
**FACULDADE DE INFORMÁTICA**
**PUCRS – Brazil**
http://www.inf.pucrs.br

# Development of An Active Network Architecture Using Mobile Agents – A Case Study

*Lucio Mauro Duarte and Fernando Luís Dotti*

**TECHNICAL REPORT SERIES**

Number 043
July, 2002

Contact:

lduarte@inf.pucrs.br

http://www.inf.pucrs.br/~lduarte

fldotti@inf.pucrs.br

http://www.inf.pucrs.br/~fldotti

Lucio Mauro Duarte is a part-time professor at PUCRS – Pontifícia Universidade Católica do Rio Grande do Sul – Brazil since 2002. He got his M.Sc. in 2002 at PPGCC/FACIN-PUCRS.

Fernando Luís Dotti works at PUCRS/Brazil since 1998. He is an associate professor and develops research in Computer Networks, Distributed Systems and Code Mobility. He got his Ph.D. in 1997 at Technical University of Berlin (Germany).


Published by the Campus Global – FACIN – PUCRS
Av. Ipiranga, 6681 - Partenon
90619-900 Porto Alegre – RS – Brazil

# Development of An Active Network Architecture – A Case Study

Technical Report 026/2002

*Lucio Mauro Duarte (M.Sc.)*
*Fernando Luís Dotti (Ph.D.)*

## 1 Introduction

The continuous growth in processing and communication capabilities led to massive distributed computational environments, e.g. the Internet. These environments are often called *open environments*, being characterized by: massive geographical distribution; high dynamics (appearance of new nodes and services); no global control; partial failures; lack of security; and high heterogeneity. Building distributed applications for such environments is a complex task. Research efforts have been directed to manage this complexity through the development of new paradigms, theories and technologies for distributed applications. Within this context, *code mobility* [1] has received special attention due to its flexibility and potential use in various application fields, e.g. network management [2], electronic commerce [3], distributed information retrieval [1], advanced telecommunication services [4], active networks [5], and workflow management systems [6].

One of the objectives of the *ForMOS project (Formal Methods for Mobile Applications in Open Environments)* is to create a framework to support the development of mobile applications. The innovative aspect of the framework is the use of *Object-Based Graph Grammars (OBGG)* [7] as the underlying unifying formal method for a set of integrated tools [8]. OBGG is a restricted version of *Graph Grammars* [9] that includes object-based concepts as encapsulation and communication through message passing. Each one of the tools developed in the ForMOS project supports a way of addressing the generation of correct mobile applications. The framework encompasses, by now, a formal specification language, a simulation tool and a mapping scheme to generate code from OBGG specifications.

In this report is presented the active network architecture developed as a case study during the work presented in [10]. This architecture was modeled with the OBGG language, what made it possible to simulate its behavior as well as to generate code for it using the tools developed in ForMOS project. The code was generated for a mobility support platform allowing the execution of the architecture in a real environment.

The architecture considers the use of mobile agents to implement some of its elements. *Mobile agents* are mobile codes that are autonomous in the sense that they can migrate when it is necessary and to where they need. Because of this characteristic, mobile agents can be used to implement elements with execution independence and migration autonomy.

This work is organized as follows: Section 2 presents the concepts involving active networks; Section 3 presents the active network architecture; Section 4 describes

the formal specification of the architecture; Section 5 discusses the simulation and execution results of two testing scenarios; and Section 6 contains the final remarks.

## 2 Active Networks

The traditional function of a network is to route packets from one node to another. Processing within the network occurs to provide routing, congestion control and quality of service (QoS) schemes. So, in traditional networks processing occurs in a static way, through predefined instructions used in the configuration of network devices. Because of this limited processing capacity, these networks can be regarded as *passive networks*. According to [11], some problems that can be identified considering *passive networks* are:

- the difficulty of integrating new technologies and standards into the shared network infrastructure, demanding the substitution of devices or their reconfiguration by network administration;
- poor performance due to redundant operations at several protocol layers;
- the difficulty to accommodate new services in the existing architectural model.

Recently, some applications that require computations within the network have emerged, such as firewalls, web proxies, multicast routers and mobile proxies, increasing the need for an architectural support for these computations.

In order to overcome all the presented problems, the idea of *active networks* was proposed. Active networks [5] represent a new approach for network architectures. They are named *active* because the routers and switches in such networks can perform computations on user data flowing through them. Programs can also be dynamically inserted into the network nodes to configure them according to the needs of the applications in execution. This way, packets have the capacity of carrying not only data but also the code to be executed in remote nodes. Therefore, the user has the possibility of "programming" the network, providing the programs to be used by the routers and switches to execute their computations. In an active network the difference between network internal nodes (routers, switches, etc.) and user nodes is tenuous, since both are able to perform the same computations. Hence, the user can view the network as a part of his application and can adapt the network to obtain the best performance of his/her application.

As presented in [11], there are three approaches for active networks architectures:

- *Active packets*: Packets (here called *capsules*) flowing in the network carry code to be executed in the nodes where they pass through. Nodes have only the capacity of performing computations using the received codes. Some examples of this approach are *Smart Packets* [12] and *Active IP Option* [13];
- *Active nodes*: Packets carry only identifiers or references to predefined functions present in the nodes. So, packets carry the identification of what kind of function must be executed to process their data. Nodes own the codes of functions needed to process the data present in the packets, according to the identification provided by them. Examples of this approach are *DAN* [14] and *ANTS* [15];
- *Active packets and nodes*: This approach combines the other two presented approaches. Packets carry real and simple codes and nodes own more complex codes. This way, packets don't carry a complex and large code,

which resides in the nodes. The simple code carried by the packets can be used to determine how their data must be processed or which function provided by the node must be used to process the data. Therefore, such architectures allow users to choose between the two other approaches according to their applications needs, i.e., this approach allows using either active packets or active nodes approach. As examples of this approach we can mention *SwitchWare* [16] e *NetScript* [17] architectures.

Some known applications of active networks are network management [18], active congestion control [19], active multicast [20] and active caching [21].

## 3 An Active Network Architecture

As said before, in [10] an architecture for active networks involving mobile components was created as a case study to test the tools developed in the ForMOS project. Such architecture, as designed, demands the intense use of mobility. This characteristic made the development of this case study very valuable to prove the usefulness and powerfulness of the implemented tools. Furthermore, representing active networks in terms of our formalism makes it possible to simulate, analyze and generate an implementation for it. Since active networks are a new area, there is a lack of simulation and analysis tools. Besides this, the existence of a way of mapping the specification to a programming language and to a mobility support platform prevents the occurrence of errors when one translates from the formalism to executable code. This helps in maintaining the correctness of the code according to the specification.

The architecture considered is composed by capsules, active nodes, services, code bases and naming servers:

- *Active nodes* are the nodes of the network and can send and receive capsules in unicast and broadcast modes. They provide services to the received capsules;
- *Capsules* represent the packets transferred over the network. They can carry data and code. Each type of capsule is handled by a service;
- A *service* is an entity that owns the code to be executed with the data of a specific type of capsule. Services execute in active nodes and determine what types of capsules can be handled in each active node;
- A *code base* is an entity that maintains instances of services available in the network. It provides instances of services required to handle specific types of capsules to the active nodes;
- A *naming server* is an entity that owns a list of all registered entities, associating a name to a reference to each entity. This way, a naming server provides the mapping of a name to an entity reference. It enables the communication between entities with location transparency. The naming server also contain the list of local services of an active node.

This architecture follows the active nodes approach, according to the classification discussed in Section 2, once the capsules carry only the identification of the service it requires to process its data.

Active nodes are static entities and capsules, services, code bases and naming servers are mobile entities. That is, all entities of the architecture, except the active

nodes, can change their location. In fact, only capsules and services take profit of mobility in the current state of the architecture.

The basic behavior of the proposed active network architecture will be described in the next section, where the formal specification of this architecture is presented.

## 4 Specification of the Proposed Architecture

The specification of each entity in OBGG involves the definition of a type graph and a set of rules. The *type graph* of an entity presents its internal attributes, the messages it can handle and the messages it can send to other entities (messages are presented with their parameters).

The type graph for an active node (AN) is depicted in Fig. 1. The active node is presented with its attributes on the left side. Each attribute is presented with its respective name and type. On the right side of the AN there are the messages the active node can receive from other entities. The parameters of each message are presented on the right side of it. On the right side of Fig. 1 are the messages an active node can send. The destination of each message is represented by a *to* arrow. The entities that behave as a static entity are represented by two concentric ellipses and mobile entities by a simple ellipse.

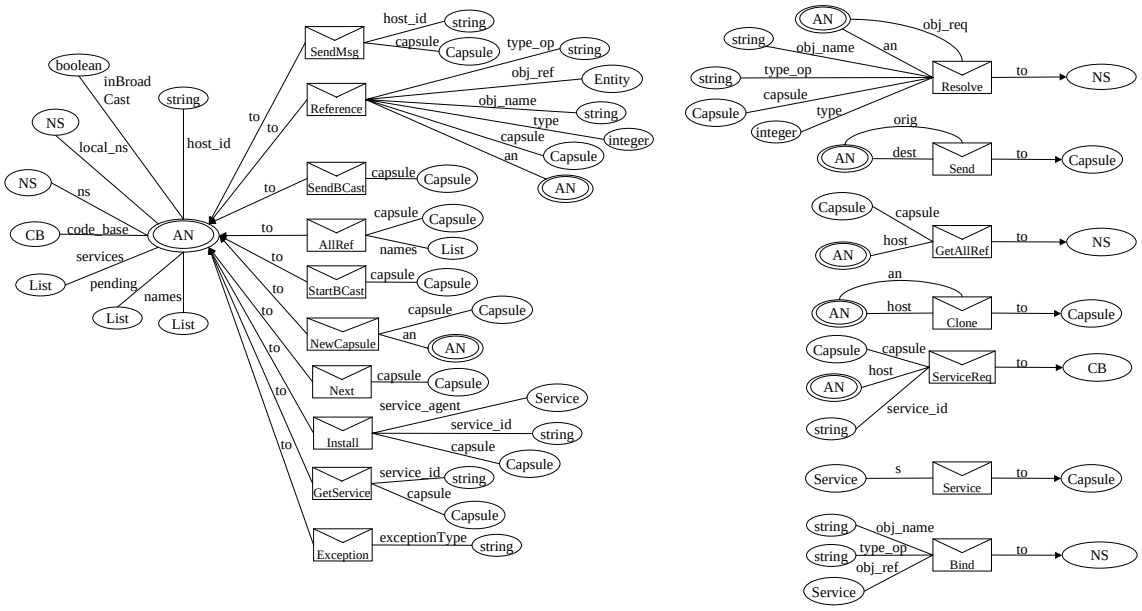


**Fig. 1. Type graph for an active node.**

An AN is identified by a *host id*, which is used to communicate to other nodes. Each AN owns a reference to a code base from which it can require instances of services. The AN owns also a list of the local services presently available and a reference to a local naming server used to register the references to the local instances of the services and references to known nodes (nodes which the AN is connected to).

