A Role-Based Infrastructure for the Management of Dynamic Communities

Alberto Schaeffer-Filho[1], Emil Lupu[1], Morris Sloman[1], Sye-Loong Keoh[1], Jorge Lobo[2], and Seraphin Calo[2]

[1] Department of Computing, Imperial College London
180 Queen's Gate, SW7 2AZ, London, England
{aschaeff, e.c.lupu, m.sloman, slk}@doc.ic.ac.uk
[2] IBM T. J. Watson Research Center
19 Skyline Drive, Hawthorne, NY, 10532
{jlobo, scalo}@us.ibm.com

Abstract. This paper defines an operational framework for specifying and establishing secure collaborations between autonomous entities that need to interact and depend on each other in order to accomplish their goals, in the context of mobile *ad-hoc* networks. We call such collaborations *mission-oriented dynamic communities*. We propose an abstract model for policy-based collaboration that relies on a set of task-oriented roles. Nodes are discovered dynamically and assigned to one or more roles, and then enforce the policies associated with these roles according to the description of the community. In this paper we focus on the roles that are needed to provide management and security functions for dynamic communities.

1 Introduction

This paper seeks to address the problem of specifying and establishing a secure collaboration between autonomous entities that depend on each other and need to interact in order to accomplish their goals. We call such collaborations *mission-oriented dynamic communities*. Dynamic communities of autonomous entities, such as unmanned autonomous vehicles or robots in general, can be used to perform tasks that are dangerous or even impossible for humans. Such communities can be deployed in emergency operations after floods or earthquakes where teams of agents coming from different organizations are assembled for a mission; for reconnaissance of areas where hazardous chemicals or explosives may be present; or search and rescue missions involving teams of unmanned vehicles and rescue personnel. In these examples, the collaboration between the autonomous entities is crucial to accomplish the intended goals. Our objective is to dynamically create a secure collaboration between initially untrusted nodes without manually pre-configuring all nodes for their desired functions. Instead, a community must autonomously evolve and manage itself without human intervention. Thus, the main challenge is to devise a flexible infrastructure for

D. Hausheer and J. Schönwälder (Eds.): AIMS 2008, LNCS 5127, pp. 1–14, 2008.

community specification and management that can cater for such various requirements of many different applications.

Our approach is based on previous work on doctrines [1], which has been extended to cater for both management and application *roles*. In particular we focus on security management within the community. A community specifies a dynamic collection of roles to which nodes are assigned dynamically when discovered or as the mission context changes. Roles define two classes of policies, obligations and authorizations [2], that specify how the roles interact with each other in the scope of the community, as well as the services and resources that they allow other roles to access. Role assignment may be subject to *constraints* defined in the community specification that guarantee the integrity of the community. The doctrine approach concentrated management functions into a single coordinator node, whereas we explicitly identify a set of security management roles and their policies, and we split the management tasks across distributed collaborating nodes that are assigned to these roles. Our approach is flexible in that new roles can be easily defined for different management functions, depending on the risk context or the security requirements associated with each mission-oriented community. We also propose a methodology using policies to define flexible protocols for role interactions based on finite state machines which can be easily adapted to specific application requirements.

This paper caters for three important aspects regarding the community management: firstly, how our approach devolves management roles to community members by dynamically loading management tasks across distributed nodes; secondly, how to build a secure collaboration relying on a set of basic security mechanisms, which may be easily extended to address new security requirements; thirdly, how our infrastructure scales-down and can be deployed in resources with limited computational power and memory, typical of search and rescue applications (collaborations of autonomous robots or rescue personnel carrying small computing devices in the field).

The implementation of the framework defined here uses the infrastructure provided by *Self-Managed Cells* (SMCs) and the Ponder2 policy framework, which were developed at Imperial College [3], [4]. We use a scenario of a reconnaissance community of *unmanned autonomous vehicles* (UAVs), which form a mobile *ad-hoc* network. Typical examples of UAVs in this community are video surveillance and information aggregation vehicles that need to collaborate in order to achieve their goals [5], [6]. With respect to our past work, this paper extends [1] with: an enhanced role model for security management, an implementation on Gumstix and Koala robots rather than in a simulation, overhead/evaluation measurements, a formalization of community behavior and revisions based on Ponder2 rather than Ponder. Additionally, this paper also explores the uses of the middleware described in [3], [4].

