A New Organisational Framework for Network Modelling Using a Multi-Agent System

Marc Lemercier and Dominique Gaïti
[1] *Laboratoire de Modélisation et de Sûreté des Systèmes (LM2S)*
University of Technology - Troyes (UTT) – France, 12 rue Marie Curie, BP 2060, Troyes Cedex
Email : { marc.lemercier, dominique.gaiti }@utt.fr
[2] *LIP6*
University of Paris 6, 8 rue du Capitaine Scott, 75015 Paris – France
Email : dominique.gaiti@lip6.fr

Abstract In this paper, we present a platform called MadKit that allows the generic development of multi-agent system based on the organisational concept. Three main objects are defined: agents, groups and roles. We propose to illustrate this concept of organisation in a telecommunication environment. Our main objective is to show the possible use of the MadKit platform in the context of network modelling. In a first approach, we propose a model and implement the superposition of ATM traffics with the help of the MadKit synchronous engine. In a second and third approach, we have modelled the active network architecture (ANTS) and a LEO telecommunication satellite network. We propose the modelling of our three examples by organisational structures. These solutions fill the gap of the actual modelling processes when the network becomes complex (different Quality of Service, resources management, ...) and dynamic.

Keywords: Network modelling, multi-agent platform, ATM, Active Network, Constellation of satellites.

1. INTRODUCTION

The increase of throughput proposed by current telecommunication systems makes network evaluations difficult to realise. At the same time, users ask for a real Quality of Service (QoS) and for more and more services.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35584-9_19

More traffic, a better QoS and a large service offer are not design to work in a coordinated manner in the actual network. New concepts such as the active network, able to modify the equipment behaviours, brings a new dimension in the dynamic of the network and a response to our problem. But nowadays, no simulator is able to handle the changes of network behaviours in face of a breakdown, a routing modification or an intelligent flow control able to adapt itself to the network state. It is obvious that we have to analyse systems in a dynamic manner and no more in a static manner. All these evolutions are difficult to handle. Many simulation environments allow the modelling of telecommunication systems [OPNET 2001], [HyPerf 2001], [Modline 2001], [NS2 2001]. But these techniques have limits when the environment becomes dynamic.

One of our project named MACSI [RNRT 2001] has for purpose to study and to propose new methods for modelling and simulating telecommunication networks. These methods will take into account the dynamic aspect and the evolution of networks as well as the notion of behaviour. A second objective will be to give an evaluation of the quality of results in this dynamic context. Our project has to propose a new methodology of network simulations taking into account the environment by using a multi-agent approach. This solution considers that users' traffics and the behaviour of the network result from interactions between the different elements (actors) in a telecommunication system. We choose to design a model of the system by different actors (agents of modelling) and to simulate this system. Each actor is autonomous and is defined by its knowledge and its interactions.

In this paper, we focus on the organisational approach to model a telecommunication model. Behaviour modelling constitutes another approach and is not the purpose of this proposal. The remainder of this paper is organised as follows. Section 2 presents a multi-agent platform called MadKit. This platform is useful to describe our system in an organisational manner. We show different concepts used by MadKit and a way to simulate complex systems. Some applications in a network environment are proposed in section 3. Three examples are proposed and described: a simulation of the ATM traffic, a model of the ANTS active network and an organisation of a constellation of satellites. Section 4 concludes this paper.

2. THE MADKIT PLATFORM

Several multi-agent platforms have been proposed allowing the development of complex system with the help of agents. The main insufficiency of these approaches is the lack of an organisational structure for the agents. Some researchers [Ferber 2000] have proposed a multi-agent platform named MadKit based on three concepts: agent, group and role. The generic development of a multi-agent system and the agents' organisation constitute the central proposal of this platform [Madkit 2000]. Two structure levels are proposed: the group and the role. An agent belongs to one or several groups, and inside a group an agent can play one or several roles. A role can be seen as a particular function of an agent.

From the agents' cooperation point of view, these organisation concepts allow to structure dialogues between agents. An agent can communicate directly with an other agent identified by its address or can broadcast the same message to each agent with a given role in a group.

2.1 Architecture of the MadKit platform

The platform has been built with respect of the conceptual model based on notions of agents, groups and roles. It is based on a Java little kernel responsible of the management of agents with the help of three components: a "*group and role manager*", a "*synchronous engine*" and a "*local messaging*". To lighten the kernel, a certain number of its services has been agentificated (i.e. transformed into MadKit agents), such as the "*GraphAgent*" or the "*AgentLister*" of the madkit distribution [Madkit 2000]. The independent graphic interface is based on the Java Beans specification. Thus, each agent is responsible for its graphic interface. It is possible to use several functional modes for the platform such as simple consoles, Java applets, or the MadKit default environment: the G-box [Gutknecht 2000]. It allows to launch the execution of agents and to visualise their behaviours.