The type graph for a capsule is presented in Fig. 2.

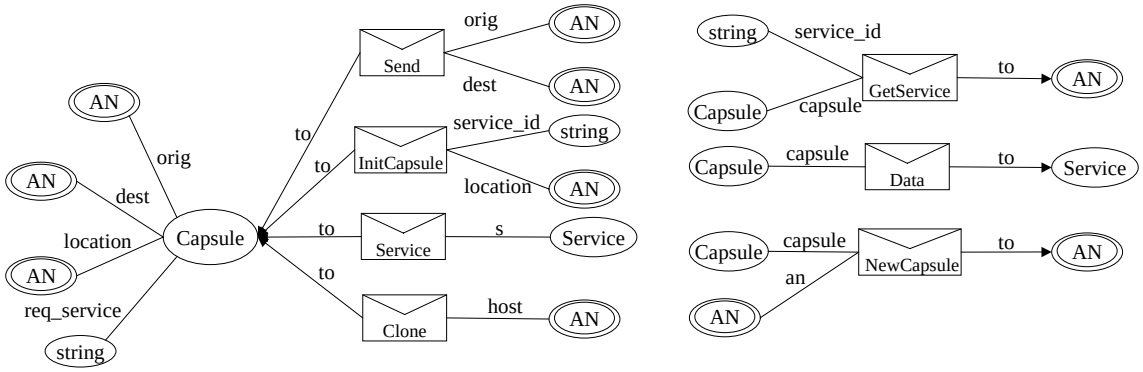


**Fig. 2. Type graph for a capsule.**

A capsule has a reference to its origin node and to its destination node as well as to its current location. It also owns an attribute used to store the identification of the service required to handle this capsule. This way, the active node that receives this capsule must provide the defined service to process the data of the capsule.

Fig. 3 depicts the type graph of a service.

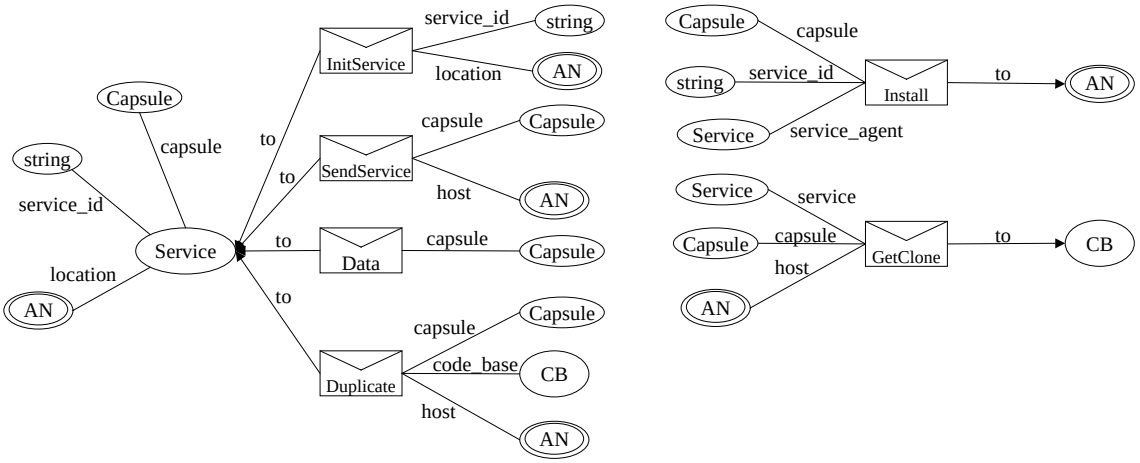


**Fig. 3. Type graph for a service.**

A service has an identification and stores the information of its current location and a reference to a capsule it must handle.

The code base (CB) stores the list of all available services in the network and a reference to a local naming server that contains references to the services instances. Besides this, the CB owns the information of its current location. The type graph for a code base is presented in Fig. 4.

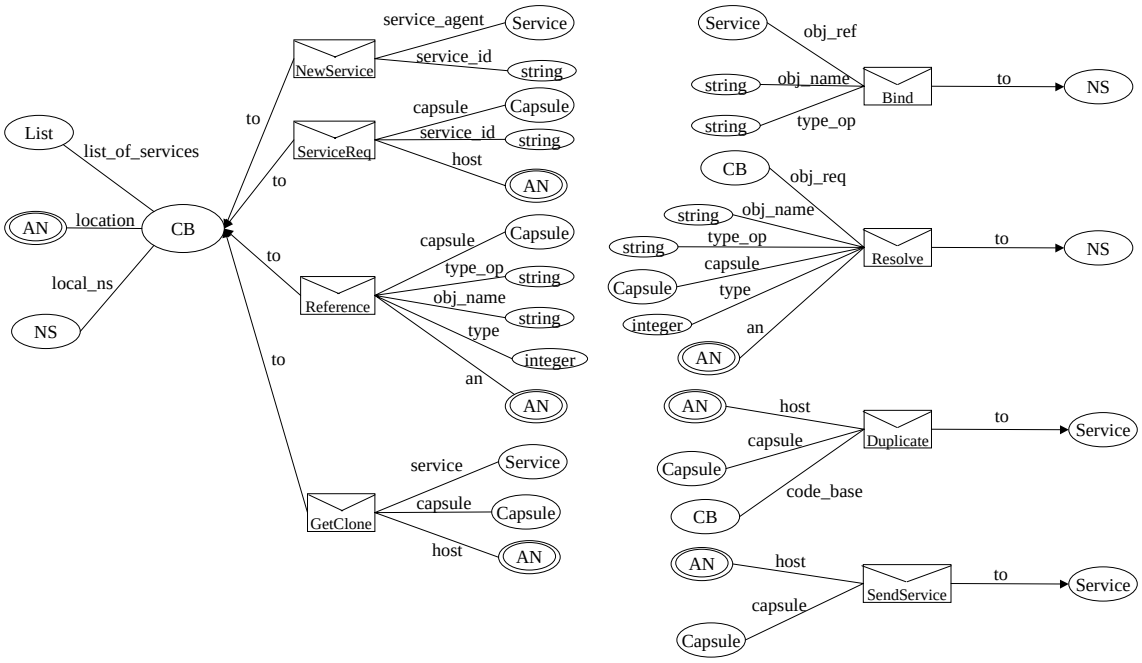


**Fig. 4. Type graph for a code base.**

The type graph of the naming server (NS) will not be presented here. It is widely discussed in [22], presenting its basic behaviour. In this work it is just considered that the naming server is an entity that receives requests from other entities and maps names to references.

As said in the beginning of this section, an OBGG entity is defined by a type graph and a set of rules. The set of rules associated to an entity defines its behavior. The application of rules successively changes the state of the system, starting from an initial state, called *initial graph*. The initial graph describes the entities involved in the system and the initial values of their attributes. An OBGG rule is composed by a left side, a right side and a condition. An OBGG rule is graphically represented as shown in Fig. 5.

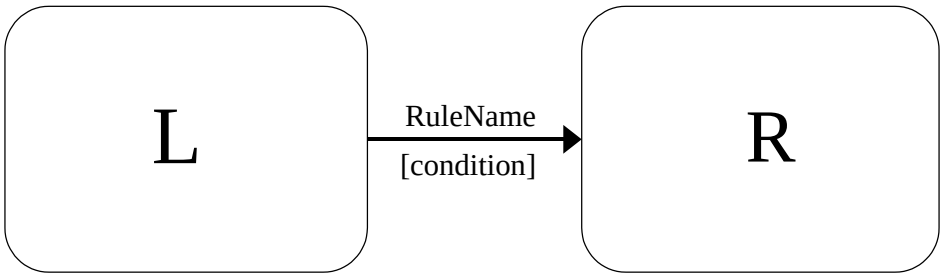


**Fig. 5. Basic OBGG rule.**

A rule can be applied whenever the left side of the rule is a sub-graph of the current system state graph. That is, whenever the graph described in the left side of the rule is currently present in the system state graph, the rule is able to be applied. Besides this, a rule can demand another condition to be applied. This condition is described as a logical expression presented below the transition arrow. Since OBGG is as an object-based formal language, the left side of a rule always must contain a message as a part of the graph. This message represents the message received by the entity that triggers the rule application. This defines that a rule in OBGG always handle a message. The condition associated to the rule is optional and can only consider values of the internal

attributes of the entity the rule belongs to and of the parameters of the message handled by the rule.

If the left side of a rule occurs and the condition is satisfied, the rule is said *enabled*. An *enabled rule* is a rule that can be selected for application. Many rules may be applied in *parallel*, as long as they do not have write access to the same items of the state (even the same rule may be applied in parallel to itself, using different matches). Two (or more) enabled rules are in *conflict* if their matches need write access to common items. If rules conflict, the choice of which one of them will be applied is non-deterministic.