The paper is structured as follows: Section 2 describes our role-based community model. Section 3 presents the security requirements and how protocols for management and security of communities can be specified using policies. Section 4

describes our prototype and Section 5 outlines the related work. Finally, Section 6 presents the concluding remarks.

2 Role-Based Community Model

A community specification describes a set of task-oriented *roles* that need to collaborate in order to achieve their goals. The specification contains a number of *policies* that must be enforced by different entities, according to their roles in the community. Nodes are assigned to roles in order to perform specific tasks in the community, based on their credentials and capabilities. The community specification also defines a set of *constraints* relating to role assignments. Policies are of two types: *obligation* and *authorization* policies.

Obligation policies are of the form:

$$on < event > do \\ if < conditions > then \\ < target >< action >;$$

Obligations cater for the adaptive behavior of nodes. They specify what management actions (also referred to as *methods*) must be performed in response to events, provided a set of conditions is satisfied. The event is a term of the form *e(a1,...,an)*, where *e* is the name of the event and *a1,...,an* are the names of its attributes. The condition is a boolean expression that may check local properties of the nodes and the attributes of the event. The target is the name of a role where the action will be executed and so the target must support an implementation of the action. The action is a term of the form *a(a1,...,am)*, where *a* is the name of the action and *a1,...,am* are the names of its attributes.[1] The attributes of an event may be used for evaluating the condition (to decide whether to invoke the action or not), or they may be passed as arguments to the action itself. Implicitly the role to which this policy belongs is the *subject* of the obligation i.e. the entity enforcing the policy, and the action is invoked on a *target* role. Note the target may be the same as the subject i.e. a role may perform actions on itself.

Authorization policies are of the form:

$$auth[+/-] < subject > \longrightarrow if < condition > then \\ < target >< action >;$$

These policies are access control rules that specify what actions a *subject* is allowed (positive authorization) or forbidden (negative authorization) to invoke on a *target*. The subject and the target are role names. The action and the condition are defined like in obligations. Authorization decisions could be made by one or more specific roles in the community, but our current implementation is based on the target making decisions and enforcing the policy as we assume target nodes wish to protect the resources they provide to the community.

[1] To simplify notation an obligation policy can have a list of target-action pairs, all evaluated when the event is true and the condition holds.

Let R be a set of roles, A a set of authorization policies and O a set of obligation policies. For any role r in R, r is defined in (O, A) as the collection of obligation policies in O and authorization policies in A such that the subject in the obligation policies is r and the target in the authorization policies is also r.

A community may also specify a set of *constraints* expressing additional conditions on role assignments. We currently support two types of constraints: *cardinality* and *separation of duty* constraints. Cardinality constraints (CC) are defined as a relation between a role and a minimum and a maximum number of instances that the role can have in the community. Hence:

$$CC \subseteq R \times \mathbb{N} \times \mathbb{N}$$

Where $\mathbb{N}$ denotes the set of natural numbers, and for any tuple $(r, n, m) \in CC, n \leq m$, and r cannot appear in more than one tuple in CC.

Separation of duty constraints (SC) [7] are defined by a relation which specifies that a node cannot be assigned to a set of roles at the same time (e.g. the same node may not perform roles for *"handling hazardous chemicals"* and for *"supplies delivery"* simultaneously). Hence:

$$SC \subseteq \wp(R)$$

Where $\wp(R)$ denotes the power set of R. A set s in SC indicates that no node in the community can be assigned to all the roles in s simultaneously.