2.1.1 Creation of agents

The MadKit platform proposes useful services for the creation of generic agents using the Java class *Agent* of the MadKit kernel. These agents constitute the solution for the development of a multi-complex agent system. The agent initialisation in term of groups/roles is described in the method *activate()* automatically launched by the MadKit kernel. The manager of the system has many primitives grouped in the Java API of the platform that will be able to be called from the method *activate()*. Thus, an agent can create and integrate groups and roles, has the capacity to launch and afterwards to kill others agents. This method *activate()* can be seen as the constructor of the agent. Then the agent activity is described by its method *live()*.

In its kernel, the MadKit platform integrates several message formats allowing dialogues between agents. The approach followed by MadKit is both textual messages or document, classically used in an object context

(StringMessage class or XMLMessage class) and more structured messages from the artificial intelligence area (ActMessage class, ACLMessage class and KQMLMessage class).

A method allows all agents to send a message, directly to another agent whose address is known with the primitive *sendMessage()*, or to broadcast this message to all agents with a given role in a group using the method *broadcastMessage()*. Concerning the reception of messages, each agent has a mailbox where received messages are memorised. Two primitives of low level allow the access to messages. The non-locking primitive *isMessageBoxEmpty()* looks into the mailbox and the locking primitive *nextMessage()* extracts and returns the first message from the mailbox.

We used the development kit of the MadKit platform to illustrate its main concepts: the notion of organisation brought by the membership to groups and roles, the exchange of messages between agents.

2.1.2 Remarks and statement

The elements used by MadKit for the transfer of information between agents imply that multi-agent systems constructed from this platform are characterised by a total distribution of knowledge, partial results and methods used to get a result. In this case, agents have a view more or less precise of the other agents of the system. They have therefore to be able to get and to represent intentions and commitments of the other agents. The main difficulty of this approach is generally the representation of each agent's knowledge as well as the update of this knowledge [Pujolle 2000].

Agents extending the MadKit *Agent* class are executed inside a Java thread. Although this approach constitutes a less expensive solution than to process classical systems, the execution in parallel of several hundreds of agents is not foreseeable. This solution is useful only for small applications of multi-agent systems. In a complex system simulation context, it is necessary to have another kind of agent, closer to the notion of a Java object, which do not need the use of a thread. The following paragraph presents solutions proposed by the MadKit platform in the case of a complex system in general or a multi-agent simulation.

2.2 A MadKit proposal for complex simulations

A new architecture called "*The Synchronous Engine*" is proposed in the platform. It is based on five elements:

1. The ReferenceableAgent. In the framework of the synchronous engine, an agent can be planned or monitored. It is necessary to have a reference allowing a direct access to the agent what was not possible for security

reasons with the first release of MadKit. Therefore, the designer has to specify these intentions by using the *ReferenceableAgent* class;

2. The Scheduler agent. The Scheduler agent is a standard MadKit agent that has to manage all *Activator Agents* for the execution of synchronized agents;
3. The Activator class. The Activator tool defines a planning policy. It works with the *Scheduler agent* to obtain the list of Referenceable agents. An *activator* belongs to a group and has a role. Its method *update()* allows it to discover dynamically the agent implementation from a given group and role. The *Scheduler agent* will invoke its method *execute()* to launch the execution of the simulation;
4. The Watcher agent. The *Watcher agent* manages a list of probes. This is not a threaded agent as the *Scheduler agent*, but it has therefore to be executed in association with a scheduler;
5. The Probe class. The *Probe* class allows defining a code for the exploration of a *ReferenceableAgent.*

Figure 1 presents elements of this synchronous engine such as it is explained in the MadKit documentation [Gutknecht 2000].

