P-SEAN: A Framework for Policy-based Server Election in Ad hoc Networks

Yacine M. Ghamri-Doudane*, Sidi-Mohammed Senouci**, and Nazim Agoulmine*
*Networks and Multimedia Systems Research Group, IIE and University of Evry joint Research Group
IIE, 18 Allée Jean Rostand, 91025 Evry Cedex – France.
ghamri@iie.cnam.fr, Nazim.agoulmine@iup.univ-evry.fr
** CORE / M2I / R2A R&D Unit, France Telecom R&D
2 Av. Pierre Marzin, 22307, Lannion, France.
sidimohammed.senouci@francetelecom.com

Abstract— **The client-server model is a commonly used model for distributed application programming. Most of group-collaboration applications, such as network gaming, rely on this model. We call a group-collaboration application an application where any participating entity can centralize the shared information and play the role of server. In wired networks, the choice of the participating entity playing the server role has a limited impact on the performances of the application, the station or the network. However, in mobile ad hoc networks (MANETs) an inadequate server choice can have a side effect on the network-, the application- or the wireless station-performances. The objective of our current work is to propose a novel and complete framework for server election and maintenance in ad hoc networks. This framework, called P-SEAN for Policy-based Server Election in Ad hoc Networks, uses two concepts in order to perform the server election process: the *Serving Ability Degree* and the *Situational Server-Election Policy*. Hence, the proposed framework implements the different situations for server election and maintenance in ad hoc networks as policy-rules (*Situational Server-Election Policy*). This election and maintenance are mainly based on factors such as connectivity, processing power, RAM capacity, remaining battery life, etc. These factors define the *Serving Ability Degree* of each ad hoc node. The motivation behind using the Policy-based Networking paradigm is to render our framework extensible to incorporate additional application-specific criteria. In addition to these two main concepts, we also propose a complete architecture for a P-SEAN-enabled service and a lightweight protocol for *Server Election and Maintenance Exchanges*.**

Keywords— server election, ad hoc networks, client-server, group-collaboration applications, policy-based networking, Ponder, COPS.

I. INTRODUCTION

Mobile Ad hoc NETworks (MANETs) [1] are self-organized and autonomous networks that have the potential to provide wireless and mobile computing capabilities in situation where efficient, economical and rapid deployment of communication is required, and where the use of a wired or an infrastructure-based wireless network is either too expensive or impractical. These networks are characterized by dynamic topologies, bandwidth-constrained variable-capacity links, and limited survivability. Furthermore, ad hoc nodes are battery-operating nodes having various ranges of processing power and storage capacities.

The client-server model is used by a significant part of nowadays network applications. For some of these applications, each participating entity can either play the role of client or server. This is the case for network gaming applications for instance. Indeed, in this kind of distributed applications, one of the participating entities is chosen to centralize the shared information and distributing these to the other entities that take part to the distributed application. Hence the former runs a server and serves all the other entities (clients). We call this kind of distributed applications group-collaboration applications.

The client-server model is also widely used in ad hoc networks. Indeed, in addition to usual group-collaboration applications such as network gaming, several distributed-management and control applications proposed in the literature uses this model. Hence, for instance, different proposal for partition detection/forecasting [2,3] relies on a central server. Another example is the instantiation of the Policy-based Management architecture in ad hoc networks proposed in [4]. This instantiation uses also centralized policy-servers. The assumption pursued by all these schemes is that all ad hoc nodes are capable of running the server functionality. They also suggest that this server should be chosen regarding to the performances of the network, the nodes or the application. However, these schemes do not propose any solution for realizing such appropriate server election.

Due to the performance problems that can arise from a bad server choice, it is clear that a more accurate policy for server election, than the simple 'first arriving node policy', is necessary. This policy should be based on criteria such the connectivity, the node processing power, its remaining battery life, etc. According to these criteria, the server task will be delegated to the node that has the best characteristics to assume this role. As far as we know, there are no proposals in the literature for server election approaches based on joint application/node/network performances. Our aim in this work is to propose a complete and extensible framework that allows realising the server election and maintenance processes based on situational and application-specific criteria.

In order to realise the server election process the proposed framework bases its operations on two main concepts: the *Serving Ability Degree* and the *Situational Server-Election Policy*. Hence in our framework, each node will be assigned with a *Serving Ability Degree* (SAD) that will be calculated using predefined criteria. Mainly, the SAD calculation is based on the node performances and its location in the network. Once the SAD calculated, a set of algorithms, dealing with the various situations that may happen during the lifetime of the ad hoc network, are applied in order to implement the server election process. These algorithms form the *Situational Server-Election Policy*. Indeed, each elected server has a lifetime that is dependant on the situations that may emerge in the ad hoc network: the degradation of the server characteristics (materialized by an important decrease of its SAD), the sudden disappearance of the server (server falling down for instance), the necessity of server replication as the server is leaving the network (the proximity of server departure, network partition detection/forecasting …) or as a better node in the network may host the server functionality, conflict resolution in presence of contending servers…. Hence each elected server has a lifetime in the ad hoc network after which it has to be replaced. In order to make the server election process extensible allowing the incorporation of application-specific criteria (in addition to the numerical criteria taken into account by the SAD calculation), we suggest to use the Policy-based Networking (PBN) paradigm [5]. Using PBN, the *Situational Server-Election Policy* is specified as a set of policy-rules. This is realised through the use of the Ponder policy specification language [6,7]. In addition to the definition of the *Serving Ability Degree* and the *Situational Server-Election Policy* concepts, we propose a complete architecture for a P-SEAN-enabled service. This architecture includes the two concepts presented above and uses a lightweight protocol for handling the communications related to server election and maintenance (*Server Election and Maintenance Exchanges*). This latter is designed as an extension to the Common Open Policy Service (COPS) protocol [8]. Hence, a complete and extensible framework for Policy-based Server Election in Ad hoc Networks (P-SEAN) is proposed. Note that the choice of the Ponder policy specification language and the COPS protocol is mainly motivated by their flexibility and extensibility features.

The rest of the paper is organized as follows: Section 2 introduces the Policy-based Networking paradigm; it also presents the Ponder policy specification language and the COPS protocol. The SAD calculation within each terminal is discussed in section 3 followed by a detailed description of the *Situational Server-Election Policy*. Section 5 presents the P-SEAN-enabled service architecture and the used lightweight protocol for handling the *Server Election and Maintenance Exchanges*. Finally, Section 6 concludes the paper and presents future work.

