

When Network Management Agents Become Autonomous, How to Ensure Their Reliability?

Morsy M. Cheikhrouhou and Jacques Labetoulle
Institut Eurécom, Corporate Communications Dep.
BP 193, Sophia-Antipolis Cedex - France
email: {Morsy.Cheikhrouhou, Jacques.Labetoulle}@eurecom.fr

Abstract

We propose to provide a prototype of an agent-based network management system in which, unreliable agents can be detected by the other agents. The detection method is based on having the agents testing each other by comparing their respective beliefs.

The agents are specified using an abstract agent functional model. This model is based on a BDI (Belief Desire Intention)-like mental cycle and uses KQML as an agent communication language. The result is a high-level specification expressed in terms of abstract agent mental attitudes.

1. Introduction

Centralized Network Management Systems (NMS) are being abandoned by the NM R&D community. Instead of having a central manager interacting with a large number of unintelligent agents, the trend is evolving towards having distributed autonomous intelligent agents or middle-level managers performing high-level management tasks [3, 7]. This distributed approach solves the main problems of centralized NMSs, namely, the bandwidth bottleneck around the central management console and the processing overload of its CPU. Moreover, deploying autonomous management agents helps providing localized, therefore, prompt reactions to network problems. Many such autonomous agent architectures have been and are being proposed. One kind of these architectures, the BDI-like (Belief, Desire, Intention) architectures are being investigated in the Network Management Team in Institut Eurécom. BDI-like architectures describe an agent's behavior using a set of mental categories evolving in a mental cycle that allows the agent to take decisions and to act on the environment.

Such agents, being endowed with enhanced decision-taking capabilities and managerial knowledge, are supposed to be able to perform long term tasks in an independent

and autonomous way. However, these agents are deployed in the same network they are managing, which is prone to faults and performance degradation problems. Due to such anomalies, the agents may themselves be affected and therefore, become unreliable. The unreliability of an intelligent agent might have critical effects on the managed network, since the agent is supposed to have important management responsibilities and capabilities. Moreover, detecting the unreliability of an agent is not straightforward since it need not interact with the other agents or with the human administrator to perform its duties.

This was not a critical issue in the classical management agents. In the Internet model, SNMP (Simple Network Management Protocol) agents do not perform any action unless explicitly solicited, via polling GET queries, and confirmed Set requests. Therefore, the management station knows the state of the SNMP agent each time it is queried. In the OSI model, the management protocol CMIP (Common Management Information Protocol) is connection-oriented. This provides sufficient guarantee regarding the limited functionality of the agent.

The purpose of the work presented in this paper is twofold. Firstly, it provides a mechanism that distributedly allows to detect the possible unreliability of intelligent agents while performing their management tasks. Secondly, it describes an abstract agent model built using a BDI architecture and how this model is used to specify and develop the mechanism of reliability detection.

The paper is structured as follows. The problem of intelligent agent reliability in the network management context, as well as the adopted solution, are described in Section 2. We describe in Section 3 an abstract agent functional model based on well-known agent paradigms and languages such as the BDI-oriented architecture and the agent communication language KQML. Next, we detail in Section 4 how our abstract agent model was used to provide an abstract specification of the adopted solution. Finally, we conclude the paper with an outlook of possible improvements.

2. Problem Position

Increasingly nowadays, networks are managed in a hierarchical, yet evolving to a distributed manner [3, 9, 7]. The managed network is divided into sub-networks or domains that are managed more or less independently by autonomous agents. The distribution is introduced into management systems mainly to overcome the bandwidth bottleneck around the central management station and to offer a degree of fault tolerance. As a matter of fact, in a multi-domain managed network, when an agent that manages a particular domain becomes unreliable, the manageability of its domain becomes questionable, but the other domains remain correctly managed. However, if an agent is allowed to perform critical management tasks that may for example affect the performances, or even worse, compromise the network security, it is necessary to promptly detect when this agent turns unreliable. In addition, if the agent is responsible for the management of a sensitive server of which all the network domains make use, an erroneous agent action may compromise the overall function or performances of the whole network. Therefore, it is compulsory, even within a distributed NMS, to be able to detect the failures of the management agents.