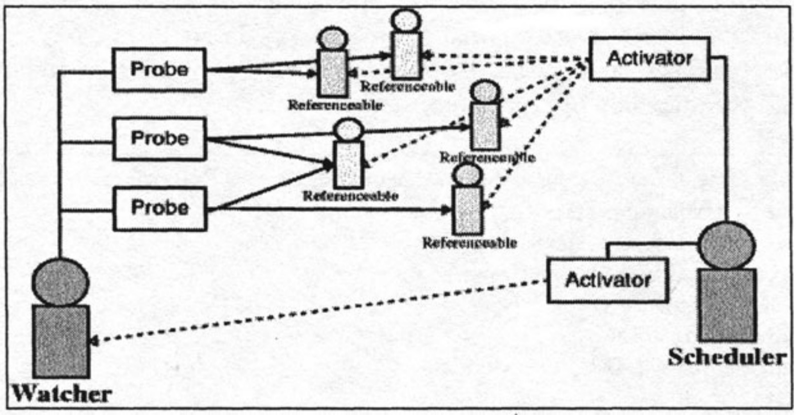


Figure 1. MadKit synchronous engine

3. APPLICATIONS TO NETWORK

In the context of our MACSI project [MACSI 2001], we wish to conceive a simulator able to take into account both the complexity of current high throughput networks as well as an active network in which the behaviour of equipment can be modified by the traffic encountered. The main objective of this study is to validate if possible the use of the MadKit platform for complex network simulations.

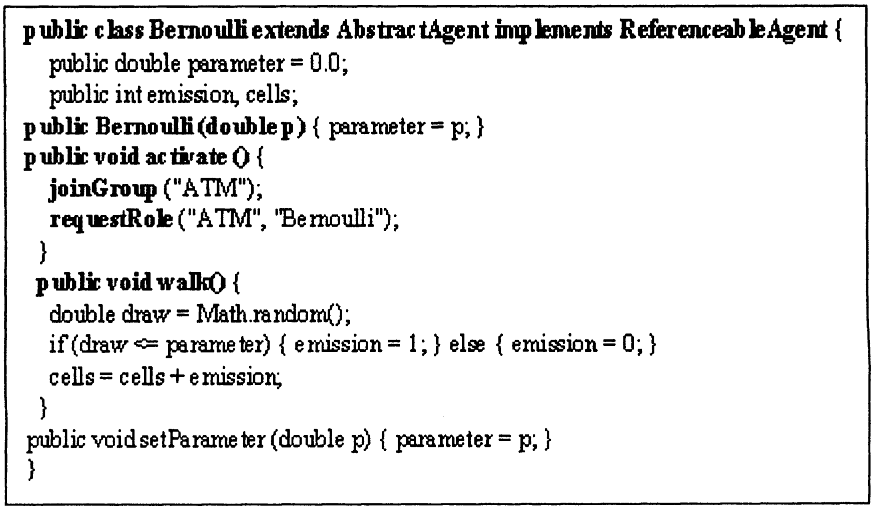
This paragraph presents a model and an implementation of ATM traffics with the help of light agents (*ReferenceableAgent*) of the MadKit synchronous engine. We then model the active network ANTS and a constellation of satellites.

3.1 Example 1 : ATM traffic

To illustrate the possible use of the MadKit synchronous engine in a network context, we have chosen to simulate a superposition of ATM traffics. In an ATM network, time can be split in time-slots equal to the time of a cell emission. The chosen traffic model is a Bernoulli source with parameter P (P ∈ [0, 1]). Thus, to each time-slot, a source sends an ATM cell with a probability P. MadKit Referenceable agents represent these sources.

3.1.1 Step 1: creation of a Bernoulli class

Figure 2 presents the code of an agent implementing the Java *ReferenceableAgent* interface that represents a Bernoulli(p) source. The variables (***emission*** and ***cells)*** are defined as attributes of the class to be then observed with the help of probes (see step 4).



```
public class Bernoulli extends AbstractAgent implements ReferenceableAgent {
   public double parameter = 0.0;
   public int emission, cells;
public Bernoulli(double p) { parameter = p; }
public void activate () {
   joinGroup ("ATM");
   requestRole ("ATM", "Bernoulli");
  }
  public void walk() {
   double draw = Math.random();
   if (draw <= parameter) { emission = 1; } else { emission = 0; }
   cells = cells + emission;
  }
public void setParameter (double p) { parameter = p; }
}
```

Figure 2. Bernoulli agent

The method *activate()* insures the recording of the agent in the *ATM* group with the *Bernoulli* role. The *walk()* method represents really the traffic: to each call of this method, we undertake a random draw that is then compared to the value of the parameter p. The attributes *emission* and *cells* are updated correspondingly.

