content rather than the name of the agent that provides it. Services can be invoked using the `askEverybody` primitive, whose syntax is:

```
askEverybody(Request, onAnswer, onFail)
```

The parameters are the same as in the `askOne` primitive seen before, except that in this case the recipient agent is not specified. It is the runtime support that will send the request to all (and only) the agents that provide the specified service. Every time that a reply is received, the `onAnswer` function will be called. Instead, the `onFail` function will be called only if no agents replied at all. Inside the `onAnswer` function it is possible to check if the current reply is the last one using the `allAnswers` predicate. The `allAnswers` is a boolean predicate that returns `true` if the current response is the last reply for the associated `askEverybody`, `false` otherwise.

3 Transparent timeouts

Timeouts are used to provide a framework that can be adaptable to different situations. The timeouts used in NOWHERE are called "transparent timeouts" because they are managed by the architecture itself, so that the user does not have to deal with them. In NOWHERE, timeouts are countdown timers that

are activated when a certain primitive is issued or, in some cases, received. Each timeout is associated to a custom message containing an action to do when the countdown timer reaches zero. Usually the action is to execute the failure continuation for the associated primitive.

Every timeout object contains:

- A message, which encodes the action to be taken when the countdown timer reaches zero.
- Two extra parameters: the `agentType` and `agentReactiveness`.

The message associated with every countdown timer is automatically sent using the Facilitator when the countdown reaches zero.

The value for the countdown is calculated using the properties `agentType` and `agentReactiveness` which are associated to each agent. The `agentType` property can be considered an upper bound of the time that the agent will wait. It defines the maximum time that an agent will wait for external replies. If no replies are given during this time, then the failure continuation is fired. The `agentReactiveness` is instead the minimum time that an agent will wait for an answer.

Every communication primitive can be associated to a custom couple of `agentType` and `agentReactiveness` properties. The `agentReactiveness` property affects how the interaction with the recipient agent will be managed by the Facilitator. A low value will force the Facilitator to check the recipient agent very frequently, in order to promptly find crashes. On the other hand, using high values the Facilitator will accept network lags or temporary failures of the recipient agent. For the implementation of the `askOne` communication primitive, these properties are managed using the following algorithm:

1 - The Agent executes the `askOne` primitive.
2 - The associated Facilitator sends the message containing the primitive.
3 - The Facilitator starts a countdown timer set to the lower value between `agentReactiveness` and `agentType`.
4 - When the Facilitator receives the reply before that the countdown reaches zero, it will halt the countdown and forward the received message to the dispatcher (**the success continuation fires**).
5 - When the Facilitator receives a `NeedMoreTime` message before that the countdown reaches zero, the `agentType` value will be decremented by the actual number of milliseconds already passed since the countdown started. The algorithm **continues to step 3**.
6 - When the countdown reaches zero, the message associated to the countdown timer will be forwarded to the Dispatcher (**the failure continuation fires**).

The algorithm has a loop (lines 3-5) which will end with the success or failure continuation, in lines 4 and 6. The `NeedMoreTime` message is automatically generated and managed by the Facilitator. Timeouts are contained in the `CountdownRepository`, a structure that provide two basic mechanisms: `stop`, to halt a specific timer, and `restart`, to restart it.

In order to explain this algorithm we introduce a simple scenario, in which AgentA executes an `askOne` primitive in order to invoke a service from AgentB. Four different cases can be obtained:

1. AgentB replies in due time: the time waited by AgentA for the reply is less than the maximum allowed time set by AgentA (`agentType`). This case is illustrated in Figure 2, where FA and FB indicate the Facilitator of AgentA and the Facilitator of AgentB respectively.
2. AgentB has already crashed when AgentA invokes the service. This case is illustrated in Figure 3.
3. AgentB receives the request, but it crashes (or a network error occurres) before replying, so that AgentA never receives a proper reply. This case is illustrated in Figure 4.
4. AgentB does not reply in due time, that is AgentA does not receive the reply in the maximum allowed time (specified by `agentType`). This case is considered in Figure 5.