Once the failure of an agent is detected, it becomes even possible to have a further improvement by re-affecting the management tasks of the unreliable agent among the other agents in a way to ensure that the whole network continues to be reliably managed. This provides a property of graceful degradation to the distributed management system.

The work presented in this paper provides a first step towards this interesting improvement. To ensure that the whole network is still managed even if a number of agents become unreliable, it is necessary to install a mechanism that continuously checks the reliability of the agents. When unreliable agents are detected, the management tasks that they have been performing are re-distributed amongst the other still-reliable agents. At some time in the future, the agent with the abnormal behavior might recover, for example following a human intervention, and the tasks that have been re-distributed on the other agents should be assigned back to the recovered agents.

To test the reliability of an agent, that we call the *testee*, another agent, the *tester*, can be used. The tester can check a subset of the testee's beliefs against its own beliefs. For example, if the testee agent is monitoring some network elements, the tester agent may monitor a subset of the same network elements, and regularly compares its beliefs on them with those of the testee. If the beliefs match, then the tester decides the testee is reliable, otherwise, the testee is considered to be unreliable.

However, the result tells only the belief of the tester on the reliability of the testee. But nothing ensures that

the tester is itself reliable. SLD (System Level Diagnosis) brings a solution to this problem [2, 1]. SLD allows to deduce the reliable set of entities that are mutually testing each other, each entity having to test a minimum number of other entities. Each entity reports whether the entities that it tests are reliable or not. By confronting the results with each other, SLD allows to deduce a core of reliable entities. Therefore, by making the agents in the management system continuously test the reliability of each other, they can conclude which are the agents that become unreliable. They can consequently perform task re-affectation to avoid the unreliable agents. The continuous testing and the application of SLD allows to detect the agents that might recover.

We propose to build agents that are able to perform the reliability testing while performing their usual management tasks. When an agent is asked about the reliability of a testee, it has to periodically compare its beliefs with its own. For prototyping reasons, we chose as a management task for all the agents, the task of monitoring the global statuses of the network elements in each agent's domain.

3. The Agent Model

Our view of the agent technology as applied to Network Management tends to the use of hybrid agents that are capable of both deliberative and reactive behaviors [8, 6]. The deliberative behavior allows the agent to have long term activities and provides it with decision-taking and reasoning capabilities. The reactive behavior allows the agent to have prompt responses and to setup appropriate reflex actions to changes in the managed network.

In addition, KQML [5] is chosen as an agent communication support. KQML (Knowledge Query and Manipulation Language) offers the advantage of having standard semantics of the intention expressed on an exchanged messages. Also, it provides a rich set of message types, allowing to easily express the attitude wanted from the message. Finally, messages in KQML are written according to a simple syntax that can be easily read and understood.

Furthermore, we perceive the agent as composed of two layers, namely the *Deliberative Layer* and the *Operational Layer*. The deliberative layer offers facilities and supports the deliberative behavior of the agent, while the operational layer provides an environment for the execution and control of the agent actions.

According to these three considerations, we defined an abstract agent functional model described in the following sections.

3.1. The Deliberative Layer

We model the agent deliberative behavior using a set of mental categories that evolve within a mental cycle. The

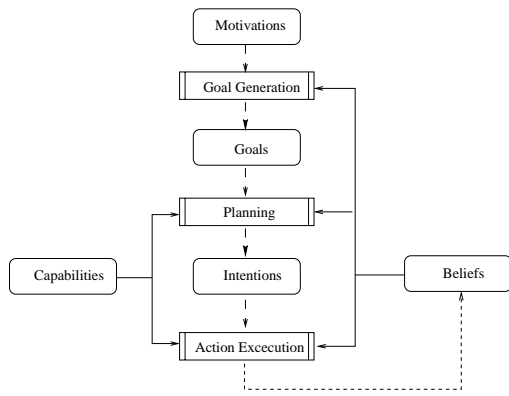


Figure 1. Agent's Mental Cycle

mental categories we use in our model are *motivations*, *goals*, *intentions*, *beliefs* and *capabilities* (Figure 1). They are explained below.