II. POLICY-BASED NETWORKING AND PONDER

The aim of the Policy-based Networking technique is to allow the integrated management of all network, system or application components throughout a same management system. This latter will then allow applying a global management strategy (the policy) to all concerned components. Such technique can be used for network, system or application management. We then suggest using this technique for managing the server election in group-collaboration applications that base their operations on the client-server model. Hence, situational and application-specific rules can be introduced and used together in order to elect, maintain and re-elect the most suitable server at a given instant. The use of Policy-based Networking can also allow to easily extending the scope of the application management beyond the server election process. In this latter case P-SEAN can both participate and take benefits from largest management architecture.

In order to specify the policy-rules involved in the *Situational Server-Election Policy*, we used the Ponder policy specification language. For its part, the communication between the entities evolved in the group-collaboration application is handled throughout a Lightweight Protocol. This protocol is implemented as an extension to the COPS protocol. Then, in the following subsections, we present both Ponder and COPS. We also motivate their choice regarding to the P-SEAN objectives.

A. Ponder: A Language for Policy Specification

Ponder is a declarative object-oriented language that have been designed in order to allow the specification of management policies for distributed-object systems. Initially proposed for the specification of security policies, Ponder appeared as a flexible and extensible language. It is extensible as it can also be used for the management of any discipline other than security. Hence, in addition to security policies, realised throughout the **authorisation**, **delegation**, **information filtering** and **refrain** policies [6,7], it allows the definition of generic policy-rules called **obligations** in Ponder.

The obligation policy is a typical '***If*** *Condition* ***Then*** *Action*' policy-rule. The *Condition* part is a Boolean expression that can either return *true* or *false* when evaluated. In fact, in Ponder this part is composed of two entities: the *constraints* and the *events*. In the following we briefly present the syntax of the **obligations**, **constraints** and **events** concentrating on the language elements that we use to specify the *Situational Server-Election Policy*. For more details on the Ponder policy specification language, please refer to the original references [6,7].

```
inst oblig      ruleName  "{"
on                        event-specification ;
subject         [<type>]  domain-Scope-Expression ;
do                        obligation-action-list ;
[when           constraint-expression ;          ] "}"

inst event      eventName = eventExpression ;

inst constraint constraintName = predicate ;
```

Figure 1. The syntaxe of Ponder events, constraints and obligation-policy.

As shown in Figure 1, **obligations** (**inst oblig**) specify the actions that have to be **do**ne **on** the occurrence of a set of **events** and **when** a set of **constraints** are verified. The actions are targeting a specific domain scope defined as the **subject** of the **obligation**. The **events** are elements that become *true* instantaneously, eventually allowing the change of state of the

Condition part of the **obligation** policy. We can say that the **events** trigger the **obligations** when reacting to a state change within the managed entity.

Note that, under the keyword **event** (cf. Figure 1), the Ponder language allows defining complex events that are a combination of simple events. This is realized in order to simplify their reuse in several **obligations**. The combination of events is realized through the use of the Boolean operators. The Boolean combination is also used for the definition of complex **constraints** for reuse purposes. In the following we will also use these two keywords for clarity purposes defining events and constraints with clear denominations.

In our work, we suggest to use Ponder to implement the *Situational Server-Election Policy* due to its flexibility. Furthermore, thanks to the Ponder extensibility features, the server election process could easily be extended to incorporate any other type of policy-rules and specifically those related to application-specific criteria. Hence, if required by the application, security policies can easily be introduced in the server-election process for instance.

B. COPS Protocol

The IETF[1] Resource Allocation Protocol (RAP) Working Group has specified a scalable and secure framework for policy definition and administration [8]. This framework introduces a set of components to enable policy-rules definition, saving and enforcing: the Policy Decision Point (PDP), the Policy Enforcement Point (PEP), and the Policy Repository (Figure 2.a). PEP components are policy decision enforcers located in network and system equipments. The PDP is the component responsible for high-level decision-making process. This process consists of retrieving and interpreting policies, and implementing the decision in the network through the set of PEPs. The Policy Repository contains policy-rules that are used by the PDP. In order to exchange policy information and/or decisions, the PDP interacts with each PEP using one of the several protocols specified or extended for this purpose. Among them, the Common Open Policy Service (COPS) protocol [8] is the one which was designed specifically by the IETF to realize this interaction.

COPS [8] is a lightweight client/server protocol allowing the exchange of policy information between a PDP and its PEPs. This exchange is realized through six main messages as described in Figure 2.b. The client open (OPN), client accept (CAT) and client close (CC) messages allows respectively to manage the connection initiation, acceptance and termination between the PEP and the PDP. After a connection establishment between the PEP and its serving PDP, the PEP transmits requests for decisions to the PDP using the REQ message. In response to a REQ, a decision message (DEC) is sent by the PDP. Then, the PEP reports the outcomes to the PDP via the RPT message. The *Keep Alive* (KA) message is used in order to allow monitoring the server/connection health. To do so, KA messages are periodically exchanged between the PEP and the PDP elements. Hence, if one of these elements does not receive KA message from its correspondent for a certain time interval, the PEP discovers the disappearance of the PDP and *vice versa*.

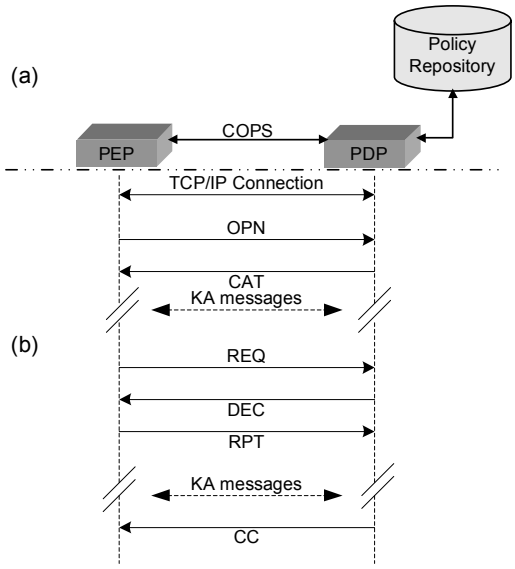

Figure 2. Policy-based Management: (a) the IETF framework, and (b) the COPS protocol operations.

As in the P-SEAN framework the *Situational Server-Election Policy* is implemented through the use of policy-rules, we suggest using the IETF's framework to implement P-SEAN. Note however that this is not our lone motivation. Indeed, as will be described in details below (cf. Sections IV and V.A), the node hosting the server in P-SEAN-enabled services is in charge of taking policy-decisions concerning the server lifecycle. These decisions have to be enforced by the other participating nodes (clients). These nodes have also to periodically send application-management information (typically their SAD) towards the server. Remind that the SAD is an evolving parameter as its value may depend on time-evolving parameters (connectivity, remaining battery life …). Thus, as policy-decisions and management information has to be exchanged between the server and its clients, a possible candidate protocol to realize these exchanges could be a specific extension of the COPS protocol. Thus, the management plan of the server part in P-SEAN-enabled services encompasses a PDP while the clients run PEP elements in their management plan. As we will show later (cf. Section V.B), the flow of message exchange defined by the COPS protocol (Figure 2.b) can be easily extended in order to allow incorporating the specific messages related to the *Server Election and Maintenance Exchanges*.