The set of constraints C of a community is defined by the union of its cardinality constraints and separation of duty constraints, $CC \cup SC$. Finally, a community description i is defined by the set of roles R, the sets of policies O and A, and the set of constraints C:

$$Community_i = < R, O, A, C >$$

The abstract model representing a community is illustrated in Fig. 1. Although there is some similarity with the RBAC model [8], our roles are not just limited to

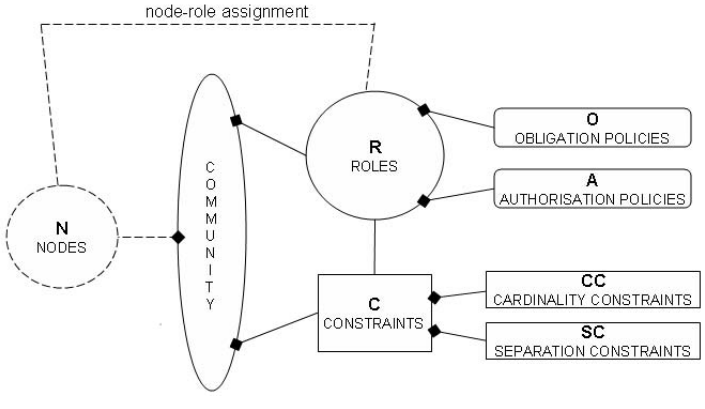

Fig. 1. Dynamic community (solid lines represent the community specification and dashed lines represent the dynamic assignment of new nodes to this community)

defining authorizations in terms of privileges, but they also cater for obligations. We do not support role inheritance in our community model because of runtime penalties it may incur in a distributed environment and also because such inheritance would not apply to the obligations which are also part of the roles [9].

Nodes are dynamically assigned to one or more roles defined in the community specification. As described in the next section, this assignment is usually undertaken by a *coordinator* role and most flexibly defined by a set of policies determining to which role a newly discovered entity should be assigned based on its capabilities. In order to characterize the assignment of nodes to roles, we rely on the abstraction of *interfaces*. An *interface itf* is defined by a tuple $< Cap, Met, Eve >$, representing the sets of capability descriptions Cap associated with this interface, and also the sets of methods Met and events Eve that it offers.

- Met : the collection $\{Met_k : 1 \leq k \leq o\}$ represents a set of method names offered by this interface.
- Eve : the collection $\{Eve_l : 1 \leq l \leq p\}$ represents a set of event names offered by this interface.
- Cap : the collection $\{Cap_i : 1 \leq i \leq m\}$ represents a set of high-level capability names that this interface offers. Each capability Cap_i is associated with a subset of the collection $\{Met_k : 1 \leq k \leq o\}$ of methods and a subset of the collection $\{Eve_l : 1 \leq l \leq p\}$ of events.

While capabilities represent the functionality of an interface at an abstract level (e.g. *"video"*, *"storage"*), methods and events describe its functionality at the implementation level.

An interface definition may be associated with a role, thus identifying the functionality expected of a node before it can be assigned to that role or with a node itself, thus identifying the functionality provided by that node. The uses are also referred to as *expected interfaces* and *provided interfaces* respectively. A node can be assigned to a role if its provided interface *entails* the expected interface of the role. The *entailment operator* for interfaces is defined as follows: for any $itf_a < Cap_a, Met_a, Eve_a >$, $itf_b < Cap_b, Met_b, Eve_b >$, we say that itf_a entails itf_b, or $itf_a \models itf_b$, if:

$$(Cap_b \subseteq Cap_a) \wedge (Met_b \subseteq Met_a) \wedge (Eve_b \subseteq Eve_a)$$

Therefore, $itf_a \models itf_b$ if all the elements of itf_b are also present in itf_a. Informally, a node can be assigned to a role if its provided interface is more general than the role's expected interface.

3 Secure Community Management

This section describes how the community model presented in Section 2 can be applied for the management of secure components in a dynamic community. We start by outlining the security requirements and the community operations, and then introduce a methodology for the specification of management protocols in dynamic communities.