- **Beliefs:** They reflect the agent's perception of the external world. Typically, the agent's beliefs are stored in a database that holds the management information about the network, and possibly include information about the other agents or the agent itself.

We use a simple relational notation of beliefs. Each belief is a relation tuple having a determined number of parameters or fields. For example, the belief "host 'esteron' is down" can be written as follows:

```
(Host :name esteron :status down).
```

Belief querying can be done using logical variables. For example, assuming the above belief is asserted in the agent belief database, then the statement:

```
(Host :name esteron :status ?s)
```

holds with the value of variable *s* bound to 'down'. To avoid ambiguity when using variables, we use *?var* to bind the variable to some field value, and *\$var* to use the value to which variable *var* is bound.

Finally, logical expressions can be combined together using the usual logical operators. For example:

```
(Printer :name ?p) and (OutOfOrder :host $p)
```

holds when variable *p* is bound to the name of a printer that is out of order.

- **Capabilities:** They describe the actions that the agent can perform, mainly to interact with the external world. In some way, the execution of an agent action could be seen as an instantiation of a certain capability. The actions that the agent can perform can be either primitive actions, or composite actions organized in *plans*. There are four types of primitive actions.

1. **Sensors.** They are actions that the agent performs to perceive the external world. Part of the agent's beliefs are provided and maintained through the activation of its sensors. Therefore, a sensor provides a mapping from the changes and events that occur in the network to structured beliefs.

2. **Effectors.** They are actions that affect the external world, for example by changing the configuration parameters in the case of a managed network.

3. **Reactors.** They allow the agent to perform defined actions when specified situations occur on the network. The agent can use reactors to have prompt reactions to events that may occur. Precisely, a reactor links a plan of actions to a situation expressed on the agent beliefs.

4. **Calculators.** These are a particular kind of actions that allow to compute or deduce new beliefs from others. For example, suppose that the printing system status is Ok only if the statuses of all the printers in the network are Ok. A calculator can be used to maintain the printing system status belief by continuously checking the statuses of all the printers in the system.

- **Motivations:** Motivations are the main essence of actions in the agent. A motivation lets the agent have preferences towards certain states of its environment. It can be viewed as an expression that the agent continuously tries to satisfy. When a motivation is violated, the agent tries to figure out the reason behind it, then generates goals that are believed to help satisfying the motivation when achieved (Figure 1).

At this stage, abstraction is made on which goal generation mechanism is better suited for the agent. This deliberation process should be chosen at a later stage in the agent development according to the application needs. For example, goal generation can be done by comparing the motivation expression to the current beliefs. By analyzing the resulting differences, the agent can determine what goal is to be generated.

- **Goals:** A goal denotes a state that the agent wants to achieve through the execution of a certain plan of actions [4]. A sequence of actions among the agent capabilities are identified in order to be executed.

Similarly to the goal generation process, abstraction is made on which planning method should be used. At a later stage in the agent development process, it will be decided to whether an embedded planning system or a simple search in the agent's plans is better suited for the application requirements.

We identify two kinds of goals: *achievement goals* and *maintenance goals*. The action plan generated for

an achievement goal should only achieve the goal at some moment in time, whereas the plan generated for a maintenance goal should ensure that the goal expression is hold continuously.

Finally, negative expressions can be used in goal expressions. This can be used for example to cause a preceding achieved goal to be cancelled (or unachieved). Of course, the planning process that is to be chosen has to take into account whether closed world assumption is assumed or not.