As commented before, the application of a rule must consume a message (the message handled by the rule). A rule, when applied, may also modify internal attributes of the entity that received the message and generate new messages to other entities or to itself. This way, an application of a rule brings the system to a new state. The right side of a rule determines the graph that represents this new state, i.e., the graph that is the result of the modifications that occur in the system graph when the rule is applied. The modifications that can occur when a rule is applied are:

- Items in the left side that are not present in the right side of the rule are *removed*;
- Items in the right side that are not present in the left side are *created*;
- Items that are present in both sides of the rule are *preserved*.

The basic behavior of the proposed architecture can be described by the rules associated to the involved entities (the system behavior is composed by the behavior of their entities and their interactions). The active nodes are the center of the architecture, so they can be used as a starting point to explain how does the architecture work. The active node (AN) executes three functions: sending capsules in unicast mode, sending capsules in broadcast mode and providing services to received capsules.

### 4.1 Sending capsules in unicast mode

The first and simpler function of an AN is sending capsules in unicast mode. This function is activated when the AN receives a *SendMsg* message, containing the capsule to be sent and the identification of the destination of capsule, as presented in *SendMessage* rule depicted in Fig. 6.

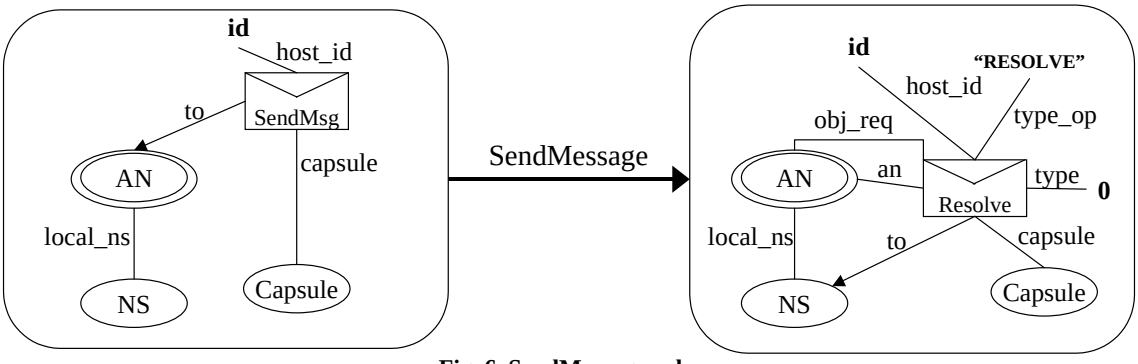


**Fig. 6. SendMessage rule.**

The AN then sends a request to its local naming server to get a reference to the destination node. When the local naming server returns the requested reference (*GetReference* rule in Fig. 7), the AN sends a message to the capsule informing that it must go to the indicated node.

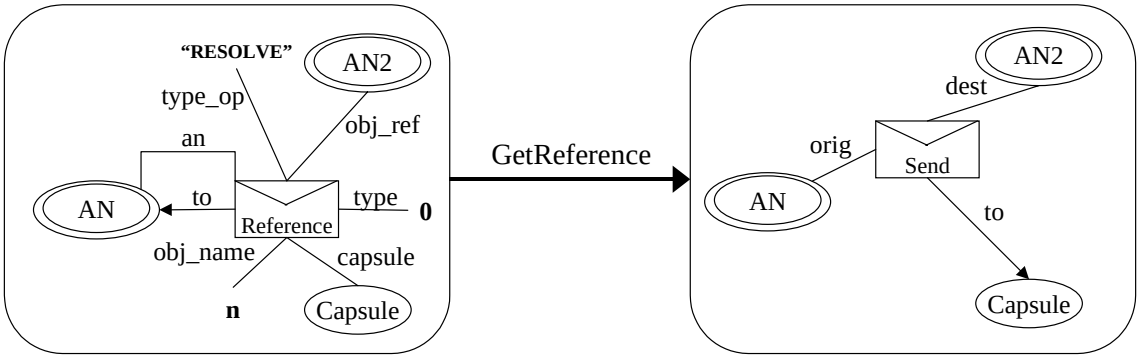


**Fig. 7. GetReference rule.**

The reception of the message sent by the AN, causes the start of the capsule moving process (*GoToHost* rule in Fig. 8), which takes its way to the destination node. Whenever a capsule receives a Move Message, it sends a message to its current location (AN where it is at) asking to move to the destination node. The AN then sends a message to the destination node informing that the capsule wants to be received by it. According to some requirements that must be attended to receive the capsule[1], the destination node sents back a message to the origin node informing whether it can receive the capsule or not. If the capsule can be received by the destination node, the origin node sends a message to the capsule informing it can move to the destination node. As soon as the capsules receives the accept of the destination node, it moves itself to its new location and change its internal information of current location, updating it to refer to the new node where it is at now. The origin node updates it internal state removing any reference to the capsule and the destination node updates its internal state adding the new capsule. After all these updates, the capsule continues its execution at the new node. The complete moving process specification is described in details in [10].

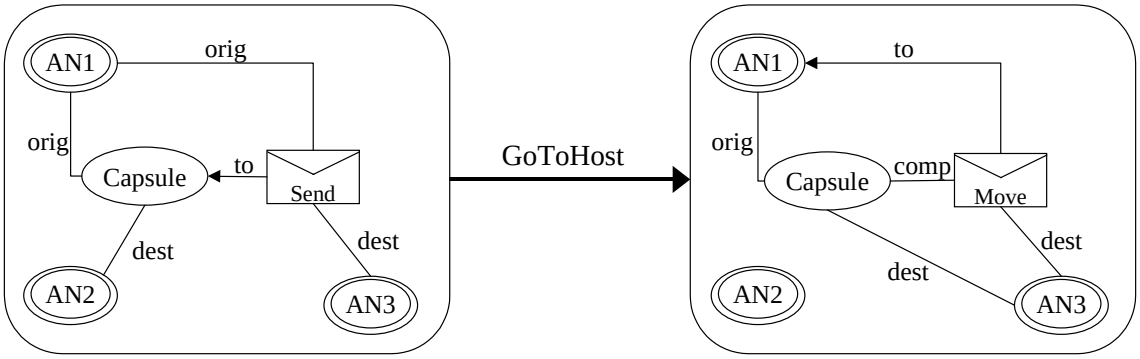


**Fig. 8. GoToHost rule.**

## 4.2 Sending capsules in broadcast mode

The second function of an AN is sending capsules in broadcast mode, i.e., sending copies of the same capsule to all known nodes. This function is activated when the AN

[1] These requirements are specified for each type of capsule and they may include processing power, storage capacity, etc. that are needed to support the capsule execution. If we consider the use of identifications due to security needs, the identification of the capsule should be considered appropriate and secure to the destination node.

receives a *SendBCast* message, containing the capsule that must be sent. Because of some restrictions imposed in this specifications, an AN can only send capsule in broadcast at a time. The attribute *inBroadCast* is used to control if the AN is currently executing a broadcast or not (see *SendBroadCast* rule in Fig. 9).

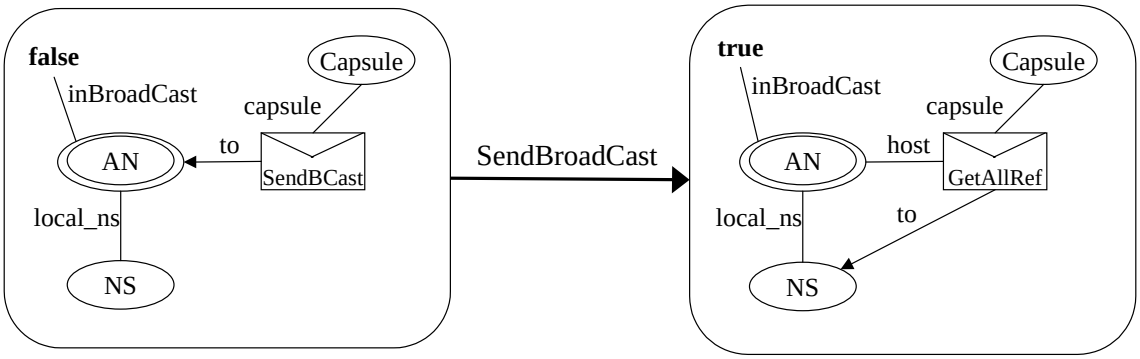


**Fig. 9. SendBroadCast rule.**

If the AN is not executing a broadcast when it receives a request to do it so, it asks the local naming server for a list of all known nodes. The list is returned by the naming server and is stored in the *names* attribute of the AN (see *StartBroadCast* rule in Fig. 10). The AN then sends itself a message (*StartBCast*) to start the broadcast.

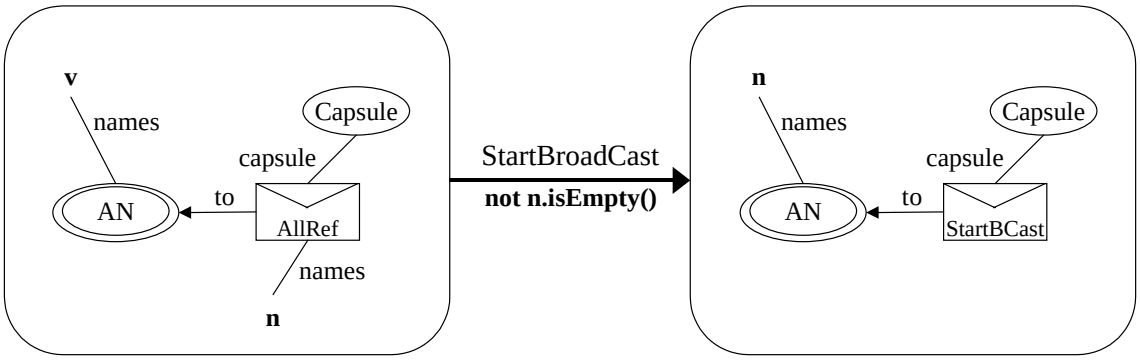


**Fig. 10. StartBroadCast rule.**

This message causes the activation of the *ResolveNextName* rule (Fig. 11). The AN removes the first identification of the list of known nodes and send it to the naming server requesting a reference to this node.