3.1.2 Step 2: The activator

Figure 3 shows the implementation of the activator agent. The role of this agent is to execute a slot-time of simulation by invoking the methods *walk()* of all Referenceable agents already defined. The platform API memorises to this end the set of agents in a table. The *setParameter()* method allows to change the parameter value of Bernoulli sources during the simulation.

```
public class ATM_Bernoulli_Activator extends Activator {
  public ATM_Bernoulli_Activator (String group, String role) {
   super (group, role);
  }
  public void execute () {
    for (int num = 0; num < agents.length; num++)
      ( (Bernoulli) agents[num]).walk();
  }
  public void setParameter (double p) {
    for (int num = 0; num < agents.length; num++)
      ( (Bernoulli) agents[num]).setParameter (p);
}}
```

Figure 3. Activator code

3.1.3 Step 3: The Scheduler

The Scheduler agent possesses an essential function in the synchronous engine. The code (figure 4) of the method *activate()* presents its main responsibilities: record an agent in the group "*ATM*" with the role "*scheduler*", creation of all Bernoulli agents using the *lauchAgent()* primitive, then creation of an observer (see step 4). The method *live()* presents the scenario of simulation. The method *execute()* of the *sources Activator* and the *Observer Activator* are launched in a loop until the end of the simulation.

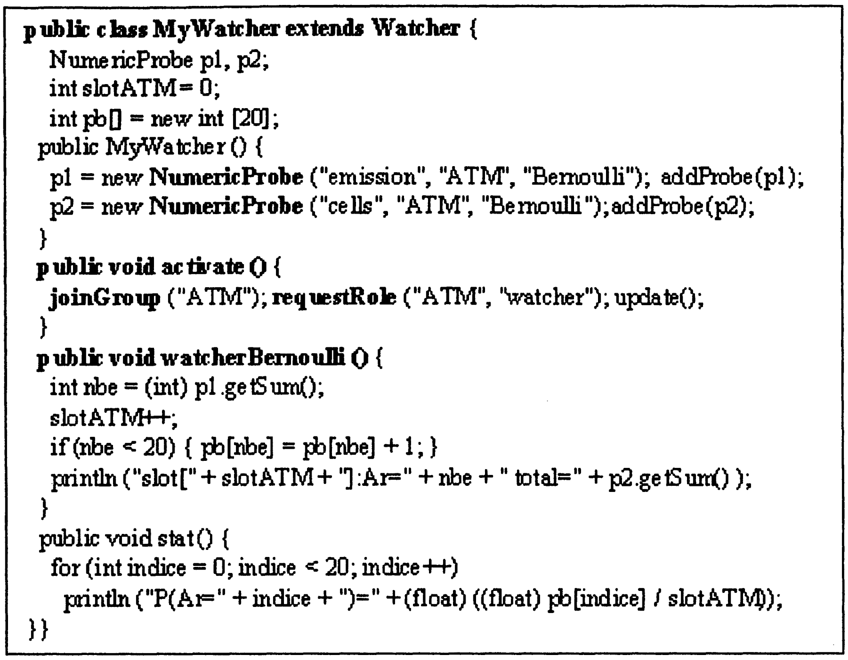
```
public class MySchedulerAgent extends Scheduler {
  int sourcesB = 10, delay= 100;
  double bparameter = 0.1;
  boolean end = false;
  ATM_Bernoulli_Activator a1;
  MyWatcher mObs;

 public void activate () {
  foundGroup ("ATM");
  requestRole ("ATM", "scheduler");
  for (int indice = 0; indice < sourcesB; indice++) {
    launchAgent (new Bernoulli (bparameter), "B", false);
  }
  mObs = new MyWatcher();
  launchAgent (mObs, "my_watcher", true);              // true =GUI
 }
 public void SetParamB (double p) {
  a1 setParameter (p);
 }
 public void live () {
  a1 = new ATM_Bernoulli_Activator ("ATM", "Bernoulli");
  SingleMethodActivator a2 = new SingleMethodActivator (
       "watcherBernoulli", "ATM", "watcher");
  addActivator (a1);
  addActivator (a2);
  update();
  while (!end) {
    pause (delay); a1.execute(); a2.execute();
  }
  mObs.stat();
 }}
```