III. SAD: SERVING ABILITY DEGREE

As previously stated, the main characteristic of ad hoc networks is that they do not rely on any predefined infrastructure. In such networks, due to the mobility of nodes it may happen that some nodes become temporarily unreachable. Furthermore, the nodes forming the ad hoc network may be heterogeneous in terms of capacities and battery life. If we consider that a set of nodes within the ad hoc network have to take part to a group-collaboration application and that one of

[1] IETF: Internet Engineering Task Force

them has to centralize certain tasks and information, it would be more judicious to choose the "most suitable" station to play this role rather than doing a random choice. In order to determine the "most suitable" station, a *Serving Ability Degree* is associated to each station. This *Serving Ability Degree* is not a static parameter. This parameter should take into account the different factors that can impact application-, node- or network-performances. It determines the ability of each node to host the server functionality.

The *Serving Ability Degree* can either take into account static factors, such as the processing power and node's Random Access Memory (RAM) capacity, or moving-in-time factors, such as battery life or network topology. However, depending on the tasks that the server may accomplish and/or on the generated network overhead that can be induced if the server is far from its clients, these factors do not have the same importance in the server election. Hence, the formulations "more adequate" and "most suitable" are application specific. They depend on different factors and with different degrees. For this reason, we do not propose a single formula to calculate the *Serving Ability Degree* but a guideline towards doing this calculation. Following this guidelines, we also give an example illustrating the SAD calculation for a particular application.

A. SAD Calculation: Principal

There are several factors that are susceptible to have an impact on the performances of the application, the network or the node hosting the server. Among these, we can cite: node-dependant factors such as the processing power, remaining battery life or RAM capacity, and connectivity-dependant factors such as the node position in the network or the link-capacities. The performance impact of these factors depends on the application behavior. For instance, the connectivity-dependant factors have an important performance impact for the applications generating important data exchanges between the server and its clients. Furthermore, these factors do not have the same impact degree. Some of them have greater impact than others. Hence, for applications generating important data exchanges between the server and its clients, the remaining battery life is a more important factor to deal with in the server election than the processing power for instance.

From this, we can say that the *Serving Ability Degree* calculation depends on the application behavior. It is application specific. For this reason we propose a generic SAD calculation process. One can also note that the list of factors presented above is not exhaustive. Indeed, depending on the application behavior other factors can have a side effect on the performances of the application, the network or the node hosting the server. Proposing a generic SAD-calculation process has also the advantage of being extensible to include other factors.

If (all necessary conditions are verified) ***then***
{
/* SAD calculation */

$$SAD = \frac{1}{\sum_{i=1}^{N} a_i} \sum_{i=1}^{N} a_i f_i(x);$$

/* x being the crisp value, f_i the Fuzzification function, N the number of considered factors, and a_i the weight corresponding the i^{th} factor*/
}
Else /* Case where the node can not run the server
SAD = 0;

Figure 3. Serving Ability Degree calculation algorithm.

From the application point of view, the performance factors can be classified onto two categories:

- Necessary conditions: this category encompasses the criteria that are essential to the ad hoc node in order to run the server functionality. Such necessary conditions are minimal processing power, minimal RAM capacity, or multi-task capabilities for instance. Thus, if one of these conditions is not met, the terminal is declared not eligible to host the server functionality. In this case, the SAD value corresponding to this terminal will be set to the null value (cf. the ***else*** part in Figure 3).
- Performance aspects: this category allows differentiating between eligible nodes (i.e. those that satisfy all necessary conditions). Performance aspects rely mainly on the node-dependant and connectivity-dependant factors presented above. As these factors are heterogeneous, we suggest using a method similar to the Fuzzification procedure used in Fuzzy Logic [9] in order to calculate the SAD value. The Fuzzification is a procedure where a membership function is applied in order to translate crisp input values to values belonging to the same space, the fuzzy space. Once the Fuzzification procedure applied and the crisp values brought to the 'fuzzy' space, a normalized weighted addition of the obtained 'fuzzy' values is realized in order to compute the SAD value (cf. the ***if*** part in Figure 3). Note that the weights correspond to the performance-impact degree of the corresponding factors. Hence, there are defined to match the impact, of hosting the server functionality, on the application-, the node-, and the network-performances. As this impact depends on the application behavior, these weights are application specific. Let us also remind, that the word Fuzzification here relates only to the transformation of the crisp values into values belonging to the same space and not to the whole process involved by Fuzzy logic.

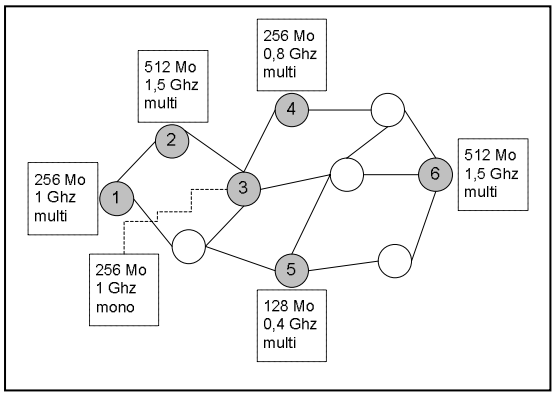

Figure 4. The considered ad hoc network.

B. SAD Calculation: an Example

In order to illustrate the SAD calculation in ad hoc networks, let us consider the example shown in Figure 4. The concerned ad hoc network is composed of 10 nodes. We assume that only the gray-numbered nodes are willing to participate to the group-collaboration application. The remaining nodes participate only to the connectivity. In our example, we assume that the ad hoc network uses a proactive routing protocol [1]. This allows each node to know its distance (in number of hops) with all the other nodes in the ad hoc network. The triplets (RAM capacity, processing power, multi- or mono-task system) characterizing each ad hoc node are also depicted in Figure 4.

The factors that we consider in this example are the following:

- Necessary conditions: Multi-task machine, RAM capacity ≥ 256Mo, and processing power ≥ 1Ghz.
- Performance criteria: In addition to the processing power and the RAM capacity, the node hosting the server may preferably have a central position in the network. The central position degree of a node is given by the mean distance between this node and all the other nodes participating to the group-collaboration application.