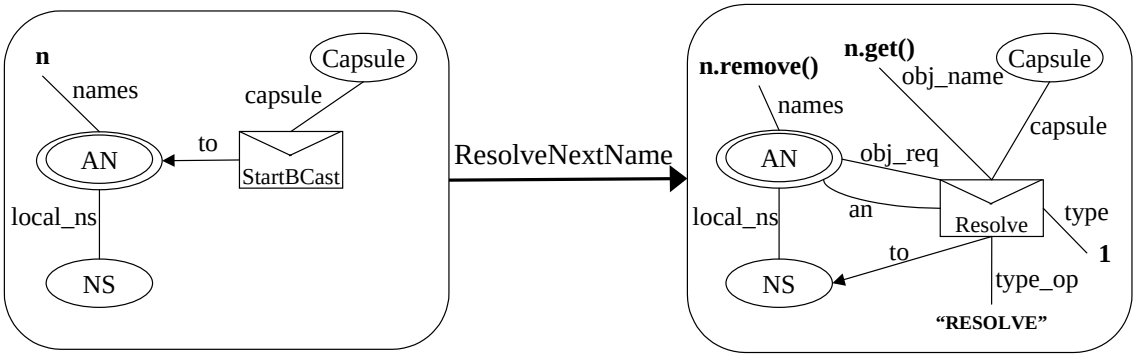


**Fig. 11. ResolveNextName rule.**

When the reference is received (*CloneCapsule* rule in Fig. 12), the AN send a message to the capsule that must be sent in broadcast, asking it to clone itself, and another message to itself to get the next identification of the list.

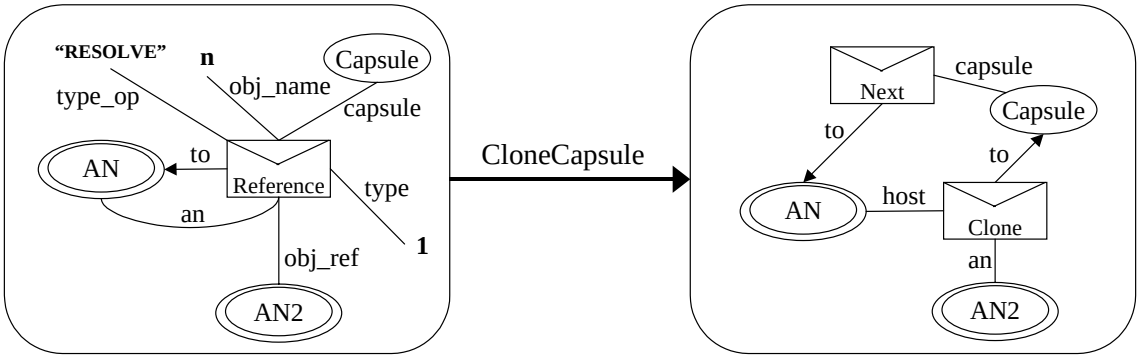


**Fig. 12. CloneCapsule rule.**

The duplication of the capsule is necessary to generate copies of it to be sent to each node in the list. The duplication of a capsule occurs according to the *CreateClone* rule scheme (Fig. 13). A *rule scheme* is like a template of a rule, presenting the basic components of the rule. This means that each specific application can add other components if necessary. That is, considering the case of *CreateClone* rule, the *Init* message used to initialize a capsule could have other parameters besides the *service_id* that would be useful to a given application.

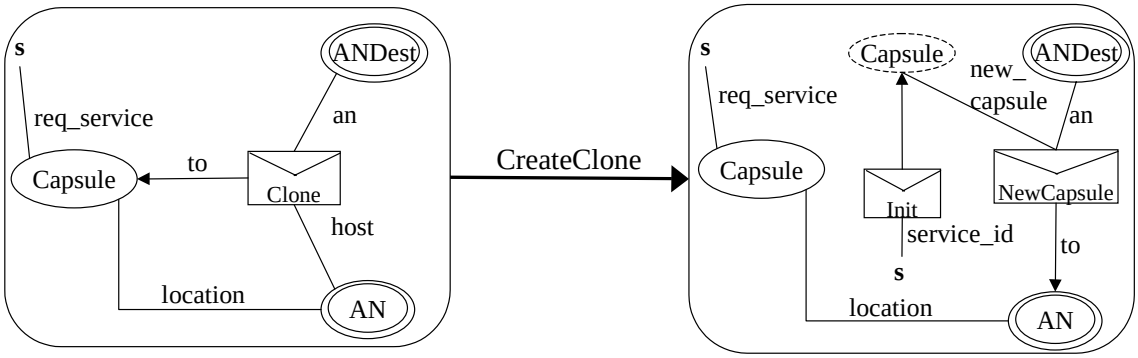


**Fig. 13. CreateClone rule scheme.**

Once the copy of the capsule is created, it is initialized with the *Init* message (Fig. 14) and a message containing a reference to the newly created capsule is sent to the AN that has requested the duplication.

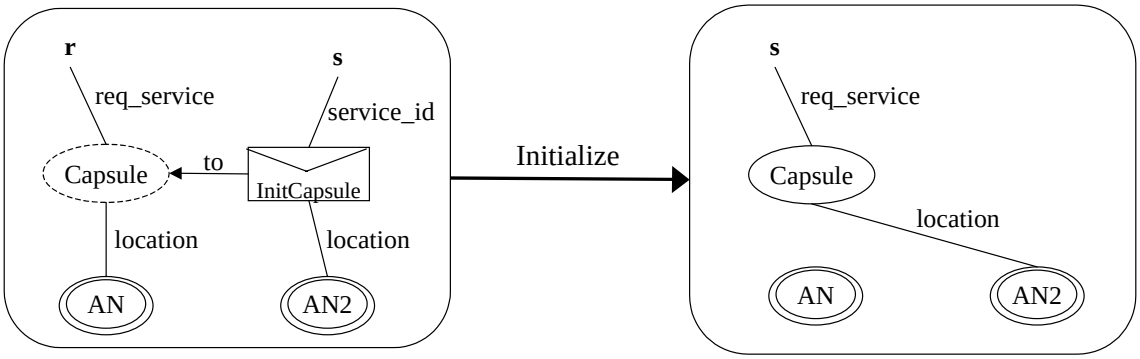


**Fig. 14. Initialize rule.**

As soon as the *NewCapsule* message is received by the AN, it sends a message to the capsule sending it to the destination node (*SendToNext* rule in Fig. 15). After this, the capsule moves to the destination as already commented in Section 4.1.

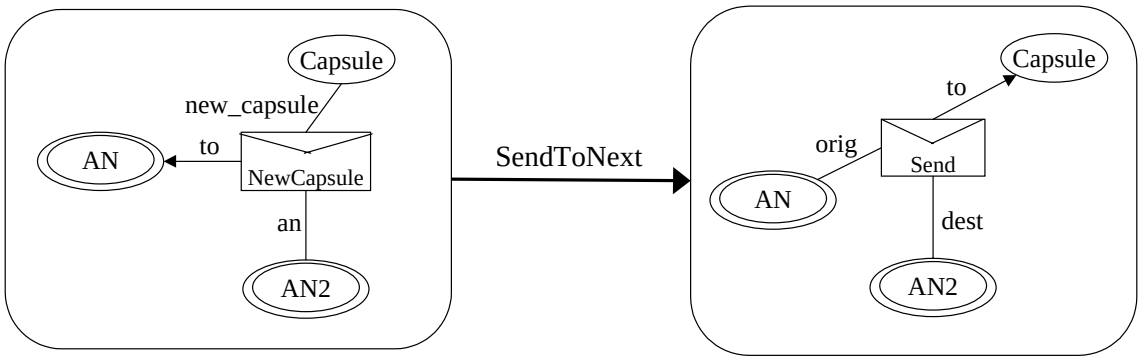


**Fig. 15. SendToNext rule.**

While the list is not empty, the AN repeat the described process to send copies of the capsule (*GetNext* rule in Fig. 16). When the list is empty, the broadcast is finished (*EndBCast* rule in Fig. 17).

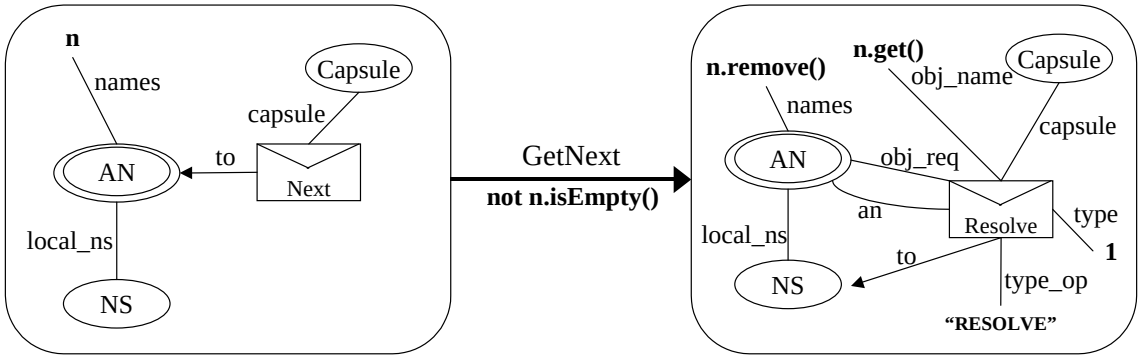


**Fig. 16. GetNext rule.**

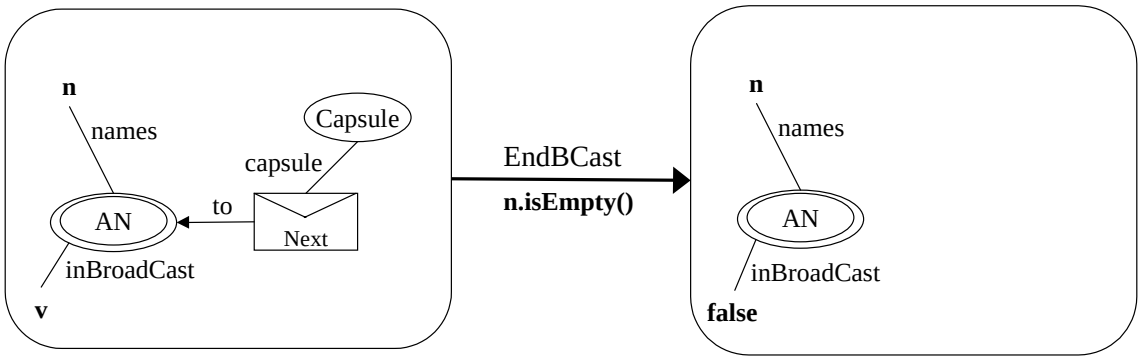


**Fig. 17. EndBCast rule.**

## 4.3 Providing service to a received capsule

After the sending of a capsule in unicast or broadcast mode, the basic behavior of the proposed architecture involves the reception of a capsule by an AN. When a capsule arrives at an AN, it requests the service necessary to handle it, as presented in *AskForService* rule in Fig. 18.