Step	Time	AgentA AgentType: 4000 msec Reactiveness: 500msec		AgentB
1	$T_0 = 0$	AgentA asks AgentB for service S. (FA starts its timer: 500 ms up to 4000 ms)	→	AgentB begin to compute the service. (FB starts its timer: <500ms, up to <4000 ms))
2	$T_1 < 500$ ms	(FA restarts its timer)	←	(FB sends a NeedMoreTime message and restarts its timer)
3	$T_2 < T_1 + 500$ ms	(FA restarts its timer)	←	(FB sends a NeedMoreTime message and restarts its timer)
4	$T_3 < T_2 + 500$ ms	AgentA executes the success continuation (FA stops its timer)	←	AgentB sends the reply (FB stop its timer)

Fig. 2. Success Invocation of a Service

The `agentType` parameter associates an agent to a specific class of agents with similar interactive characteristics. In principle, any numeric value can be associated to this parameter using the `setAgentType` primitive. However, NOWHERE suggests a predefined set of default values:

– *Real Time Agent*, for agents that need a reply in 2 seconds.
– *Web Agent*, for agents that need a reply in 4 seconds.
– *Worker Agent*, for agents that need a reply in 1 minute.
– *Truster Agent*, for agents that can wait indefinitely for a reply. This is needed for example when the sender agent wants to dispatch a task and it does not know a priori how much time the task will take. Of course, if the recipient crashes before receiving reply, then the Facilitator of the sender agent will fire a failure continuation.

Step	Time	AgentA AgentType = 4000 msec Reactiveness: 500msec		AgentB
1	$T_0 = 0$	AgentA asks AgentB for service S. (FA starts its timer: 500 ms up to 4000 ms)	→	AgentB begin to compute the service. (FB starts its timer: <500ms, up to <4000 ms))
2	$T_2 = 500$ ms	AgentA executes the failure continuation (FA stops its timer)		Crash or Network error occurred (If FB not crashed then FB stops its timer)

Fig. 3. Failure Invocation of a Service (AgentB is already Crashed)

Step	Time	AgentA AgentType = 4000 msec Reactiveness: 500msec		AgentB
1	$T_0 = 0$	AgentA asks AgentB for service S. (FA starts its timer: 500 ms up to 4000 ms)	→	AgentB is crashed or not reacheable
2	$T_2 = 500$ ms	AgentA executes the failure continuation (FA stops its timer)		

Fig. 4. Failure Invocation of a Service (AgentB Crashes before Replying)

These values were defined according to the work made by Nielsen in [11], one of the standard reference for the Web usability.

4 Case study: The FIPA Contract Net Protocol

The purpose of this case study is to compare the solution obtained using the NOWHERE approach to the solution provided by Jade, a state-of-the-art agent platform. We choose a slightly modified version of the classic Contract Net[13], fully described in the FIPA specification[7]. The Contract Net protocol allows an agent to distribute tasks among a set of agents by means of negotiation. The modified version considers only a single manager agent, the `Initiator`, and a set of worker agents, the `Responders`. Moreover, the FIPA Contract Net also includes rejection and confirmation communicative acts.

In the following we just recall the basic principles of the protocol, described in detail in the FIPA specification. A representation of this protocol is given in Figure 6 which is based on extensions to UML1.x[12]. The sequence diagram describes the inter-agent transactions needed to implement the protocol, where the diamond symbol indicates a decision that can result in zero or more communications being sent, depending on the conditions it contains.