The weights assigned to each of these criteria are: $a_1 = 1$ for the RAM memory, $a_2 = 4$ for the processing power, and $a_3 = 2$ for the topological criterion. One can note that the processing power is the most important criterion followed by the topological criterion and the RAM capacity respectively. The Fuzzification functions $f_i(x)$ used in our example are illustrated by Figure 5.

From the necessary conditions, the eligible nodes are nodes 1, 2 and 6. The SAD values calculation for these nodes gives:

- SAD(1) = 1*0.1+4*0.1+2*0.748 = 1.996
- SAD(2) = 1*0.4+4*0.55+2*0.82 = 4.24
- SAD(6) = 1*0.4+4*0.55+2*0.712 = 4.032

From this, we can remark that the topological criterion is the one that allow differentiating nodes 2 and 6.

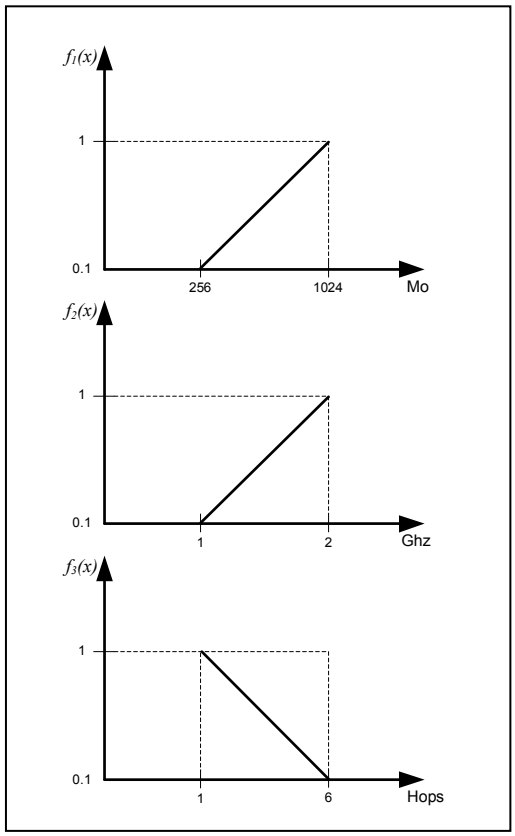

Figure 5. The fuzzification functions $f_i(x)$ used in our example.

IV. SITUATIONAL SERVER-ELECTION POLICY

In addition to the calculation of the *Serving Ability Degree* for each node, the distributed server election process must take into account the dynamic feature of ad hoc networks. By distributed server election process we mean server election, maintenance and replication when required. This entire process, called *Situational Sever Election Policy*, relies on the different situations that may occur in an ad hoc network. These situations include:

- Node arrival and departure.
- Server replication due to network partitioning, node departure/disappearance or when a node with a better SAD exists in the network.
- Conflict resolution in presence of contending servers.

In order to implement the *Situational Sever Election Policy*, we suggest using the Policy-based Networking paradigm. The motivation behind this is twofold: simplicity and extensibility. Indeed, as we will show in the following subsections the *Situational Server-Election Policy* can be easily implemented using policy-rules. Furthermore, the use of the Policy-based Networking paradigm can allow the simple extension of our proposal to incorporate supplemental application-specific policy-rules in the server election process.

In the following subsections, we describe the different situations that may occur in an ad hoc network and their impact on the server election process. The resolution of each of these

situations is discussed and the corresponding policy-rules are then specified using Ponder.

A. New Node Arrival

At the application launch, the terminal calculates its SAD and then tries to detect the presence of a server. Server search can be realized throughout a service discovery protocol such as SDP (Service Discovery Protocol) or Salutation [10,11]. The server search process lead to one of the following situations:

- There is no server currently running in the network; however a server is starting by another node (***ServerStarted*** = ***false***, ***ServerStarting*** = ***true***).
- There is no server currently running in the network and no one is trying to start the server (***ServerStarted*** = ***false***, ***ServerStarting*** = ***false***).
- A server already exists in the network (***ServerStarted*** = ***true***).

These different situations are handled by the three policy-rules depicted in Figure 6.

When starting a P-SEAN-enabled service, an ad hoc node updates the state of the (***NewArrival(s)***) event. This update is realised throughout the setting of the ***node.arrival.instant*** temporal variable. Hence a node is considered to be a just-arrived-node when its ***node.arrival.instant*** is recent (i.e. during a certain amount of time after ***node.arrival.instant*** initialization). Following this update, the ***NewArrival(s)*** event triggers the three different policy-rules that allow handling the situations identified above. The first policy-rule (***NodeJustArrived1***) corresponds to the case where there is neither a server started nor starting for the corresponding application. This policy-rule allows a recently connected ad hoc node to start the server functionality (***s.StartServer()***). Note also that before launching the server, the corresponding node has to check that the value associated to its SAD is not null. Once the server launch started, the node immediately set the ***s.ServerPossible()*** variable to ***true***. Subsequently, none of the other ad hoc nodes will start the server functionality when beginning the same P-SEAN-enabled service. After server launching, the state of the ***s.ServerPossible()*** and the ***s.ServerExist()*** variables are switched to ***false*** and ***true***, respectively.

The second triggered policy-rule (***NodeJustArrived2***) corresponds to the case where a server, corresponding to the P-SEAN-enabled service, is currently starting (***ServerStarting*** = ***true***). In this case, the application is switched to the sleep mode (***Time.wait(SleepPeriod)***) for a certain period chosen in accordance to the duration of the corresponding server starting procedure. Then, it reinitialises the server search procedure (***s.arrival.instant = time.now()***). Indeed, this latter action re-trigger the three policy rules depicted in Figure 6. In the case where the server search procedure is successful, i.e. there is a server currently running in the network, the just-arrived-node connects to the discovered server and sends it its SAD. This behaviour is described by the third triggered policy-rule (***NodeJustArrived3***).

```
Events :

  event NewArrival (node) = (node.Arrival.instant () == time.now()) ;

Constraints :

  constraint ServerStarting = ((ServiceDiscovery().ServerExist == false)
             && (ServiceDiscovery().ServerPossible = = true)) ;
  constraint ServerStarted = (ServiceDiscovery().ServerExist == true) ;

Policy-rules :

  inst oblig NodeJustArrived1 {
  on         NewArrival(s) ;
  subject    s =me ;
  do         (s.StartServer() || s.ServerPossible() = true)
             -> (s.ServerExiste() = true || s.ServerPossible() = false);
  when       (s.SAD()<>0 && !ServerStarted && !ServerStarting);
  }

  inst oblig NodeJustArrived2 {
  on         NewArrival(s) ;
  subject    s =me ;
  do         Time.wait(SleepPeriod) -> s.Arrival.instant = time.now() ;
  when       (s.SAD()<>0 && ServerStarting) ;
  }

  inst oblig NodeJustArrived3 {
  on         NewArrival(s) ;
  subject    s =me ;
  do         s.ArrivalTo(ServiceDiscovery().server)
             -> s.sendToServer(s.SAD()) ;
  when       (ServerStarted) ;
  }
```

Figure 6. Policy-rules executed at the client side by a node recently joining the application.