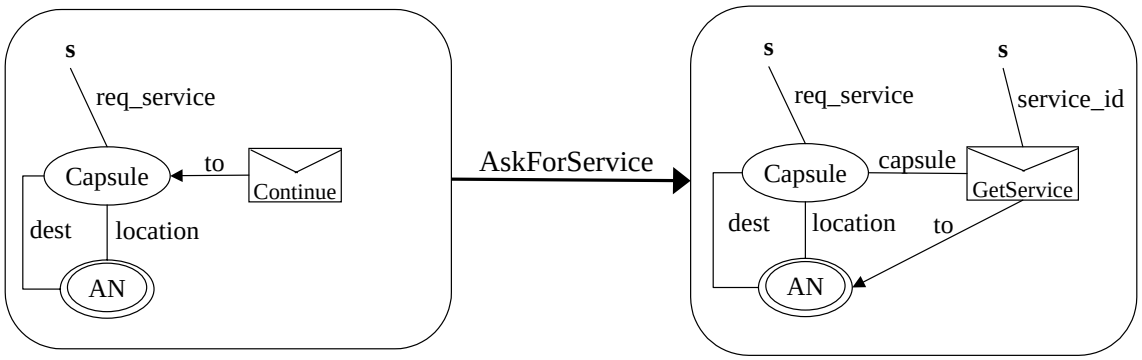


**Fig. 18. AskForService rule.**

In this situation, two things could happen: i) the AN currently owns an instance of the service needed to handle the received capsule; ii) the AN does not owns an instance of the required service and the capsule cannot be handled. In the first case, the AN simply provide the necessary service to the capsule obtaining a reference to the local instance of the service (*FindService* rule in Fig. 19) and forwarding this reference to the capsule (*ProvideService* rule in Fig. 20). Service and capsule can then interact directly. The way this interaction occurs is specific of each type of service and it is conceived together with its associated capsules. Fig. 21 presents a rule scheme to this interaction.

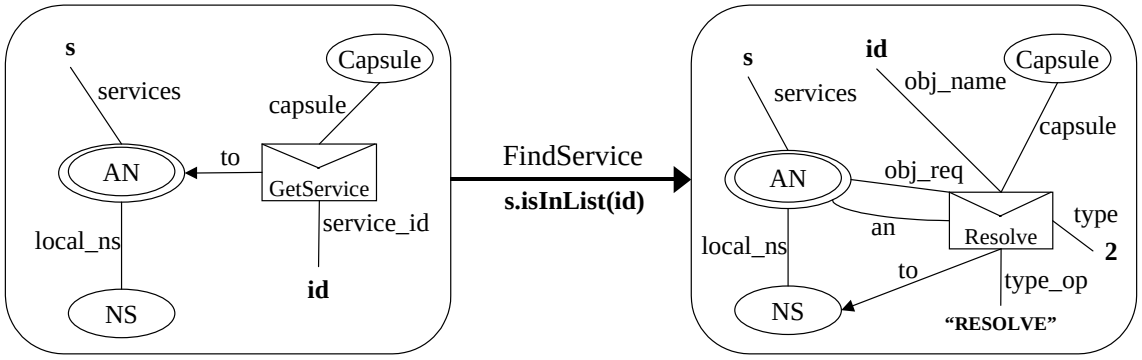


**Fig. 19. FindService rule.**

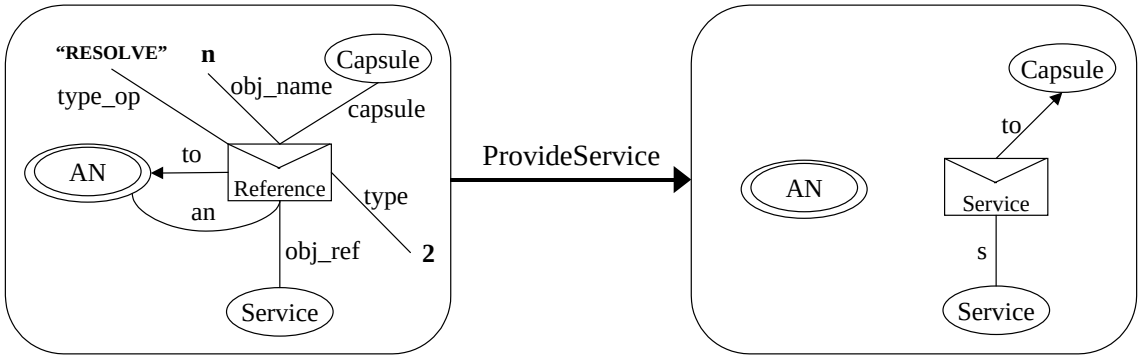


**Fig. 20. ProvideService rule.**

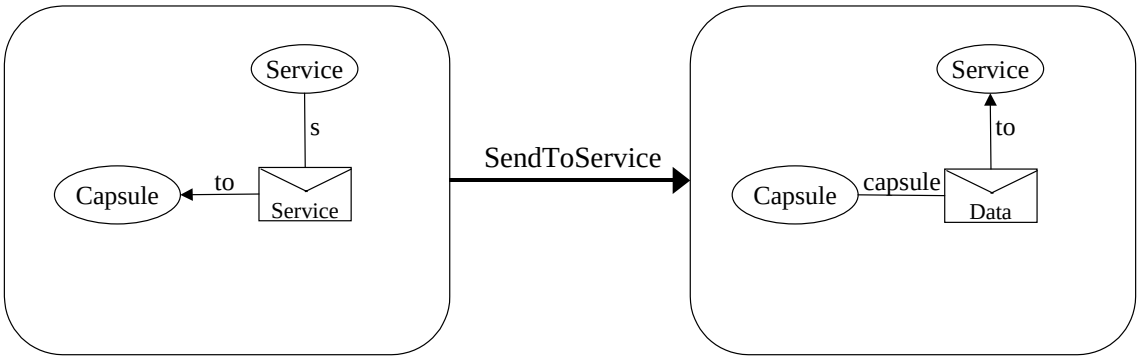


**Fig. 21. SendToService rule scheme.**

If the service cannot be provided, the AN must request an instance of the required service from a known code base (CB). The identification of the requested service is added to the pending list. This situation is illustrated in Fig. 22 in *RequestService* rule.

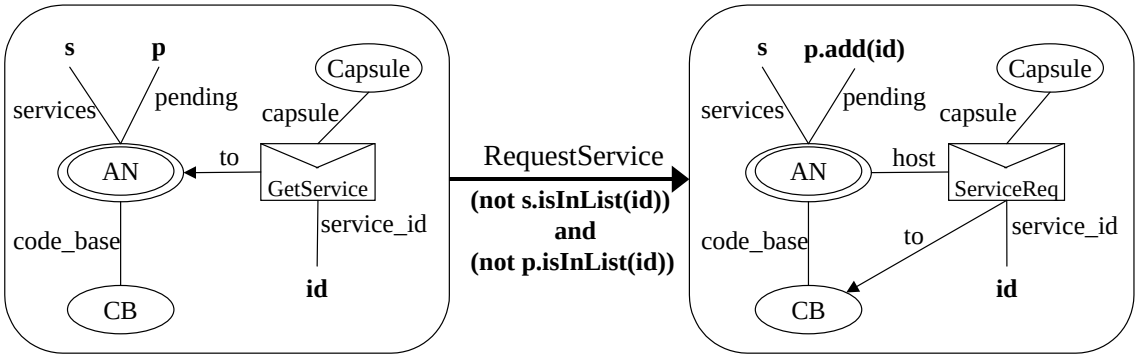


**Fig. 22. Request Service rule.**

The AN then verifies whether the required service is in its list of services or not. The *pending* attribute is a list of already requested services, i.e., a list of requests for services awaiting an answer from the CB. If the pending list contains the identification of the required service, the *GetService* message is ignored for a while and it will be handled once the instance of the service is installed. In the situation represented in *RequestService* rule (Fig. 22), the identification of the requested service is not in the list of services and is not in the pending list either. So, the AN forwards a service request to the known CB. In Fig. 23 we can see the rule applied whenever a CB receives a *ServiceReq* message from an AN.

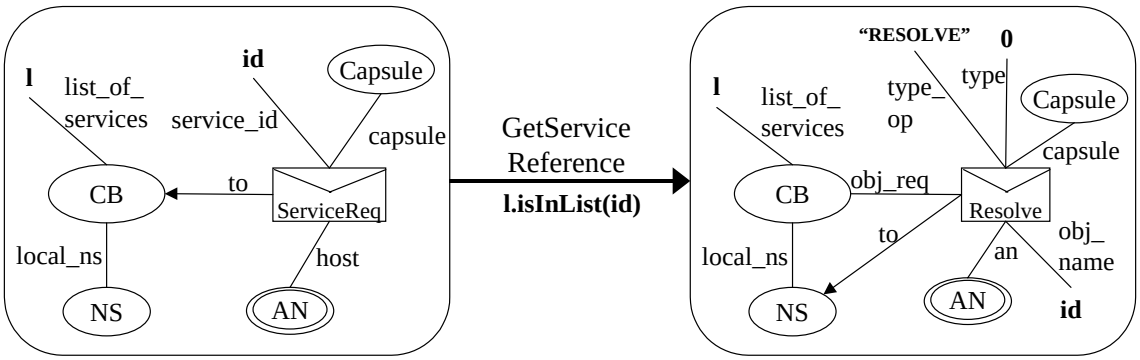


**Fig. 23. GetSeviceReference rule.**

To find the required service, the CB sends a message to its local naming server (NS) asking a reference to that service. The NS returns the reference to the service through a *Reference* message as presented in Fig. 24.