- **Intentions:** An intention is an action or a plan of actions that the agent decides to execute in order to achieve a certain goal. Therefore, the intentions corresponding to a goal are a description of how to invoke a set of agent capabilities. This description is used by the operational layer to perform and control the execution of the generated plan (Figure 1).

3.2. The Operational Layer

The agent Operational Layer is an integrated environment within which the actions intended by the agent are executed. It is delivered with intentions decided at the planning process to ensure their execution control. Therefore, the operational layer uses the capabilities description to correctly instantiate and launch the action execution. In addition, the operational layer ensures the proper update of agent beliefs following the execution or the completion of an action.

In the case of a sensor, the agent intentions can specify its activation by executing the primitive `startSensor` with the sensor parameters. For example, if the agent wants to activate `BandwidthUsageSensor` between two adjacent points, it has to execute:

```
(startSensor BandwidthUsageSensor :source
host1 :dest host2).
```

This will cause the corresponding belief to be created and maintained in the agent belief database. To stop a sensor, the agent should execute

```
(stopSensor BandwidthUsageSensor :source
host1 :dest host2).
```

This causes the corresponding belief to be removed from the agent belief database.

Similarly, calculators and reactors can be started and stopped using the respective primitives `startCalculator`, `stopCalculator`, `startReactor` and `stopReactor`. For example, an emergency reactor can be used to kill user processes that are excessively using a machine resources. It can be launched on host 'dahlia' using:

```
(startReactor KillConsumingProcesses :host
dahlia).
```

Of course, there should exist beliefs in the agent telling

which are the resource consuming processes on the target host.

Finally, effectors can only be executed in a one-shot way. For example, the kill process effector can be invoked as follows:

```
(effector KillProcess :host dahlia :pid
4585).
```

3.3. Agent Communication

Communication means are viewed as particular agent effectors and sensors. To send a KQML message, the effector `KqmlSend` is used, with the message text as parameter. Similarly, KQML messages sent by other agents are received using the `KqmlRecv` sensor.

4. Abstract Agent Design

According to the problem description in Section 2, an agent has two types of activities. The first type of activity is related to the usual management tasks that the domain for which the agent is responsible requires. For the need of this prototype, we chose the sample management task of monitoring the network elements in the agent's domain, for example to detect the global status of each network element. Hence, the first role that an agent has to ensure is that of *domain monitoring*.

The second type of activity is related to the detection of unreliable agents within the management agent system. This activity requires two roles, the role of being a *tester*, and that of being a *testee*.

The specification of these roles using the abstract agent model is described in the following sections.

4.1. The Domain Monitoring Role

Goal Generation To have an agent ensure the role of domain monitoring, it can be motivated to have the status of each of the network elements in its domain. If we suppose that beliefs that express that a network element is included in the agent's domain are expressed as follows:

```
(InMyDomain :ne hub101) //ne: Network Element
```

and that beliefs telling of the monitored status of a network element are of the form:

```
(NetworkElementStatus :ne hub101 :status
OPERATING),
```

then the domain monitoring motivation can be expressed like the following:

```
(InMyDomain :ne ?e) ⇔ (NetworkElementStatus
:ne $e) (MI).
```

This motivation makes the agent try to have the status of a network element as soon as it is or becomes part of its

domain. The motivation is violated when a network element e belongs to the agent's domain, but the agent does not have its status in its belief database. This causes the agent to create a goal of the form:

```
(achieve (NetworkElementStatus :ne $e)) (G1).
```

(M1) can also be violated when a network element is removed from the agent's domain, in the case of domain re-assignment for example. This situation also leads to the violation of the domain monitoring motivation since the statement (NetworkElementStatus :ne \$e) holds while (InMyDomain :ne \$e) does not. For this situation, the generated goal would be:

```
(achieve (not (NetworkElementStatus :ne $e))) (G2).
```

Intentions To achieve goal (G1), the agent needs a sensor to perceive the status of the network element. The StatusMonitoringSensor is defined for this reason. To start it on a network element, the agent must execute:

```
(startSensor StatusMonitoringSensor :ne $e)
```

which will create and maintain the missing NetworkElementStatus belief, thus leading to the satisfaction of the motivation.