Finally, note that these three policy-rules can not generate conflicting actions. Indeed, they can not be enforced at the same time. In fact, the constraints, allowing confirming the enforcement of the action-part of these policy-rules, can not be verified at the same time. Note also that the constraint parts of these policy-rules are complementary allowing handling all possible cases.

B. Lifecycle of Elected Server

The server management procedure has to deal with multiple situations during the server lifecycle. These situations correspond to the following cases:

- The server has to be replaced as the node hosting the server is living the network, closing the application or as one of the other nodes participating to the P-SEAN-enabled service has better characteristics to host the server functionality (i.e. a higher SAD value).
- The server suddenly disappeared (server falling down).
- Two servers are running in the network (conflict). This situation may happen in the case of the merge of two ad hoc networks on which the same P-SEAN-enabled service is running.

The following sub-sections present the policy-rules allowing handling these situations.

1) Server Departure

There are two cases after which the server functionality has to be moved from one node to another: (i) the case where the node hosting this functionality is on the point of leaving the

network[2] or closing the P-SEAN-enabled service, and (ii) the case where another node participating to the distributed application has better characteristics to host the server functionality (i.e. a higher SAD value).

- In the former case, the server functionality should be moved before the application or the network is left. This operation is however not possible if all the remaining participants have a null SAD value.
- In the latter case, there is a risk that several server-switching occur. Indeed, each time a participant has a better SAD, a server switching may happen. Thus, at start-up for instance, an important number of participants may join the P-SEAN-enabled service with somewhat small temporal shifts. As these nodes have different capabilities, it may happen that several server-switching occur during this particular period. Furthermore, as nodes can have their SAD evolving during time[3], it may happen that the SAD evolution makes the node having the best-SAD also evolves during time. Then, in order to control the number of possible server switching, we make this procedure possible only in predefined periods. We also prohibit server switching if the difference between the two concerned SAD is below a certain threshold. Hence, periodically, the server checks the SAD of each registered node. And for the best one, it first checks that the ratio between the best-SAD and its SAD is above a carefully chosen threshold. If yes, a server switching is triggered.

```
Events :

   event NodeIsLeaving(me) = (PartitionPredict() || me.Logout().begin) ;
   event ServerIsNotBest = (Best.SAD()/ me.SAD() > SADThresh) ;

Constraints :

   constraint SubstitutExist = (Best.SAD <> Null ) ;
   constraint IamServer = (ServiceDiscovery().server.adress() = =
            my.adress()) ;

Policy-rules :

   inst oblig ServerReplacementToBeDone1 {
   on         NodeIsLeaving(s) ;
   subject    s =me ;
   do         s.sendTo(BestSubstitute(),Cmd(StartServer()))
              -> StartCmd.sent =true ;
   when       (IamServer && SubstitutExist) ;
   }

   inst oblig ServerReplacementToBeDone2 {
   on         ServerIsNotBest ;
   subject    s =me ;
   do         s.sendTo(BestSubstitute(),Cmd(StartServer()))
              -> (StartCmd.sent =true ||
              StartCmd.waitReport = true) ;
   when       (IamServer) ;
   }
```

Figure 7. Policy-rules executed by the server when it initiates the switching procedure.

[2] The node can be sensed as being on the point of leaving the network if it is starting the turn off procedure or when network partition is detected/forecasted. This latter can be sensed through the use of one of the existing server-based partition detection/prediction methods [2,3] for instance.

[3] The calculation of the SAD can imply temporally evolving parameters such as remaining battery life, connectivity constraints …

Policy-rules described in Figure 7 represent these operations. Hence, the server has to be moved either when the actual node hosting the server is not the best one (***ServerIsNotBest***), or when this one is leaving the network or the application (***NodeIsLeaving(s)***). In this situation the server is moved towards its best substitute (***s.sendTo(BestSubstitute(),Cmd(StartServer())***) if this one exists (***SubstitutExist***). Note that, the server-move implies the sending of the start server command and the replication of all information needed by the started server in order to operate properly. Afterwards, the closing server updates the corresponding state variables '***StartCmd.sent = true***' and '***StartCmd.waitReport = true'***. Thus, the actual node hosting the server puts itself on standby waiting for a positive report from the newly started server (***RapportOK(BestSubstitute()***). Upon the reception of this report (cf. Figure 8), the leaving server send a message to its clients in order to redirect them towards the started server (***ServerReplacementDone***). In the case where the positive report is not received, the server will only send a message towards its clients informing them about its destiny (***ServerReplacementNotDone***).

```
Events :

   event ShouldLeave(me) = (PartitionImminence() || me.Logout().fin) ;
   event ReportOK(node) = (node.Report.reveived() &&
         (node.Report.status() = = "OK")) ;

Constraints :

   constraint IamServer = (ServiceDiscovery().server.adress() = =
            my.adress()) ;
   constraint SentStart = (StartCmd.sent = = true) ;
   constraint WaitRepor = (StartCmd.WaitReport = = true) ;

Policy-rules :

   inst oblig ServerReplacementDone {
   on                RapportOK(BestSubstitute()) ;
   subject    s =me ;
   do         (s.sendTo(All(),s.ServRedirct(BestSubstitute())||
              WaitReport = false) -> s.sendTo(All(),s.ClientClose()) ;
   when       (IamServer && SentStart) ;
   }

   inst oblig ServerReplacementNotDone {
   on                NodeShouldLeave(s) ;
   subject    s =me ;
   do         s.sendTo(All(),s.ClientClose()) ;
   when       (IamServer && SentStart && WaitReport) ;
   }
```

Figure 8. Policy-rules executed by the server when it concludes the switching procedure.

For their part (cf. Figure 9), when a node not hosting the server (***!IamServer***) is one the point of leaving the application or the network (***NodeIsLeaving(s)***), it inform the server of its intent (***s.sendTo(Server,s.ClientClose())***). Once triggered by the corresponding (***ClientClose***) event, the server removes the closed client from its substitute table, if this one is registered within this table, (***s.SubstituteTable.remove(CC.from())***).

```
Events :

   event NodeIsLeaving(me) = (PartitionPredict() || me.Logout().begin) ;
   event ClientClose = (CC.reveived() && (IamServer)) ;

Constraints :

   constraint IamServer = (ServiceDiscovery().server.adress() = =
            my.adress()) ;

Policy-rules :

   inst oblig DeconnexionClientReq {
   on              NodeIsLeaving(s) ;
   subject    s =me ;
   do         s.sendTo(Server,s.ClientClose()) ;
   when       (!IamServer) ;
   }

   inst oblig DeconnexionClientResp {
   on              ClientClose ;
   subject    s =me ;
   do         s.SubstituteTable.remove(CC.from()) ;
   when       (IamServer) ;
   }
```

Figure 9. Policy-rules executed at the client side when this one is ready to disconnect from the application.