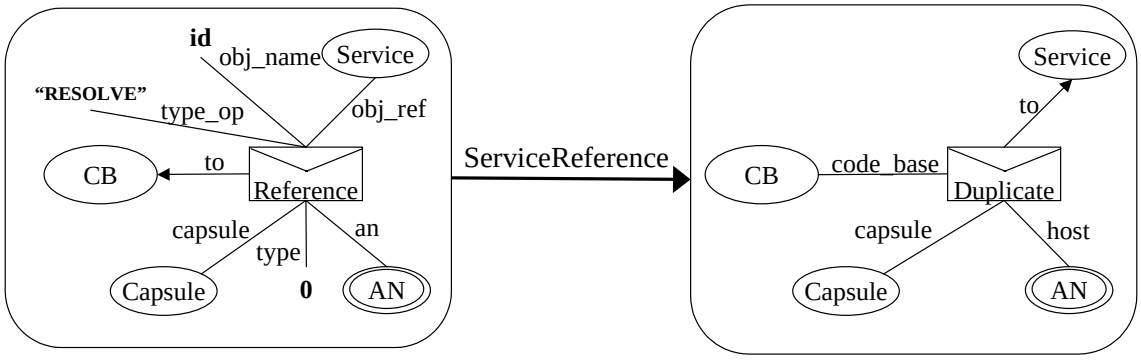


**Fig. 24. ServiceReference rule.**

When a reference to the service is obtained, the CB asks the service to duplicate itself. This way, the service copy can be sent to the AN and the original service can be saved for future requests. The service duplication is done creating a new instance of the service and copying the internal state of the original service to the copy (see Fig. 25). So, the copy has the same internal state of the original instance of the service.

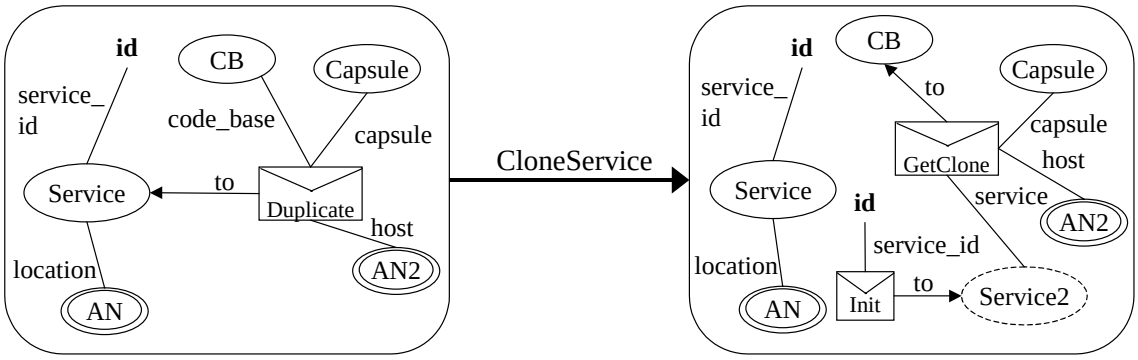


**Fig. 25. CloneService rule scheme.**

The service copy is returned to the CB through a *GetClone* message. After receiving the new service instance, the CB forwards it to the AN. This is illustrated in Fig. 26.

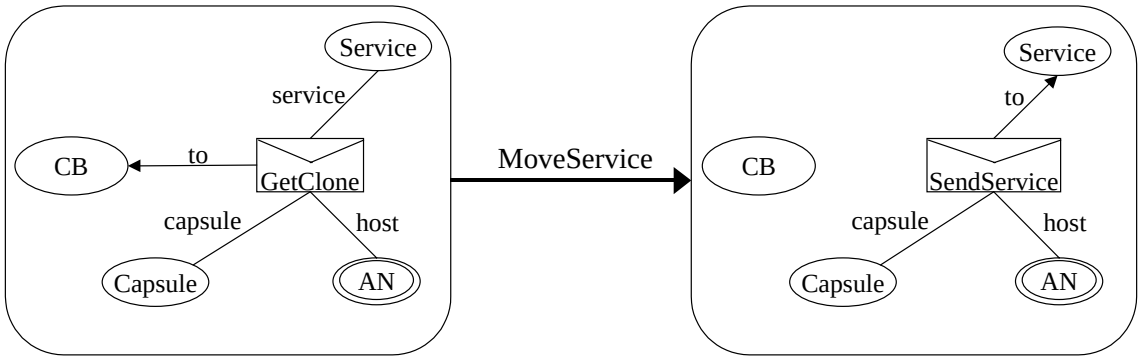


**Fig. 26. MoveService rule.**

The service instance then moves itself to the destination AN (Fig. 27).

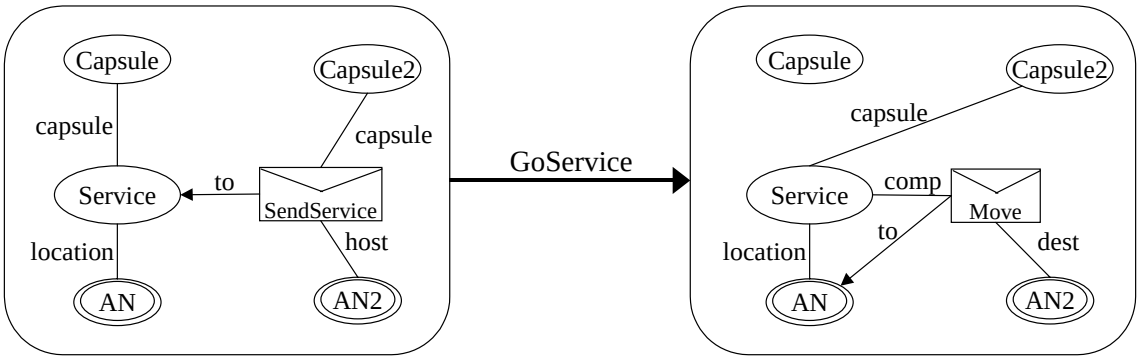


**Fig. 27. GoService rule.**

As soon as the service arrives at the AN, it requests its installation (Fig 28).

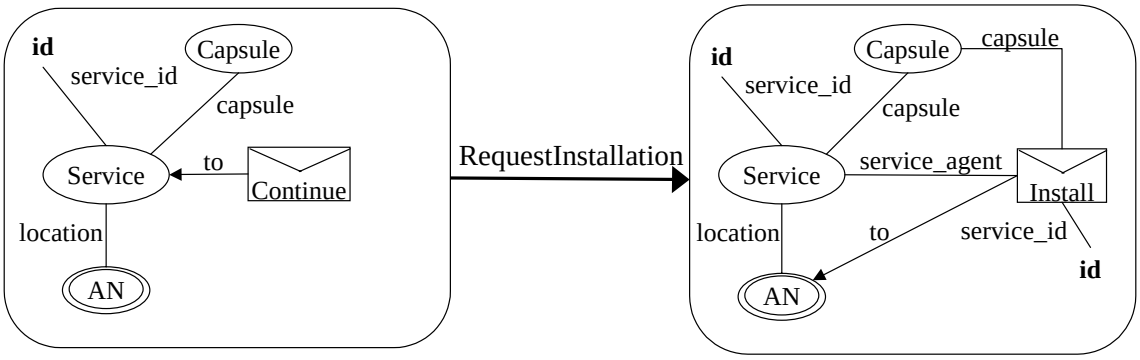


**Fig. 28. RequestInstallation rule.**

A service installation involves the addition of its the service identification to the list of services, meaning the AN can now provide this service, and the registration of the service in the local naming server, binding its service identification to the service instance (Fig. 29). The service identification is removed from the pending list and a reference to the service is sent to the capsule.

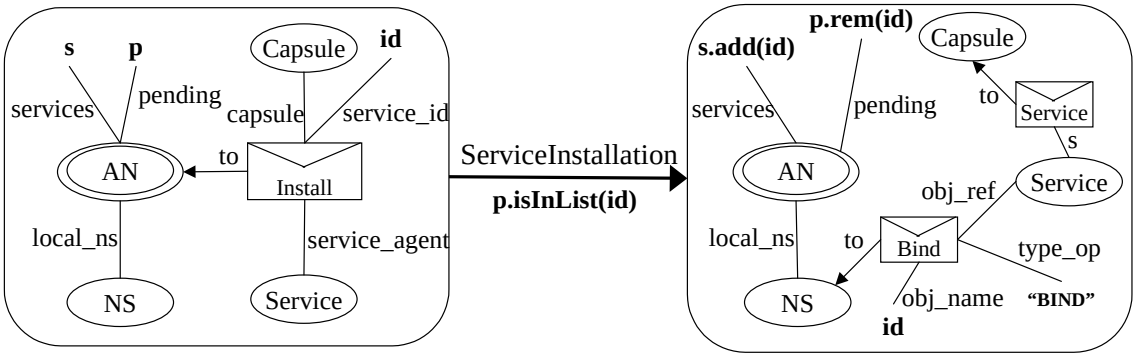


**Fig. 29. ServiceInstallation rule.**

From now on the capsule interacts with the service according to the specific application following the *SendToService* rule scheme already presented in Fig. 21.

## 5 Simulation and Execution of the Proposed Architecture

The simulator PLATUS [23] was developed using the Java language and it supports the simulation of models described in OBGG. This simulator works with

entities, which are the system components corresponding to the OBGG entities, and messages, that are the means of communication between entities. A special module called *kernel* is responsible for message delivery and for the global time control. The mapping from OBGG entities specifications to simulation entities is straightforward. Graphic tools are under development to allow the graphical creation of simulation models and their automatic conversion to the corresponding simulation code.