Figure 4. The scheduler agent

3.1.4 Step 4: the watcher

This agent has to observe the Referenceable agents with the help of probes witch accumulate statistical data on some Referenceable agent attributes. The code of this agent (figure 5) allows us to see that two numerical probes observe ***emission*** and ***cells*** attributes of all Bernoulli sources.



```
public class MyWatcher extends Watcher {
  NumericProbe p1, p2;
  int slotATM= 0;
  int pb[] = new int [20];
 public MyWatcher () {
  p1 = new NumericProbe ("emission", "ATM", "Bernoulli"); addProbe(p1);
  p2 = new NumericProbe ("cells", "ATM", "Bernoulli");addProbe(p2);
 }
 public void activate () {
  joinGroup ("ATM"); requestRole ("ATM", "watcher"); update();
 }
 public void watcherBernoulli () {
  int nbe = (int) p1.getSum();
  slotATM++;
  if (nbe < 20) { pb[nbe] = pb[nbe] + 1; }
  println ("slot[" + slotATM + "]:Ar=" + nbe + " total=" + p2.getSum() );
 }
 public void stat() {
  for (int indice = 0; indice < 20; indice++)
    println ("P(Ar=" + indice + ")=" +(float) ((float) pb[indice] / slotATM));
}}
```

Figure 5. The watcher Agent and the probes

3.1.5 Remarks and statement

This task describes the process of the MadKit synchronous engine in a network context. It is possible to complete this simulation by the definition of others ATM traffic models activated by new Activator agents. In order to complete a multi-agent simulation, the ATM switches would have to be represented by MadKit standard agents able to exchange management information on their states and their behaviours.

The main element to retain is the fact that Referenceable agents are successively activated by an Activator agent. Thus, to each simulation step, each agent executes its task defined in the method *walk()*. The real-time of the simulation step corresponds therefore to the sum of the execution time of all the methods *walk()*. By defect, there is no constraint of time for an agent. The agent has therefore to take a quick decision on the partial realisation of its activity using this method.

It is important to notice that the behaviour of the agent (or its activity) is represented by its *walk()* method that is then invoked to each simulation time unit. The activity of the agent has therefore to be repetitive or to be split into N sub-tasks achievable successively to each call by supposing that the agent knows its execution context.

In practice, in a context of simulation, *Referenceable agents* are messages, capsules in an active network context or cars in a car-traffic simulation. The benefit of the MadKit platform in this approach is the notion of probes able to get quantitative data on Referenceable agents in a very fast way.

3.2 Example 2 : ANTS Active Network

The ANTS active network architecture is a proposal of the MIT where capsules (packets containing data and code) are sent by users' applications and executed in the active nodes of the network. In practice, an active network is a network constituted with traditional IP routers and active nodes having an execution environment. For security reasons, the designers of ANTS have limited the possibilities offered to user's traffics. The manager of the active network develops and installs a certain number of generic routines identified by references. The ANTS capsule is in fact a packet containing data and references to routines of the active network. These capsules are grouped in protocols that are units of management for the system [Wetherall 1998].

Capsules of an active traffic are automatically executed on all active nodes located between the transmitter and the receiver. A mechanism based on mobile code insures the migration of routines in case of necessity. More precisely, a capsule containing the address of the receiver is encapsulated in an IP packet whose address of destination corresponds to the address of the next active node to cross.

An active node proposes some forwarding functions to find the road to the next active node. The development of a service by a user necessitates the definition of the different capsules and routines. Active nodes propose a set of primitives usable by all capsules using a Java API. An example of service can be information caching, a forwarding policy, etc. Other function of a node is to manage resources consumed by a capsule. Each capsule comprises a field *"resources remaining"* in relation to the resource level of the capsule. Active nodes decrease the number of resources consumed by each capsule. A capsule having no more resources is destroyed.

3.2.1 A MadKit model for an ANTS network

We present in this sub-section a MadKit model for an ANTS network. The ANTS network is a classical IP network with standard or active routeurs and a set of telecommunication links. On the network circulate both IP packets and capsules. We propose therefore to represent each node of the ANTS network by a MadKit standard agent. For passive nodes, a classic

MadKit agent is used belonging to the *PassiveNode* group and having the *manager* role. Similarly, for each active node, we create a classical agent belonging to the *ActiveNode* group and having a *manager* role.

IP packets and capsules are numerous in the system, and we choose therefore to represent them by MadKit Referenceable agents. An IP packet is a light agent belonging to the group *UserMessage* and playing two roles: *passivePacket* and *traffic_ID*. The first role is to distinguish a passive packet from an active capsule and the second role is to identify the traffic to which belongs this data-message. The *traffic_ID* identifier can be built from the knowledge of IP address of the transmitter and the receiver. A capsule is therefore a light agent of the group *UserMessage* and has the roles of *activePacket* and *traffic_ID*.