Goal (G2) can also be easily achieved, assuming a closed world assumption for the NetworkElementStatus beliefs, by executing:

```
(stopSensor StatusMonitoringSensor :ne $e).
```

4.2. The Tester Role

Goal Generation An agent A ensures the tester role regarding an agent B when it has a belief on the reliability of agent B. Such a belief can be written as (AgentSldStatus :agent B :status reliable). Therefore, to make agent A ensure the tester role, it should simply be motivated to having

```
(AgentSldStatus :agent B) (M2).
```

Satisfying this motivation requires the generation of multiple successive goals. Firstly, the tester should have a belief containing a set of, let us say, three representative elements in agent B's domain. Therefore, the following goal has to be generated:

```
(achieve (AgentThreeImportantElements :agent B :element1 ?elt1 :element2 ?elt2 :element3 ?elt3)) (G3)
```

Once this goal is achieved, or if it is already achieved, the next step is to have agent B's belief on elt1, elt2 and elt3. For each of these elements, a goal

```
(achieve (BelievedNetworkElementStatus :agent B :ne $elt)) (G4)
```

is generated, where the belief (BelievedNetworkElementStatus :agent B :ne \$elt :status DOWN) tells that agent B believes that the status of elt is down.

Afterwards, agent A has to start monitoring the same three elements. This is done exactly in the same way as for the domain monitoring role, i.e. by generating goal (G1).

Finally, if all the above goals are achieved, then all the beliefs required to deduce agent B's reliability status are available. The goal:

```
(achieve (AgentSldStatus :agent B)) (G5)
```

can be generated and successfully achieved.

Later in time, the domain of agent B might change. One (or more) of the three elements, let us say elt1, that have been chosen in the (AgentThreeImportantElements) may be replaced by another element, e.g. elt4. When agent A get informed of the change, the motivation becomes violated since the status belief on elt4 is missing and, hence, theSLD status of agent B cannot be maintained. Consequently, agent A will successively generate the following goals:

- (achieve (not (BelievedNetworkElementStatus :agent B :element \$elt1))) (G6).
- (achieve (not (NetworkElementStatus :ne \$elt1))), which makes the agent stop the StatusMonitoringSensor on \$elt1.
- (achieve (BelievedNetworkElementStatus :agent B :ne \$elt4)).
- (achieve (NetworkElementStatus :ne \$elt4)).

Intentions There can be two possible plans to achieve goal (G3). The first plan consists in sending a subscription KQML message to agent B, asking it for all the elements in its domain. By collecting this information, agent A can chose three network elements according for example to their degree of importance in agent B's domain. Therefore, agent A generates and maintains itself the required belief of which three elements to monitor in agent B's domain.

The second plan consists in delegating the same goal (G3) to agent B, and then sending a KQML subscription message asking for belief (AgentThreeImportantElements :agent B). Therefore, the actual determination of the three elements is performed by the testee agent itself.

Both plans lead to the achievement of generated goal (G3). However, the second plan performs better, since in the case of multiple testers for agent B, the AgentThreeImportantElements belief is computed only once. Also, communications overhead are less than in

the first plan, since the first plan requires to send a KQML `tell` message for each element in the domain of agent B.

Concerning goal (*G4*), it can be achieved by sending a subscription message to agent B, asking it for the status of the required network element. Each time agent B sends back a `tell` message stating a change in its belief on the status of `elt`, agent A converts it to a (`BelievedNetworkElementStatus :agent B`) belief.