An entity is basically modeled by an active object (a Java object with an internal thread) where: attributes of the entity are mapped to attributes of the object; a message buffer of this object is used to store the messages delivered by the kernel; and rules are mapped to associated classes with functionality to: (i) test if a rule is enabled for an entity, and (ii) apply the transformations stated by the rule on the entity. The internal thread selects the rules enabled by messages in the input buffer and triggers their application, respecting conflict situations and the non-deterministic behavior given by Graph Grammars. The main advantage of using simulation is the possibility of testing the system behavior and finding errors in the conception of the system before the implementation.

In [10] a code generation scheme for applications specified in OBGG was proposed. The code is generated to execute over a mobility support platform, allowing the execution of the application in a real environment.

These tools developed in ForMOS project were used to test and execute the proposed architecture. Two testing scenarios were specified, simulated and executed. These testing scenarios are below presented.

### 5.1 Testing Scenario 1 – Providing service for a capsule

The first testing scenario considered a simple active network as presented in Fig. 30.

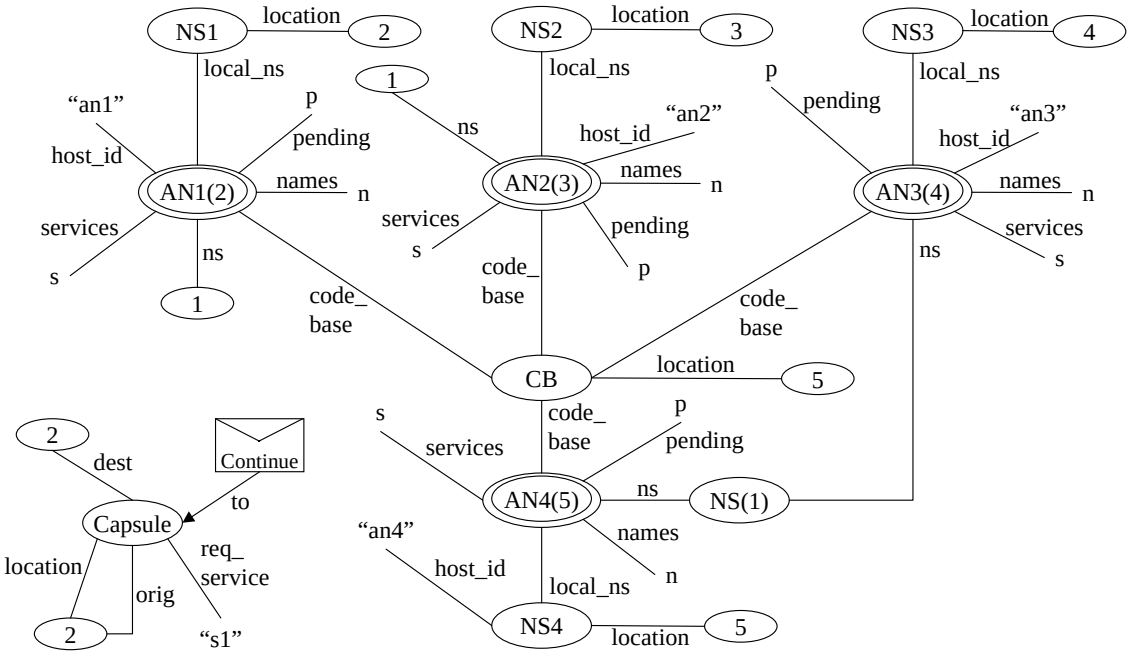


**Fig. 30. Scenario for testing the active network architecture.**

This scenario is presented as a simplified initial graph, showing 4 instances of AN, each one with its local naming server. In node *AN4*, there are the global naming

server, which stores information about the network (identifications of all AN and the references to available services), and the code base. The scenario also includes a capsule that passes through all the AN of the network and asks for the *S1* service. The *S1* service is initially not available in any AN. This way, when the capsule arrives at each AN it has to request the *S1* service from the code base, install it and provide it to the capsule. The simulation was useful to identify some errors in the specification of the architecture and correct them.

Once the simulation demonstrated that the behaviour was correct (the architecture behaves as desired), it was possible to generate code for this scenario using the scheme proposed in [10] and execute it over Voyager platform [24]. The tests were done using 4 different real nodes (machines), each one hosting an AN and a naming server. The capsule starts at *AN1* and follows its way through the other nodes until it got to *AN4*. The execution of the scenario showed that the proposed architecture works quite well, taking profit of the mobility of its components.

This scenario was small and simple, conceived only to test the architecture and analyze its behaviour.

## 5.2 Testing Scenario 2 – Routing in the active network

The second testing scenario was much more complex than the first one. This scenario involved the specification, simulation and execution of the *Dynamic Source Routing* (*DSR*) Algorithm over the active network architecture. The DSR [25] was developed to route packets between nodes of an *ad hoc* network. An *ad hoc* network [26] is a type of mobile network (nodes do not have physical connection with each other), where nodes are able to change information directly. Therefore, nodes are not connected with a base station, which controls the communication between them, differently of what occurs in the case of cellular phones. In an *ad hoc* network any node can work as a router, forwarding packets received from other nodes to their destination nodes.

The DSR is, as the name indicates, a *source routing* algorithm, i.e., the path through the network that a packet must follow from the origin to the destination node is determined before the packet is sent in the origin. In contrast to other routing algorithms, DSR does not generate periodical messages to update routing information. This means less traffic in the network and saving of battery energy of mobile devices. Although, this algorithm is recommended only to be used in small networks, containing from 5 to 10 nodes.

### *5.2.1 DSR Mechanisms*

The algorithm is composed by 2 mechanisms: the Route Discovery and the Route Maintenance.

#### *5.2.1.1 Route Discovery*

Route Discovery is the mechanism used by a node to dynamically discover a route to a destination node. It is used whenever a node does not know any route to the destination node of a packet. Discovered routes are stored in a *route cache (RC)*. Many routes to the same destination node can be cached.

According to the mechanism of Route Discovery, when a source node *S* wants to send a packet to a destination node *D*, it verifies the existence of route to *D* in its RC. If

there exists a cached route to *D*, the packet is sent following this path. Otherwise, node *S* starts the route discovery process broadcasting a *route request (RReq)* packet. This packet contains the identification of the source node, the identification of the target node (node to which a route is requested) and a list of visited nodes (initially, containing only the source node).

When a node receives an RReq, it follows the steps below presented:

1. If this node is the target of the request then: it generates a *route reply (RRep)* packet containing the list of nodes visited by the RReq, what composes the complete route from the source to the target node. The RRep is forwarded to the node that has originated the RReq. The RRep can be forwarded by three ways: through a route to the source node stored in the CR of the target node; through a route discovered by the target node performing a new route discovery process; or sending the response through the reverse path, according to the list of visited nodes present in the RReq;
2. If the nodes is not the target node then
   2.1 if the node identification is already in the list of visited nodes of the received RReq, the node ignores the request once it has already been received before;
   2.2. if the node identification is not in the list of visited nodes of the received RReq, the node verifies if it has a route to the target node in its CR
      2.2.1. if it has a route to the target node in its CR, it generates a RRep to the source node, adding the route found in the CR to the list of visited nodes of the received RReq, and sends the response to the source route;
      2.2.2. if it does not have a route to the target node in its CR, the node adds its identification to the list of visited nodes of the received RReq and broadcasts it to the next known nodes.

Each node also maintains a *recent requests list (RRL)* containing the identification of all requests recently received. This way, every time a request is received, the node verifies if it that request has already been received. If the request is in the RRL, the node does not forward it. This action prevents a flooding of repeated requests in the network.

When a node receives an RRep, it acts according to the following steps:

1. If the node is the destination of the response then it adds the route present in the RResp to its CR and sends its packets to the target node through this route;
2. If the node is not the destination of the response, it adds the route to be followed by the RRep from this node to the source node to its CR and uses it to forward the response to the next node in the route.

When a node receives a data packet and verifies it is the destination of the packet, it forwards the packet to the respective application. In the case that the node is not the destination one, it forwards the packet to the next node in the route of the data packet.

Packets awaiting a route to be sent through are stored in a buffer in the source node. Once a route is discovered, the packets destinated to the target node of this route are removed from the buffer and sent to their destination.

Timers control the CR and the RRL entries. The timers are used to determine the validity of an entry. This way, after some time, an entry is set invalid. In the case of a CR entry, it means that the route stored in the entry must be rediscovered. A timer also controls the validity of the packets waiting to be sent. This prevents the packets from waiting forever for a route.

#### *5.2.1.2 Route Maintenance*

Route Maintenance is the mechanism used to detect changes in the network topology. Changes in the network topology mean that some known routes become invalid or new routes are available.

All packets forwarded from a node to another must have its reception confirmed by the destination node. This confirmation is done sending an *acknowledgment packet*. Each data packet that must be sent or forward to another node is copied to a *retransmission buffer* in the source node. If no confirmation is received after a given interval of time, the source node retransmits the packet using the data present in the retransmission buffer. The packet retransmission is repeated a given number of times if no confirmation is received. Once the determined number of retransmission retries is achieved, the node considers that the link to the destination node is inactive. If this happens, the node generates a *route error packet (RErr).* The RErr is sent through the reverse route to inform the already visited nodes that this route is down. Each node that receives the RErr removes all routes in its CR that include the inactive link. When the source node receives the RErr, it can try an alternative known route to the same destination node or start a Route Discovery process to find another route.

As could be seen in the explanation of the DSR mechanisms, they are just used on demand, i.e., they are only initiated whenever it is necessary. Therefore, the network traffic caused by routing information is reduced.

### *5.2.2 Results*

The DSR algorithm was specified using OBGG formalism. The specified entities were the route request packet, the route reply packet, the route error packet, the data packet and the DSR service. The packets were mapped to capsules of the active network and the DSR service was mapped to an active network service. This way, the packets are transferred through the nodes of the network and each node provides the DSR service to handle this DSR packets. This specification was then mapped to simulation code.