Upon the reception of the start command (***ReceivedStart***), the targeted node must answer it by the affirmative (cf. Figure 10). It then sends a positive report to the server '***s.sendTo(Server,Report("OK"))***' and starts the server launching procedure as described in Figure 6. The other clients, will then connect to the newly elected server by sending it their SAD (***s.connectTo(server)-> s.sendTo(server,s.SAD())***). Thus, the server replacement is done.

```
Events :

   constraint ServerStarting = ((ServiceDiscovery().ServerExist== false)
            && (ServiceDiscovery().ServerPossible == true)) ;

Constraints :

   event ReceivedServRedirect = (ServRedirect.reveived()) ;
   event ReceivedStart = (StartCmd.reveived() ) ;

Policy-rules :

   inst oblig AcceptStartingServer {
   on         ReceivedStart ;
   subject    s =me ;
   do         s.sendTo(Server,Report("OK")) -> (s.ServerPossible() = true
              || s.StartServer()) -> s.ServerExist() = true;
   }

   inst oblig ConnectToNewServer {
   on         ReceivedServRedirect ;
   subject    s =me ;
   do         s.connectTo(ReceivedServRedirect.server)->
              s.sendTo(server,s.SAD()) ;
   }
```

Figure 10. Policy-rules executed in order to make effective the server replacement

2) Server Disappearance

This case represents the case where the server disappears suddenly (the node hosting the server fall down for instance). In order to handle such situation, the clients monitor the server throughout the use of the periodically-sent *Keep Alive* messages (cf. Section II.B). Hence, when one of the clients detects the server disappearance as it does not received several *Keep Alive* messages; it triggers the node arrival procedure described in Section V.A. This triggering is realized throughout the re-initialization of the ***s.arrival.instant*** variable. The policy-rules depicted in Figure 11 describe this case.

```
Events :

   constraint ServRedirectNotReceived = (!ServRedirect.reveived()) ;

Constraints :

   event ServerClose = (CC.reveived() && (CC.from() == Server.adress())) ;

Policy-rules :

   inst oblig ReinitiateServerElectionProcedure {
   on         ServerClose ;
   subject    s =me ;
    do        s.arrival.instant = time.now() ;
   when       ServRedirectNotReceived) ;
   }
```

Figure 11. Policy-rules triggered by each client at the event of server disappearance.

3) Server-Election Conflict-Resolution

In this section, we are concerned by the case where two or more servers are present in the ad hoc network for serving the clients of a same P-SEAN-enabled service. In this case, only one server has to be maintained, the others have to be closed. As for the election, the server that should be maintained is the one having the largest SAD value. Server conflicts may happen whilst two or several ad hoc networks, each of which containing a server, are merged. Note that, conflict detection is realised throughout periodic triggering of the server search procedure (Service Discovery) by 'each' server.

Hence, when a server detects the existence of another server, it sends it its SAD. Upon the reception of the SAD originating from another server, the concurrent server interprets this as a server close order. It will apply this order only in the case where its SAD is smaller than the received one. In the case where its SAD is larger than the received one, it replies back to its concurrent server by sending its SAD. Then, the server who has the smallest SAD will be closed. These situations are described by the ***ServerTwin2*** and ***ServerTwin3*** policy-rules of Figure 12. The ***ServerTwin1*** policy-rule describes how the concurrent-server detection is handled.

ServerTwin4 and ***ServerTwin5*** policy-rules (Figure 12) describe for their part the case where the SAD values of both concurrent-servers are equals. In this case the arising question is "*which of the concurrent servers must be stopped?*" This situation is qualified as a contention situation. To resolve this contention situation, we suggest using a *Binary Exponential Backoff* (*BEB*) procedure. This procedure is currently used in 802.11 networks for contention resolution between concurrent frames aiming to access the shared wireless medium at the same time [12]. Hence, in our studied situation, when a contending server discovers that its SAD is equal to the SAD of its concurrent-server in the networks, it randomly chooses a timer value between 0 and CW. It then waits until this timer is reached. When this timer is reached, the server sends a server-close order (***s.sendCloseServer(CloseServerReceived()***) to its concurrent-server. The concurrent server receiving this order has then to close. It may happen that both concurrent servers send the server close order approximately at the same time. In this case, the contention is not resolved. Then, for each of the remaining concurrent servers, the CW value is doubled, a new

timer value is randomly chosen, and the contention resolution procedure is triggered again (***CW=CW*2 -> Time.wait(rand()*CW)***). This procedure is repeated until one of the concurrent servers closes (cf. ***ServerTwin4*** and ***ServerTwin5*** policy-rules).

Events :

event *AnotherServer = (IamServer && ServiceDiscovery().server.adress() <> my.adress());*
event *ConcurantServer= (CloseServerReceived().status == true) ;*

Constraints :

constraint *Sent*SAD *= (*SAD*.sent = = true) ;*

Policy-rules :

inst oblig *ServerTwin1 {*
on *AnotherServer ;*
subject *s =me ;*
do *s.sendCloseServer(ServiceDiscovery().server.adress(), s.*SAD*()) ->* SAD*.sent = true ;*
}

inst oblig *ServerTwin2 {*
on *ConcurantServer ;*
subject *s =me ;*
do *s.sendTo(CloseServerReceived().server, 'ClosingS'))-> s.sendTo(All(),s.ServRedirct(BestSubstitute())) -> s.sendTo(All(),s.ClientClose()) ;*
when *(IamServer && CloseServerReceived().*SAD *> s.*SAD*()) ;*
}

inst oblig *ServerTwin3 {*
on *ConcurantServer ;*
subject *s =me ;*
do *s.sendCloseServer(CloseServerReceived().server,s.*SAD*()) ->* SAD*.sent = true ;*
when *(IamServer && CloseServerReceived().*SAD *< s.*SAD*()) ;*
}

inst oblig *ServerTwin4 {*
on *ConcurantServer ;*
subject *s =me ;*
do *s.sendTo(CloseServerReceived().server, 'ClosingS'))-> s.sendTo(All(),s.ServRedirct(BestSubstitute())) -> s.sendTo(All(),s.ClientClose()) -> CW =10 ;*
when *(IamServer && CloseServerReceived().*SAD *== s.*SAD*() && !Sent*SAD*) ;*
}

inst oblig *ServerTwin5 {*
on *ConcurantServer ;*
subject *s =me ;*
do SAD*.sent = false -> CW=CW*2 -> Time.wait(rand()*CW)-> s.sendCloseServer(CloseServerReceived().server,s.*SAD*()) ->* SAD*.sent = true ;*
when *(IamServer && CloseServerReceived().*SAD *== s.*SAD*() && Sent*SAD*) ;*
}

Figure 12. Policy-rules for server-election conflict-resolution.