Finally, goal (*G6*) is achieved in the opposite way. The agent will unsubscribe on the (`NetworkElementStatus :ne $elt1`) belief in agent B, and the corresponding `BelievedNetworkElementStatus` will be retracted.

4.3. The Testee Role

The testee role is a passive role, in the sense that there is no need to motivate the testee agent to make it ensure the role. In fact, the testee agent only replies to the queries sent by the tester agent, and achieves what it is asked to do. For example, when the tester receives the message containing goal (*G3*), it integrates the goal in its mental cycle and tries to achieve it. The achievement of this goal is simply done by activating the corresponding calculator `TopThreeImportantElements` which will create the belief (`AgentThreeImportantElements :agent B`) and update it whenever the domain of agent B evolves.

In addition, the testee agent must be able to manage multiple subscriptions originating from different testers. For example, suppose that there is another agent, agent C, that is also testing agent B. If later in time, agent C is no longer motivated to test agent B, then agent C should unsubscribe on agent B's belief (`AgentThreeImportantElement :agent B`), and then tells that agent B no longer needs to have this belief achieved. However, agent B should be aware that agent A still needs the belief and therefore, it should not stop maintaining it.

5. Conclusion

The purpose of this paper was two fold. From a Network Management point of view, we addressed the problem of the reliability of a distributed NMS based on autonomous Intelligent Agents. Autonomous management agents are capable of performing critical management tasks, and therefore, it is essential to ensure their reliability. We proposed a mechanism by which, the different agents are mutually testing each other by comparing subsets of their respective beliefs. The SLD algorithm applied on their mutual results allowed to detect the unreliable agents. This by itself allows to further improve the reliability of the NMS by having the other reliable agents undertake the management tasks that were performed by the unreliable agent. The result is a higher

degree of fault tolerance and graceful degradation of the distributed NMS.

From an Intelligent Agent architectural view, the work described in this paper allowed to experiment a BDI architecture in a concrete case study from the NM domain. The proposed abstract agent model allowed to specify the application in a straightforward way by using the metaphor of mental categories. The implementation of the agent system lead to a robust application.

Though this case study was developed for an academic purpose to experiment the BDI agent architecture potentials, it still can be applied in practice within a NMS based on intelligent agents.

References

- [1] G. Berthet. *Extension and Application of System-level Diagnosis Theory for Distributed Fault Management in Communication Networks*. PhD thesis, École Polytechnique Fédérale de Lausanne, Lausanne, CH, 1996.
- [2] M. Cheikhrouhou, P. Conti, J. Labetoulle, and K. Marcus. Intelligent agents for network management: Fault detection experiment. In M. Sloman, S. Mazumdar, and E. Lupu, editors, *Distributed Management for the Networked Millennium*, IFIP/IEEE International Symposium on Integrated Network Management, pages 595–610, Boston, MA, USA, May 1999. IEEE Publishing.
- [3] G. Goldszmidt and Y. Yemini. Distributed Management by Delegation. In *The 15th International Conference on Distributed Computing Systems*. IEEE Computer Society, June 1995.
- [4] T. J., Norman, and D. Long. Goal creation in motivated agents. In M. Wooldridge and N. R. Jennings, editors, *Proceedings of the 1994 Workshop on Agent Theories, Architectures, and Languages*. Springer, 1995.
- [5] Y. Labrou and T. Finin. A proposal for a new KQML specification. Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, Maryland, USA, February 1997. <http://www.cs.umbc.edu/kqml/papers/>.
- [6] H. S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(3):205–244, October/November 1996. Available at <http://www.cs.umbc.edu/agents/introduction/ao/>.
- [7] F. Somers. Hybrid: Unifying centralised and distributed management for large high-speed networks. In *Networks Operation and Maintenance Symposium (NOMS96)*, Kyoto, 1996.
- [8] M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.
- [9] S. Znaty, M. Lion, and J.-P. Hubaux. Deal: A delegated agent language for developing network management functions. In *First International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology*, 1996.