Step	Time	AgentA AgentType = 4000 msec Reactiveness: 500msec		AgentB
1	$T_0 = 0$	AgentA asks AgentB for service S. (FA starts its timer: 500 ms up to 4000 ms)	→	AgentB begin to compute the service. (FB starts its timer: <500ms, up to <4000 ms))
2	$T_1 < 500$ ms	(FA restarts its timer)	←	(FB sends a NeedMoreTime message and restarts its timer)
3	$T_2 < T_1 + 500$ ms	(FA restarts its timer)	←	(FB sends a NeedMoreTime message and restarts its timer)
4	$T_3 < T_2 + 500$ ms	(FA restarts its timer)	←	(FB sends a NeedMoreTime message and restarts its timer)
...	...	...	...	...
n	T_n=4000 ms	AgentA executes the failure continuation (FA stops its timer)		(FB stop its timer)

Fig. 5. AgentB does not Reply in Due Time

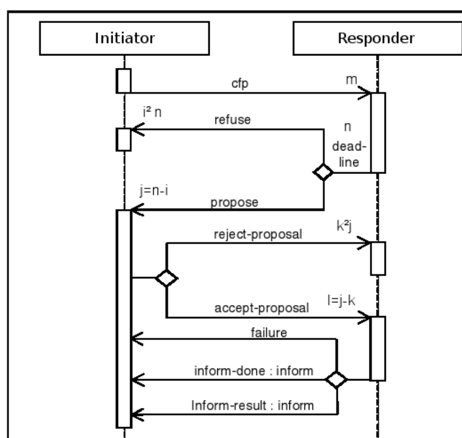

Fig. 6. FIPA Contract Net Protocol (source: FIPA Specification)

According to the FIPA specification, the Initiator agent sends a call for proposal (cfp) act, soliciting a proposal from every other m agents, specifying the task to be done. Responders receiving the call for proposals are viewed as potential contractors and are able to generate n responses. Of these, j are proposals to perform the task, specified as propose acts. The Responder's proposal includes the preconditions that the Responder is setting out for the task, which may be the price, time when the task will be done, etc. Alternatively, the i=n-j Responders may refuse to propose. Once the deadline passes, the Initiator evaluates the received j proposals and selects agents to perform the task; one, several or no agents may be chosen.

Being a FIPA compliant platform, Jade adheres as much as possible to FIPA specifications. For this reason Jade implements ad-hoc mechanisms for the FIPA Contract Net, providing facilities that simplify the programming task. In fact, the task of the programmer is just to extend the two Java classes provided for the Initiator and for the Responder role. In order to handle proposals from Responder agents, for example, the developer must only write the proper code inside a function named `handlePropose`. The Jade architecture will then invoke this function properly, for each received proposal. In the Jade platform these ad-hoc mechanisms are called *behaviours*, and are used to easily implement well defined actions, like doing repetitive tasks (using the CyclicBehaviour), simultaneously executing different tasks (using the ParallelBehaviour) or, as in this example, starting a FIPA Contract Net interaction protocol.

For the comparison we proceed in this way: first we introduce the algorithm used in the Jade platform (adapted from an example found in the Jade software distribution) and then we provide an equivalent solution for the NOWHERE architecture. For space limitations, we only analyze the Initiator agent. However, the Responder agent is based on a straightforward reactive algorithm.

4.1 The Initiator agent - Jade

The algorithm implemented by the Initiator agent is composed of 3 main steps:

1. Find the set of available Responder agents;
2. Send a cfp message to Responder agents;
3. Select and accept the best proposal;

1 & 2 - Find the set of available agents and send a cfp message to them.
The source code for the first two steps is presented in Figure 7. In the Jade platform the task of finding other agents is delegated to the Directory Facilitator component. In order to find other agents, the Initiator should first fill in a Service Description object (lines 1-2). The Service Description object contains information about the resource that we want to find. In this case we used a *type* tag to identify Responder agents. (line 2). The next block of code, lines 3-10, performs a query on the Directory Facilitator and retrieves a list of the available Responder agents. The second step is to send a cfp message to every Responder agent found in the previous step. In lines 11-18 a proper cfp message is created, specifying every collected agent as receiver, if there are any (line 11). Additionally, the agent sets a maximum timeout of 10 seconds for the proposals (line 17) and the name of the task to be dispatched (line 18). The newly created message is then automatically sent using the `ContractNetInitiator` behaviour in the third step.