Note that, before leaving, each server may transfer required state information to the lastly elected server to enable it to operate properly.

Note also that the proposed policy-rules treat the case where only two servers are competing in the network. Indeed, as a situation where more than two servers are present in the ad hoc network at the same time is not a common situation, we suggest that the conflict-detecting server treat them sequentially as long as this one is re-elected. If it is not re-elected, the newly re-elected server will be responsible for conflict detection and resolution.

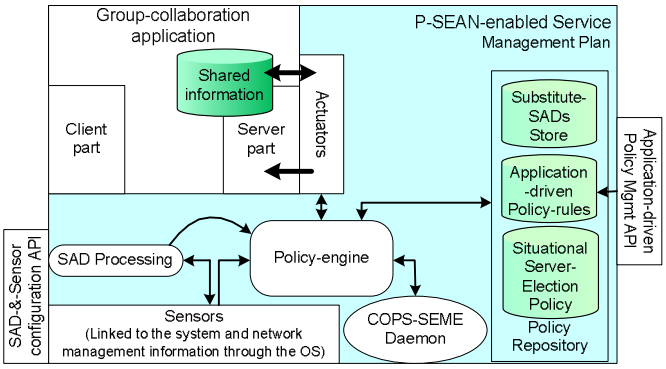

Figure 13. P-SEAN-enabled service architecture.

V. POLICY-BASED SERVER ELECTION IN AD HOC NETWORKS: ARCHITECTURE AND COPS-SEME PROTOCOL

After having presented in details the two main concepts behind the P-SEAN framework, namely the *Serving Ability Degree* and the *Situational Server-Election Policy*, let us now discuss how these two concepts could be introduced in group-collaboration applications. Indeed, a P-SEAN-enabled service includes both the group-collaboration application and its management plan (Figure 13). The management plan is designed and implemented in such a way that it is transparent to the group-collaboration application operations. It is also reusable for any other group-collaboration application.

In the following subsections, a detailed description of both the P-SEAN-enabled Service Architecture and the COPS extension for *Server Election and Maintenance Exchanges* (SEME) is given.

A. P-SEAN-enabled Service Architecture

From what we described earlier, P-SEAN is based on two main concepts: The *Situational Server-Election Policy* described in Section IV and the *Serving Ability Degree* described in Section III. The former is not application-dependent and is then unchanging. The latter is however application-driven as we stated above. Indeed, depending on the targeted service, this one would use specific sensors that have to be included in the management plan. These sensors are implemented as a set of plug-ins retrieving management information (system characteristics) from the system and/or monitoring (routing tables, remaining battery capacity …) it. These sensors are triggered by the SAD-processing module first at start-up and then periodically. When triggered, they returns or compute the targeted value (remaining battery life, mean distance to the other nodes participating to the P-SEAN-enabled service …). The SAD-processing module is a generic configurable module. Indeed, after retrieving the numerical values returned by the corresponding sensors, it applies the configured weights and the Fuzzification functions on them in order to compute the SAD value. It then triggers the policy-engine periodically in order to update the SAD value. This latter will then be compared to all the SAD values stored within the Policy Repository. Note that the Substitute-SADs Store is also periodically updated. Hence, all the SAD values of all the disconnected clients are removed. Also, all the connected clients send periodically their SAD using COPS-

SEME. This allows updating the SAD values for connected clients. Note that the COPS SEME daemon allows a P-SEAN server to receive management information from its clients (typically SAD values) and also to send policy decisions (such as the client close, server redirection, server close orders) towards its clients. In addition to the sensors defined for SAD processing, other sensors are needed by the P-SEAN management plan. These sensors are the sensors that help detecting server departure (network partitioning detection/forecasting, application or node closing …). Note that these sensors are the one that triggers the policy-rules related to server replacement (cf. Section IV.B.1).

The policy-engine takes all the decisions related to the server lifecycle. Hence, as stated above, when it runs the server, it may take policy-decisions (PDP) that have to be enforced within the other nodes. The policy-engine is also a policy enforcer (PEP). Indeed, whether the node is running the server or not, the policy engine enforces the decisions that concern the server starting or server replication. This is realized throughout the actuators. These decisions can either be received from a closing server or inferred from the *Situational Server-Election Policy*. The former case corresponds to the case where the old server has to be replicated (i.e. starting the new server within the node and replicating the shared information). The latter case corresponds to the case where the concerned node detects that there is no server currently running or starting in the network.

Note finally that in addition to the policy-rules defined by the *Situational Server-Election Policy*, the P-SEAN-enabled service may need other management policies. These policies can be related to server election, such as server election based on trust, per-cluster server election extension …, or not. These policies have to be specified in Ponder and introduced in the system throughout the Application-driven Policy Management API.

B. A COPS extension for Server Election and Maintenance Exchanges

According to the policy-rules defined by the *Situational Server-Election Policy* (Section IV), many information exchanges are realized between both the P-SEAN server and its clients. In order to realize these exchanges, a lightweight COPS protocol extension is proposed. The exchanges that have to be handled are of four types:

- Node connection to the server.
- Connection monitoring throughout KA messages.
- Decisions related to server departure.
- Decisions related to server-conflict resolution.

As will be explained in the following subsections, these situations can easily be implemented as COPS protocol extension. This is what motivated the choice of this protocol for P-SEAN policy-decisions and management information exchanges.

1) Node Arrival

When the application is launched, the server search procedure looks for a server which has already been elected and started. In the case where a P-SEAN server exists, the P-SEAN-enabled service starts as a client, connects to this server, and it sends its SAD to it. This is realized through the use of the connection procedure proposed in the COPS protocol. The SAD value will be sent within the OPN message as a client specific feature (i.e. as a named ClientSI object in the COPS terminology [8]).