The tests with the DSR algorithm working over the active network architecture were done using the logical topology presented in Fig. 31.

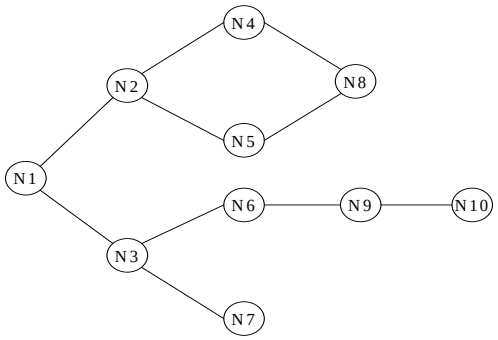


**Fig. 31. Topology used to test the DSR algorithm.**

The 10 nodes involved in the topology were active nodes. Besides these 10 nodes, a node *N0* was created to host the network code base. The node *N0* did not send or receive any DSR packets, so it was totally dedicated to the reception of requests to the code base. The node N0 also hosted the global naming server. This naming server owned the list of all services available in the network. As we were considering the execution of the DSR algorithm, the only available service was the DSR service. An instance of this service was created in node *N0* when the application was started. No other node initially owned this service, so when a node received a DSR packet, it should ask the code base for an instance of DSR service.

The tests were done using a data packets (capsules) that must be routed through the network from each node to other three nodes. The logical topology created allowed to test the main situations handled by the DSR algorithm, such as reception of repeated requests, reception of multiple routes to the same destination, treatment of route replies, forwarding of packets, among others. Another type of test involved the initialization of the CR of some nodes with some known routes. Hence, it was possible to test if the node would really forward a known route from its CR if it receives a request for this route. This reduced the time spent by a node to obtain a route to some destinations, once it was not necessary to wait the request to arrive at the destination node e return to the source node; the answer to the request was provided by a intermediate node.

The discussed tests were initially done using the simulator, allowing the improvement of the DSR algorithm specification through the correction of some errors encountered during simulation. After this phase, the specification was mapped to execution code through the mapping proposed in [10] and the real execution of the scenario occurred. It firstly involved the execution of all nodes in the same machine. Considering that the behavior of the execution was coherent to the specification and to the simulation, next tests were done executing the nodes distributed in 4 different machines and having multiple capsules to be routed. All capsules were successfully routed to their destinations and the nodes behaved as expected.

## 6 Final Remarks

This case study contributed to attest the validity of the tools developed in ForMOS project. The realization of this application using the support provided by the formal specification language, the simulator and the code generation mapping proved their usefulness.

The OBGG formalism has, as any other formal method, some deficiencies and restrictions that sometimes difficult its use. A characteristic that could be inserted in the formalism is a way to create an hierarchy of entities, i.e., to make it possible to use a previous specification of an entity and extend it with a new particular behavior, generating a new entity that extends the behavior of its predecessor. This way, it would be simpler to specify a DSR packet, for example, representing it as an extension to a capsule. At the formalism current state, this can only be represented rewritting the already specified behavior for a capsule to describe the behavior of a DSR packet, adding the other specific rules. In despite of this problem and others encountered during this case study, the formalism demonstrated to be intuitive, mainly because its graphical representation. Some patterns to this graphical representation have been followed to make it clearer, such as the definition of elements positions. An OBGG graphical editor is being developed to facilitate the use of this formalism.

The simulator demonstrated to be a very valuable tool. It contributed to certify the correctness of the specification. Some errors could be identified and corrected during the simulation phase. Such errors would not be easily found when executing the case study in a real environment. The simulator also provides a simulation time control that can be used to test the performance of a scenario. So, the latency of network links can be represented, for instance, allowing obtaining some kind of measurements.

The code generation is very useful once it is not easy to map a formal specification to code. The mapping provided in ForMOS is important not only because it facilitates the code generation for a specification in OBGG, but also because it has indicated to be coherent to the formalism. The coherency to the formalism means that the generated code assumes the same behaviour specified in OBGG. Efforts have been invested to create an automatic code generator. This code generator would be incorporated to the OBGG editor already commented and it would allow generating the code for a specification created using the editor.

Besides the contribution to the evaluation of the ForMOS tools, the case study also contributed to test the utilization of mobile components to compose an active network architecture and to implement a routing algorithm. The use of mobility gave the architecture the flexibility and the dynamics demanded by active networks. The architecture as conceived allows the easy incorporation of new services. This characteristic could be seen in the inclusion of the DSR service, since no modification was necessary to incorporate the capsules and the service needed to the execution of such routing algorithm. Multiple routing algorithms could execute simultaneously over the same architecture without the necessity of altering the architecture structure.

# 7 References


[1] FUGGETA, A., PICCO, G., VIGNA, G. Understanding Code Mobility, *Transactions On Software Engineering*, IEEE, vol. 24, 1998, pp. 342-361. Disponível em na Internet em http://swarm.cs.wustl.edu/~picco/listpub.html

[2] YEMINI, Y., SILVA, S. Towards Programmable Networks. In *Proc. IFIP/IEEE International Workshop on Distributed Systems, Operations and Management*, L'Aquila, Italy, 1996.

[3] WHITE, J. E. Telescript Technology: The Foundation For The Electronic Marketplace. *Technical Report General Magic, Inc.*, White Paper, 1994.

[4] MAGEDANZ, T., ECKARDT, T. Mobile Software Agents: A New Paradigm for Telecommunications Management. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*, Kyoto, Japan, 1996.

[5] SMITH, J. M. et al. *Activating Networks: A Progress Report*. IEEE, 1999.

[6] CAI, T., GLOOR, P., NOG, S. Dataflow: A Workflow Management System On The Web Using Transportable Agents. *Technical Report TR96-283, Dept. of Computer Science, Dartmouth College*, Hanover, NH, 1996.

[7] DOTTI, F. L., RIBEIRO, L. Specification of Mobile Code Systems Using Graph Grammars. *Formal Methods for Open Object-Based Distributed Systems IV*, Kluwer Academic Publishers, Stanford, USA, 2000. p. 45-63.

[8] DOTTI, F. L., DUARTE, L. M., SILVA, F. M. A., ANDRADE, A. M. S. A Framework for Supporting the Development of Correct Mobile Applications Based on Graph Grammars. *Submitted to the IDPT 2002*.

[9] EHRIG, H. Introduction to the Algebraic Theory of Graph Grammars. In *1st Graph Grammar Workshop, Lecture Notes in Computer Science 73 (V.Claus, H. Ehrig and G. Rozenberg eds.)*, Springer Verlag, 1979, p. 1-69.

[10] DUARTE, L. M. Desenvolvimento de Sistemas Distribuídos com Código Móvel a partir de Especificação Formal. *M.Sc. Thesis*, Programa de Pós-Graduação em Ciência da Computação, Faculdade de Informática, PUCRS, Brasil, 2002, 146 f.

[11] PSOUNIS, K. Active Networks: Applications, Security, Safety and Architectures. *IEEE Communications Surveys*, 1999.

[12] SCHWARTZ, B., ZHOU, W., JACKSON, A. W. *Smart Packets For Active Networks*. BBN Technologies, 1998.

[13] WETHERALL, D. J., TENNENHOUSE, D. L. The ACTIVE_IP Option. In *the 7th ACM SIGOPS European Workshop*, Connemara, Ireland, 1996.

[14] DECASPER, D., PLATTNER, B. DAN: Distributed Code Cashing For Active Networks. In *Proceedings of the IEEE INFOCOM'98*, San Francisco, USA, 1998.

[15] WETHERALL, D. J., GUTTAG, J. V., TENNENHOUSE, D. L. ANTS: A Toolkit For Building and Dynamically Deploying Network Protocol. In *Proc. IEEE OPENARCH'98*, 1998.

[16] GUNTER, C. A., NETTLES, S. M., SMITH, J. M. The SwitchWare Active Network Architecture. *IEEE Network, Special Issue on Active and Programmable Networks*, v. 12, n. 3, 1998.

[17] YEMINI, Y., SILVA, S. Towards Programmable Networks. In *Proc. IFIP/IEEE International Workshop on Distributed Systems, Operations and Management*, L'Aquila, Italy, 1996.

[18] YEMINI, Y., GOLDSZMIDT, YEMINI, S. Network Management by Delegation, *Integrated Network Management II*. I. Krishnan and W. Zimmer, Eds., North-Holland, 1991, p. 95-107.

[19] BHATTARCHARJEE, S., CALVERT, K. L., ZEGURA, E. W. On Active Networking and Congestion. *Technical Report GIT-CC-96/02*, 1996.

[20] LEHMAN, L. H., GARLAND, S. J., TENNENHOUSE, D. L. Active Reliable Multicast. In *Proc. IEEE INFOCOM'98*, San Francisco, CA, 1998.

[21] BHATTARCHARJEE, S., CALVERT, K. L., ZEGURA, E. W. Self-Organizing Wide-Area Network Caches, In Proc. IEEE INFOCOM'98, San Francisco, CA, 1998.

[22] RÖDEL, E. T. Especificação Formal de Aplicações Móveis: Estudo Comparativo e Métodos de Aplicação no Projeto ForMOS. *Trabalho Individual I*, Programa de Pós-Graduação em Ciência da Computação – Mestrado, PUCRS, 2001. 58 f.

[23] COPSTEIN, B., MÓRA, M. C., RIBEIRO, L. An Environment for Formal Modelling and Simulation of Control Systems. In *Proceedings of 33rd Annual Simulation Symposium*, SCS, 2000. p.74-82.

[24] OBJECTSPACE. *Voyager ORB 4.0 Developer Guide*. ObjectSpace, Inc. 2000.

[25] JOHSON, D. B., MALTZ, D. A., HU, Y., et al. The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (Internet Draft), *MANET Working Group, IETF*, 2001, 60 p.

[26] CÂMARA, D., LOUREIRO, A. A. F. Roteamento em Redes Móveis Ad Hoc. In *I Workshop de Comunicação Sem Fio, Minicurso 4*, Belo Horizonte, Brasil, 1999, p. 14-15.