3 - Select and accept the best proposal.
The code used for the third step is shown in Figure 8. Again, replies from Responder agents are managed exploiting Jade's FIPA Contract Net behaviour. The messages are handled using the `handleAllResponses` function (lines 20-44). This function is automatically called by the Jade infrastructure when all

```
   // Step 1: Find the set of available Responder agents
1  ServiceDescription sd = new ServiceDescription();
2  sd.setType("Responder");
3  DFAgentDescription df = new DFAgentDescription();
4  df.addServices(sd);
5  DFAgentDescription[] agentList = null;
6  try {
7     agentList = DFService.search(this, df);
8  } catch (Exception e) {
9     e.printStackTrace();
10 }
11 if (agentList != null && agentList.length > 0) {
   // Step 2: Send a cfp message to Responder agents
12    ACLMessage msg = new ACLMessage(ACLMessage.CFP);
13    for (int i = 0; i < agentList.length; ++i) {
14       msg.addReceiver(((DFAgentDescription)agentList[i]).getName());
15    }
16    msg.setProtocol(FIPANames.InteractionProtocol.FIPA_CONTRACT_NET);
17    msg.setReplyByDate(new Date(System.currentTimeMillis() + 10000));
18    msg.setContent("dummy-action");
```

Fig. 7. Initiator Agent - Jade solution - first fraction

the replies have been received. The code in lines 21-38 selects the best proposal, sending a REJECT message to the less competitive replies. The proposal are simply evaluated comparing them against the `bestProposal` variable that stores in every iteration the best proposal received. Replies to the Responder agents are stored in the `acceptances` Java Vector, and are then automatically sent. Finally, the code in lines 39-42 accepts the best proposal received.

4.2 The Initiator Agent - NOWHERE

The solution developed for the NOWHERE platform is shown in Figure 9. Thanks to the anonymous interaction mechanism, there is no need to search for Responder agents. The cfp message can be sent directly to all the Responder agents, that are automatically discovered. In this case the anonymous interaction mechanism relies on an agent capability, that is used to find Responder agents. This capability is provided by all the agents that want to act as Responder agents, and it is described by an external WSDL file, which can be something similar to the one presented in Figure 10. This WSDL description is then loaded in the architecture using the `loadDescription` primitive (line 2), which returns a `Description` object from a WSDL file. The code in line 4 sets the timeout to 10 seconds, accordingly to the Jade's version. A `Request` object is then instantiated with proper values and sent to Responder agents using an `askEverybody` primitive (lines 5-7). The `handlePropose` function (lines 9-20)

```
   // Step 3: Managing replies from Responder agents
19    addBehaviour(new ContractNetInitiator(this, msg) {
20       protected void handleAllResponses \
         (Vector responses, Vector acceptances) {
21          int bestProposal = -1;
22          AID bestProposer = null;
23          ACLMessage accept = null;
24          Enumeration e = responses.elements();
25          while (e.hasMoreElements()) {
26             ACLMessage msg = (ACLMessage) e.nextElement();
27             if (msg.getPerformative() == ACLMessage.PROPOSE) {
28                ACLMessage reply = msg.createReply();
29                reply.setPerformative(ACLMessage.REJECT_PROPOSAL);
30                acceptances.addElement(reply);
31                int proposal = Integer.parseInt(msg.getContent());
32                if (proposal > bestProposal) {
33                   bestProposal = proposal;
34                   bestProposer = msg.getSender();
35                   accept = reply;
36                }
37             }
38          }
   // Step 4: Evaluate the proposals and accept the best offer
39          if (accept != null) {
40             accept.setPerformative(ACLMessage.ACCEPT_PROPOSAL);
41             acceptances.addElement(accept)
42          }
43       } // This closes the addBehaviour function (line 19)
44    }); // This closes the if branch of line 11, Fig. 7
45 }
```

Fig. 8. Initiator Agent - Jade solution - second fraction

will then be called every time the Initatior agent will receive a reply. As in the Jade solution, this function will select the best proposal, sending a REJECT message to the less competitive agents (line 17). Being a synchronized method, the `handlePropose` function will avoid concurrency problem when accessing the `bestProposal` variable.