In an ANTS network, each active node can receive many packets and capsules from an active traffic. We know that the capsules' context of execution is independent between traffics. We have chosen to create another MadKit agent for each active traffic crossing an active node.

The role of this new agent (figure 6) is to manage all packets and capsules of its associate active traffic, and to be in fact the delegate of this traffic beside the *manager* agent of the active node. This delegate is able to characterise and define the behaviour of the active traffic with the help of probes, and to negotiate network's resources beside the agent *manager*. This delegate is therefore a standard MadKit agent of the *activeNode* group with *AN_ID* (active node ID) and *traffic_ID* roles. The identifier *AN_ID* designates an active node. Thus, an active node is represented by an agent *manager* and possesses one agent per active traffic crossing it. This model leans on structures of society that we present in figure 7.

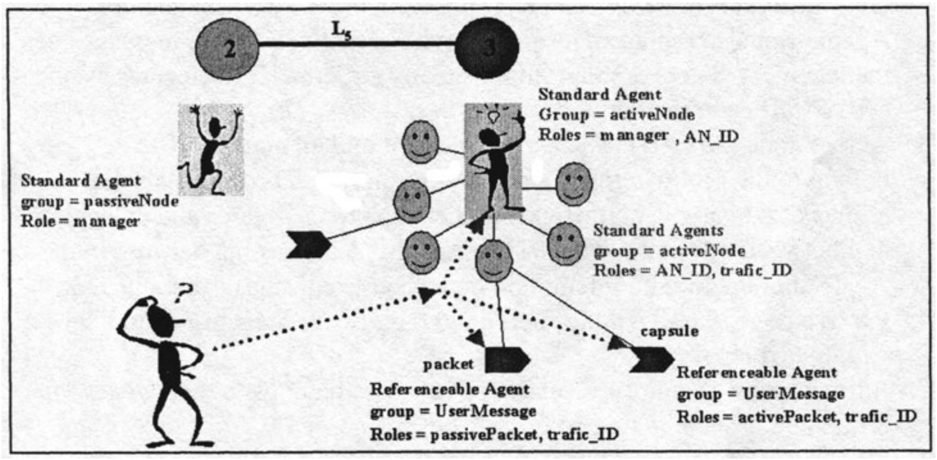


Figure 6. An ANTS active node MadKit model

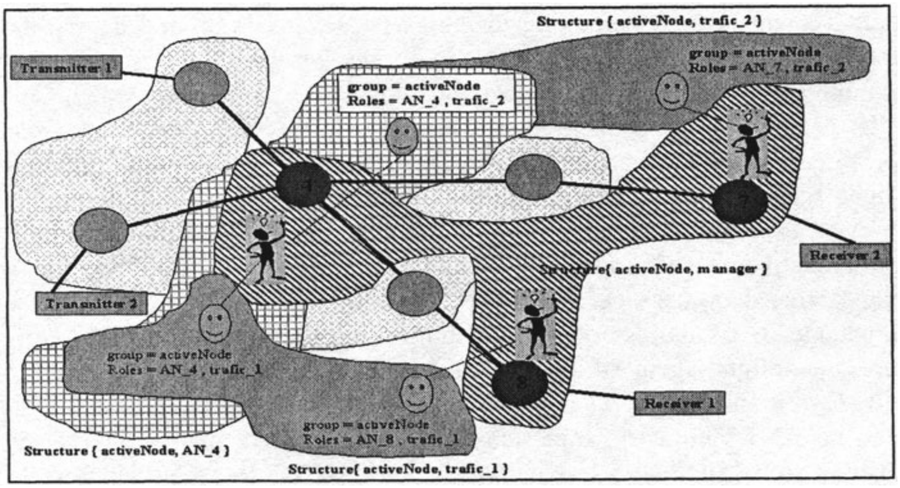


Figure 7. Multi-agent organisational structures

Five entities can be found:

1. IP packets and capsules from an active traffic. Packets and capsules of a same active traffic belong to the same *UserMessage* group and possess in common the *traffic_ID* role. There are Referenceable agents on which a *MadKit Observer agent* can put some probes;
2. Active traffic. In each active node crossed by an active traffic, an agent {group: *activeNode*; roles: *AN_ID* and *traffic_ID* } is created by the *manager agent* of the node. Although different manager agents on different active nodes create agents for a same communication, they take part of multi-agent organisational structures. This proposal allows the possibility of routine exchange proposed in the ANTS architecture. Agents can also represent a same active traffic to exchange messages of management. There are therefore as many structures of delegates as open active traffics in the network;
3. Active node (AN_ID). The *manager* agent and all active traffic delegates belong to the group *activeNode* and have an *AN_ID* role. This solution allows organising data traffics both between the *manager* and delegates, and between delegates. This last possibility is interesting because in the ANTS architecture no interaction between the capsules is anticipated;
4. Passive node. Passive nodes belong to the *passiveNode* group and have a *manager* role;
5. Active nodes. Similarly, a set of active nodes belongs to the *activeNode* group and has a *manager* role.