2) Connection Monitoring

As explained above (cf. Section IV.B.2), connection monitoring is realized through the use of periodical *Keep Alive* exchanges between the server and each of its clients as proposed in COPS [8]. The use of these messages allows also the exchange of periodical information such as the SAD values. Indeed, as explained earlier, the SAD value is an evolving parameter. Furthermore, this evolution can not be predicted as it depends on multiple parameters. Each client has to send this value towards the server in order for this latter to maintain a list of possible substitutes. To do so, the clients use the KA messages to send their SAD periodically towards the server. Note that the server do not include any information within the echo KA it sends in response. The KA message format is then extended in order to encompass client specific feature (i.e. a *named ClientSI* object), namely the SAD value. Indeed, it does not contain such object in the legacy COPS protocol [8].

3) Server Departure Decisions

When a server has to be replaced by another participating node, it first informs this node by sending an unsolicited decision message (DEC). This decision message will encompass a decision object meaning that the corresponding node has to start the server functionality. It may also contain server specific data (*ClientSI Data*). The elected substitute acknowledges positively this decision by sending a report message (RPT). This is realized after server starting.

The closing server sends a Client Close (CC) message to all connected clients. Note also that in COPS, the PDP when closing PEP sessions may use the optional PDP Redirect Address Specific Object (PDPRedirAddr) to redirect its clients (PEPs) to the alternate server. This optional feature is used in P-SEAN in the case where a positive RPT is received by the closing server.

4) Server Conflict Decisions

In the case where a server detects one contending server, it connects to the contending server as a client in order to inform it about the conflict situation. To do so, the conflict-detecting server connects to the other servers using a slightly modified procedure as the one presented in Section V.B.1.

Each COPS message belonging to a particular COPS extension, identify the COPS extension to which it belongs using a particular COPS object: the client-type. This object is the unique identifier associated to a particular COPS extension. COPS SEME, being a COPS extension, has a unique identifier it uses for its normal operations. Then, in the case where a server detects a contending server (abnormal operational mode), it connects to it following the same procedure described

in Section V.B.1 but using a COPS extension identifier that is different from the one used for normal COPS SEME operations. This specific identifier is used solely for contention resolution in COPS SEME. Hence, the contacted server will be informed by the conflict. In all cases the contending server replies by sending a Client Close message encompassing its SAD value within the Error field contained in this COPS message. As a consequence, three situations may happen:

- The contending server has a better SAD. In this case the conflict-detecting server triggers a server replacement procedure.
- The conflict-detecting server has a better SAD. In this case the contending server triggers a server replacement procedure.
- The SAD values of both servers are equal. In this case both of them enter the *BEB* procedure and resend a Client Close (CC) message after the timer expires. Remind that the *BEB* procedure is repeated until one of the two servers triggers a server replacement procedure.

VI. Conclusion and Future Work

Group-collaboration applications are client-server applications where any participating entity can centralize the shared information and play the role of server. This kind of applications is widely used in both wired and wireless networks. Among them, numerous distributed management and control applications proposed for ad hoc networks use this mode. All these proposals argue on the necessity to realize an appropriate server election regarding to the performances of the network, the wireless station or the application. Indeed, it is clear that an inadequate choice of the entity running the server functionality can have a side effect on the network-, the application-, or the wireless station-performances.

In this paper we proposed a complete framework for Policy-based Server Election in Ad hoc Networks (P-SEAN). We identified the main concepts upon which this framework should be built, namely the *Serving Ability Degree* and the *Situational Server-Election Policy*, and we formalized them. Hence, the proposed framework implements all the situations that may happen for server election and maintenance in ad hoc networks (*Situational Server-Election Policy*). These situations have been specified as policy-rules thanks to the widely used Ponder policy specification language. Our framework also bases the election and maintenance processes on factors such as connectivity, processing power, RAM capacity, remaining battery life, etc. The *Serving Ability Degree* of each participating entity is hence processed using these factors. As it may depend on application-specific criteria, we do not propose a single formula for the processing of the *Serving Ability Degree* but a guideline towards doing this processing. Note that this one is performed using a method inspired from the Fuzzification procedure (Fuzzy Logic). Finally, in order to realize the *Server Election and Maintenance Exchanges*, a lightweight extension of the COPS protocol has also been proposed.

Note that our framework is generic and can be used with any group-collaboration application. It is also independent from the application implementation. One of the main features of the proposed framework is its extensibility. Indeed, as we used the Policy-based Networking paradigm, we made our framework capable of incorporating any additional application-specific criteria whether this one are specified as policies. Hence, the server election process may encompass policy-rules related to trust, to cluster/zone based election, or others.

Our framework uses several timers (KA timers, Contention Windows, Sleep Periods …). These timers have to be tuned adequately. The target of our future work is to made practical experiments and simulations in order to adequately adjust these timers. A complete simulation study aiming to demonstrate the robustness degree of such framework regarding to node velocities and mobility models will also be the subject of our future work.

References

[1] C.E. Perkins (Ed.), Ad Hoc Networking, Addison-Wesley, December 2000.

[2] K. H. Wang, and B. Li, "Group mobility and partition forecasting in wireless ad-hoc networks," in Proceedings of IEEE International Conference on Communications (ICC 2002), Vol. 2, pages 1017–1021, April 2002.

[3] H. Ritter, R. Winter, J. Schiller, "A Partition Detection System for Mobile Ad-Hoc Networks," First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON 2004), October 2004.

[4] K. S. Phanse, L. A. DaSilva, S. F. Midkiff, "Design and demonstration of policy-based management in a multi-hop ad hoc network," Elsevier Ad hoc Networks Journal, Vol. 3, N° 3, pp. 389-401, May 2005.

[5] D. C. Verma, Policy-Based Networking–Architecture and Algorithms, New Riders Publishing, November 2000.

[6] N. Damianou, N. Dulay, E. Lupu and M. Sloman, "The PONDER Policy Specification Language," Workshop on Policies for Distributed Systems and Networks (Policy'01), January 2001.

[7] N. Damianou, N. Dulay, E. Lupu et M. Sloman, "Ponder: A Language for Specifying Security and Management Policies for Distributed Systems, The Language Specification Version 2.3," Imperial College Research Report DoC 2000/1, Octobre 2000.

[8] J. Boyle, R. Cohen, D. Durham, S. Herzog, R. Raja et A. Sastry, "The COPS (Common Open Policy Service) Protocol", RFC 2748, Janvier 2000.

[9] L. A. Zadeh, "Outline of a New Approach to the Analysis of Complex Systems and Decision Processes," IEEE Trans. On Systems, Man and Cybernetics, 1973, 28-44.

[10] E. Guttman, C. Perkins, J. Veizades et M. Day, "Service Location Protocol, Version 2," RFC 2608, Juin 1999.

[11] The Salutation Consortium, http ://www.salutation.org/.

[12] LAN MAN Standards of the IEEE Computer Society,"Wireless LAN medium access control (MAC) and physical layer (PHY) specification," IEEE Standard 802.11, 1997.