4.3 Discussion

The first thing to observe is that the solution obtained with Jade exploits a set of ad-hoc facilities to manage interactions in the Contract Net protocol. Even using these facilities for Jade the source code of the solution based on NOWHERE is much compact.

Analyzing in details the Jade solution, we can observe that two main features are provided by the Jade's FIPA Contract Net behaviour:

```
1  bestProposal = -1;
2  Description cfp = loadDescription("http://maya.unibo.it/cnp.wsdl");
3  public void startAgent() {
4     this.setAgentType(10000);
5     Request r = cfp.getRequest();
6     r.setParameter("taskName", "dummy-action");
7     askEverybody(r, "handlePropose", null);
8  }
9  public synchronized void handlePropose(Message m) {
10    Response r = cfp.retrieveResponseFromMessage(m);
11    int proposal = (Integer) r.getParameter("proposal");
12    if (proposal > bestProposal) {
13       bestProposal = proposal;
14       bestProposer = m.getSender();
15    }
16    else
17       inform(m.getSender(), new Message("REJECT_PROPOSAL")
18    if (allAnswer() && bestProposer != null)
19       inform(m.getSender(), new Message("ACCEPT_PROPOSAL")
20 }
```

Fig. 9. Initiator Agent - NOWHERE solution

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="contractNetProtocol"
  targetNamespace="http://www.maya.ei.unibo.it/wsdl/cnp.wsdl"
  xmlns:tns="http://www.maya.ei.unibo.it/wsdl/cnp.wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <message name="DoTaskRequest">
    <part name="taskName" type="xsd:string"/>
  </message>
  <message name="DoTaskResponse">
    <part name="taskResult" type="xsd:string"/>
  </message>
  <portType name="DoTask">
   <operation name="doTask">
    <input message="tns:DoTaskRequest"/>
    <output message="tns:DoTaskResponse"/>
   </operation>
  </portType>
</definitions>
```

Fig. 10. The Contract Net Protocol WSDL Description

- the facility to automate some tasks, like to automatically reply to Responder agents with rejection or acceptance of proposals storing the answers in a vector (Fig. 8, lines 30 and 42);

– the facility that allows the developer to consider just the correct proposals in the `handleAllResponses` function, so that the developer does not have to deal with faulty agents.

While these features make the programming task easier, Jade provides them only for the Contract Net implementation. On the contrary, the NOWHERE approach provides a general purpose built-in mechanism which can be used in many contexts. The anonymous interaction mechanism, for example, can be used to send a message to every agent in the network that satisfies a set of specific criteria (such as to be a Responder agent). Regarding failures, both solutions add functions to handle low-level communication problems, implementing an appropriate `handleFailure` function. Again, the NOWHERE architecture provides these features as built-in. Thus they can be used for implementing any kind of interaction protocol. The general idea behind NOWHERE is to simplify the agent programming task, allowing the developer to concentrate in writing the code he/she is working on, avoiding as much as possible the need to explicitly write code to handle failures. Moreover, the NOWHERE architecture provides a fault tolerant system that implements a much more sophisticated algorithm than a simple communication timeout. With regard to inter-agent communication, is important to note that NOWHERE agents can be realised in any programming language including AI languages or knowledge representation languages, provided that they react to a well defined protocol based on the standard primitives of the ACL. Further advantages of using $\mathbb{FT}$-$\mathbb{ACL}$ primitives is that they satisfy a set of well defined properties[3]. The resulting communication will then be free from problems like communication deadlock and starvation.

The FIPA contract net protocol does not take into consideration failures of Responder agents receiving the `ACCEPT_PROPOSAL` message. However, using the continuations mechanism, it is easy to add this feature. A transaction mechanism can be realized using the (more lightweight) continuations, for example with following pseudocode:

```
21 public void askProposer(String proposer)
22    askOne(proposer, acceptRequest, contractNetOk, getNextProposal)
23 public void getNextProposal()
24    if (proposal.hasNext())
25       askProposer(proposals.next())
```

The `askProposer` function (line 21-22) is used to send a request for the acceptation of the proposal to a Responder agent, whose name is specified as a parameter. The `contractNetOk` function will be executed if the Responder agent replies correctly. Otherwise, the `getNextProposal` function (lines 23-25) is executed. The effect of the compensation is to restart the acceptation phase, sending a request to the agent author of the second best proposal, and so on.

5 Conclusions

In this paper we have presented NOWHERE, a communication infrastructure that facilitates agent development supporting Knowledge Level agents. Inter-agent communication is performed by means of an advanced Agent Communication Language ($\mathbb{FT}$-$\mathbb{ACL}$) that, like other popular ACLs such as FIPA ACL [6] and KQML [5], is based on the speech acts theory. However, if we consider concurrency related issues the expressive power of these languages is very different. For example, FIPA ACL sends every communication performative as content of asynchronous message passing. On the contrary the $\mathbb{FT}$-$\mathbb{ACL}$ performatives used in NOWHERE can be classified in a few well defined patterns [4], each one with a different concurrent semantics. Every performative consist of a complex behaviour that is fundamentally different from a simple send primitive. The comparison between different solutions to the Contract Net Protocol, provided in this paper, helps to highlight the effects of the adoption of this approach.

References

1. F. Bellifemine, A. Poggi, and G. Rimassa. Jade: a fipa2000 compliant agent development environment. In *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*, pages 216–217, New York, NY, USA, 2001. ACM Press.
2. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1. Available online at: `http://www.w3.org/TR/wsdl`, 2001.
3. N. Dragoni and M. Gaspari. Crash failure detection in asynchronous agent communication languages. *Autonomous Agents and Multi-Agent Systems*, 13(3):355–390, 2006.
4. N. Dragoni and M. Gaspari. Performative patterns for designing verifiable acls. In *Proc. of the Tenth International Workshop on Cooperative Information Agents (CIA)*, volume 4149 of *Lecture Notes in AI*, Berlin, Germany, 2006. Springer Verlag.
5. T. Finin, Y. Labrou, and J. Mayfield. KQML as an Agent Communication Language. In *Software Agents*, pages 291–316. MIT Press, 1997.
6. FIPA Communicative Act Library Specification. Available online: `http://www.fipa.org/`, 2002. Document number: SC00037J.
7. FIPA Contract Net Interaction Protocol Specification. Available online at `http://www.fipa.org/specs/fipa00029/SC00029H.pdf`, 2002.
8. M. Gaspari. Concurrency and Knowledge-Level Communication in Agent Languages. *Artificial Intelligence*, 105(1-2):1–45, 1998.
9. D. Guidi. A communication infrastructure to support knowledge level agents on the web. Technical Report UBLCS-2007-06, Department of Computer Science, University of Bologna, 2007.
10. S. Mullender. *Distributed Systems.* ADDISON-WESLEY, 1993.
11. J. Nielsen. *Usability Engineering.* MA Academic Press, 1993.
12. J. Odell, H. V. D. Parunak, and B. Bauer. Representing agent interaction protocols in UML. In *AOSE*, pages 121–140, 2001.
13. R. G. Smith. The Contract Net Protocol: High Level Communication and Control in a Distributed Problem Solver . *IEEE Transactions on Computers*, 29(12):1104–1113, 1980.