3.2.2 Organisational model for an active node

This model of an ANTS network leans on organisational concepts of the MadKit platform. We have associated to each element of the ANTS architecture, an agent or a group of agents with respect to the main constraints of the ANTS network. We can see that the use of the MadKit development kit allows constructing a simulation of this active network. For that, we need to describe the network topology, and some scenarios of simulation on this network. These descriptions can be written in a XML document to insure an inter-operability between several multi-agent systems (MadKit, DIMA, etc.).

All along this paper we focus on the organisational approach. Another approach concerns the behaviour-based model of an active node. This approach is currently under study in the University of Technology of Troyes and in the University of Paris 6 [Merghem 2001]. A part of activities of an ANTS node can be represented by a reactive approach. Indeed, the *manager* agent has to arbitrate traffics' requests from all active traffic delegates. However, it seems interesting that this *manager* be able to represent the behaviour of its node and to construct a view of the behaviour of the other active nodes of the network. In order to do that, the MadKit platform proposes the use of a cognitive agent (JessAgent). This improvement is for the moment under integration to the platform. An expert system engine is integrated to a traditional agent. The behaviour of an agent can thus be described by a set of rules.

3.3 Example 3 : Constellation of satellites

The satellite network model is inspired by the one proposed by the French national centre for the space research (Centre National de Recherche Spatiale (CNES)) for our project 'Constellation of Satellites for Multimedia applications'. The 72 LEO satellites are equally distributed among 9 orbits of radius 1603 km and 50 degree equatorial inclination, and have a minimum elevation of 17.5 degrees. Each satellite is equipped with up and downlink transceivers of 155.5 Mbit/s bandwidth, and 4 bidirectional intersatellite links (ISL) also of 155.5 Mbits/s [Sigel 2000].

Thus each satellite has a permanent connection with the previous and the next satellite on the same orbital plan, and two other satellite links to its left and its right. These two last links are not permanent. Several roles are necessary to model the organisation of agents. We have chosen the following notation:

a) O_ id: orbital plan number id (id in [1,9])
b) S_ xz: satellite in the orbital plan x (x in [1,9]) and rank z (z in [1,8])

c) N_ s: neighbourhood of the satellite S

Figure 8 presents five other MadKit agents that we propose to describe the neighbourhood of a satellite. The definition of satellite neighbourhoods produces a great number of roles. We are going to begin with a description of agents located on one satellite (in our example the satellite number 44). The first agent is a creator agent (C) that is the superintendent of the role N_44 (neighbourhood of the satellite 44). This agent is a MadKit agent that belongs to the satellite 44 (group1, role S_44) and does not participate to the communications between satellites (group1, role OUT). It is the creator of the neighbourhood 44 (group2, role C and N_44). The second agent is an agent P (PREVIOUS), located on the satellite 44 (group1, role S_44) but belonging to the neighbourhood of the satellite 43 (group2, role P and N_43) that is the previous satellite on the same orbital plan. This agent is the delegate of the neighbourhood N_43 centred on the satellite 43. So, a satellite has one delegate for each four neighbours. The third agent is an agent N (NEXT). It is the delegate of the neighbourhood of the satellite following on the same orbital plan (N_45). And, the last two agents are delegates of the two closer satellites on others orbital plans.

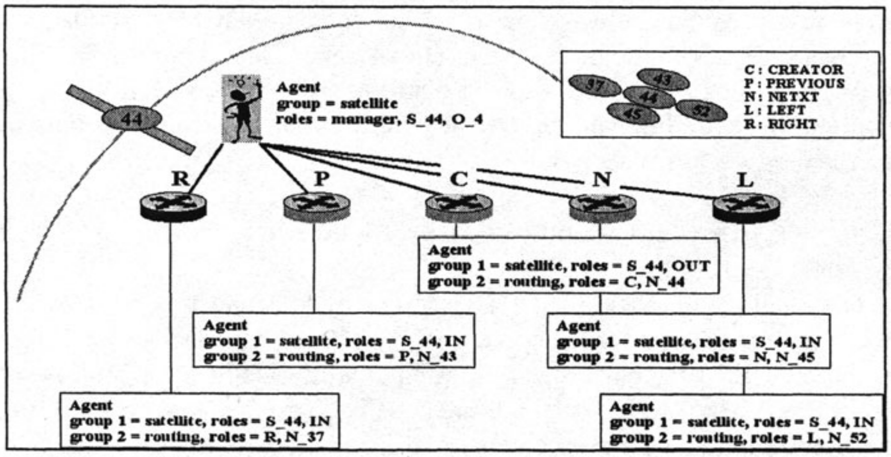

44
Agent
group = satellite
roles = manager, S_44, O_4
37
43
44
45
52
C: CREATOR
P: PREVIOUS
N: NETXT
L: LEFT
R: RIGHT
R
P
C
N
L
Agent
group 1 = satellite, roles = S_44, OUT
group 2 = routing, roles = C, N_44
Agent
group 1 = satellite, roles = S_44, IN
group 2 = routing, roles = P, N_43
Agent
group 1 = satellite, roles = S_44, IN
group 2 = routing, roles = N, N_45
Agent
group 1 = satellite, roles = S_44, IN
group 2 = routing, roles = R, N_37
Agent
group 1 = satellite, roles = S_44, IN
group 2 = routing, roles = L, N_52

Figure 8. MadKit agents for neighbourhood definitions

This set includes the satellites 44, 37, 43, 45, 52. These agents belong to a first group *satellite* allowing a precise identification of these agents (membership to a given satellite), and belong also to a second group *routing* that identifies the function of this agent in the definition of a neighbourhood.

This model defines several MadKit organisational structures and allows the information exchange between agents of even groups with the same role. We notice that this representation is extremely rich and allows thus to define several groups of distribution

4. CONCLUSIONS

In this paper, we proposed the use of the MadKit multi-agent platform in our telecommunication system context. We have presented our main objectives namely the organisational modelling of complex network configurations. This approach allows the conception of several societies of agents in order to organise exchanges of messages without any disruption for all the agents in the simulation.

Other studies have to be performed on the area of the network device modelling. Indeed, the MadKit platform proposes only the language Java for the development of agents and does not propose libraries for network equipments. We work currently to the construction of this generic library usable with MadKit and also with others multi-agent systems. As presented here, the problem of the network behaviour has not been described. Indeed, we work now on the definition of a set of behaviours (careful, careless, etc.) allowing the description of the intern functioning of a network router [Merghem 2001]. Then, a high-level manager agent knowing these behaviours will be able to alter the policy of a router according to criteria of traffic QoS. The final goal of our study is to provide an agent-based network simulation techniques useful when the network becomes dynamic.

REFERENCES

[15] [Ferber 2000] "Multiagent systems for telecommunications: from objects to societies of agents", Jacques Ferber, networking 2000, Paris.

[16] [Gutknecht 2000] "MadKit Development Guide", Olivier Gutknecht, MadKit documentation, version 2.0. http://www.madkit.org

[17] [HyPer 2001] http://www.hyperformix.com/products/products.htm

[18] [MACSI 2001] http://www-lm2s.utt.fr/macsi/

[19] [Madkit 2000] "Madkit official web site", http://www.madkit.org

[20] http://community.madkit.org/

[21] [Merghem 2001] Leila Merghem and Dominique Gaïti, "Active Network Modelling and Simulation: a Behavioural Approach", Smartnet 2002, Finland.

[22] [Modline 2001] http://www.simulog.fr/

[23] [NS2 2001] http://www.isi.edu/nsnam/ns/

[24] [OPNET 2001] http://www.opnet.com/products/modeler/home.html

[25] [Pujolle 2000] Guy Pujolle "Les réseaux", 3ème édition, book in french. Eyrolles 2000.

[26] [RNRT 2001] Official Web Site of RNRT projects http://www.recherche.gouv.fr/technologie/reseaux/rnrt.htm

[27] [Sigel 2000] Eric Sigel, Bruce Denby, Sylvie Le Hégarat-Mascle, "Application of ant colony optimization to adaptive routing in a LEO telecommunications satellite network", IEEE Transactions on Networking, July, 2000

[28] [Wetherall 1998] David J. Wetherall, John Guttag, and David L. Tennenhouse "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols", IEEE OPENARCH'98, San Francisco, CA, April 1998.