3.1 Security Requirements and Management Roles

Security management in dynamic communities requires supporting the functions of *authentication*, *membership management* and *access control*. In addition, a set of management procedures is required for the *coordination* of communities. These are essential mechanisms because they ensure that all members are authenticated before joining the community, that the community keeps track of all participants and their roles (and can detect failures), that access control restrictions apply to all resources and services offered by nodes, and that the vital management procedures for community maintenance are performed. We describe in this section how the basic security management requirements for communities are fulfilled.

The *coordinator* role specifies the overall management of the community and groups tasks related to community bootstrapping and assignment of new members to roles, as well as the validation of constraints. Initially, the coordinator role is responsible for broadcasting messages advertising the community to nearby nodes. It enforces policies that govern the preferred assignment strategy of nodes to roles based on their capabilities (but which remains subject to the node's interface satisfying the assignment condition described in Section 2). Whenever a node is assigned to a role, the policies associated with that role are loaded into the node. The coordinator checks whether the minimum requirements for the community are met and whether separation of duty constraints are satisfied. If the coordinator detects that the constraints are not being met, it may try to reassign roles in the community. If this is not possible the coordinator may decide to dissolve the community. At bootstrap, the node that instantiates the community specification is automatically assigned to the coordinator role. In addition, at this point the coordinator node may be also assigned to other critical management roles (e.g. *authenticator* role): however, the coordinator may delegate one or more of these roles as the community evolves and new members join it.

The *authenticator* role validates the identity and attributes of nodes that wish to join the community. A typical approach for authentication is based on the use of public-key certificates and we assume that only nodes possessing certificates signed by trusted CAs are able to join a community. To this end, public-keys of the certification authorities (CAs) that are relevant to the community may be pre-loaded in the community specification. This avoids the need to contact a CA and is necessary in deployed environments where access to a network infrastructure may be intermittent or non existent. Our initial implementation is based on a PKI solution and uses *X.509 digital certificates*. Non-PKI based approaches are currently also being investigated.

The *membership manager* role keeps track of the members in the community. The community must deal with nodes which move out of communication range, run out of battery or disconnect. If a member does not signal its presence within a given time period, it is considered to have left or become disconnected and the membership manager informs the other members that a node has left. This causes the constraints of the community to be reevaluated by the coordinator, as the departure of a member may violate the cardinality constraints.

Finally, *access control* is our last basic security requirement. Our current implementation distributes the access control enforcement amongst all (target) roles to allow them to protect their resources and permit access to specific subject roles (see Section 2). However, if an entity is not able to enforce its own access control policies, it may outsource these control decisions to a specific role in the community or to its own trusted agent. Note that the community is not limited to these management roles; new roles can be specified to perform additional security or management procedures as required.

3.2 Community Management Overview

When receiving the community broadcast sent by the *coordinator*, a node presents its *X.509 digital certificate* to the community's *authenticator*, which then performs the node validation. The node also validates the authenticator's credentials and the community description and decides whether to join the community or not. If mutual authentication is successful, potential roles for assignment are selected by the *coordinator*, according to the node's capabilities: policies specify the preferable assignment strategy by listing one or more *"required"* capabilities for each role. These are matched against the node's capabilities. The matching is performed by selecting the roles whose list of required capabilities is contained in the node's list of actual capabilities. Among these roles, only those satisfying the cardinality and separation of duty constraints are selected, and the node is finally assigned to these roles by the *coordinator*.

The assignment process includes transferring to the node the obligation and authorization policies that are part of a role specification, event definitions needed by those policies, and a subset of the domain structure of the coordinator is copied to the node: this contains a list of all roles defined by the community (which can be seen as placeholders) and the members currently assigned to each role (i.e., nodes associated with each placeholder).

The consistency of the membership database is kept using a soft-state strategy, where nodes that do not periodically renew their entry with the *membership manager* are automatically removed, using the algorithm described in [1]. Notice, however, that this algorithm is solely performed by the *membership manager* role. There is obviously a trade-off on how frequently nodes should revalidate their soft-state and how often updates in the membership list should be propagated by the *membership manager* to the other members. For this reason, these actions are modeled by policies which can be easily changed to adapt updating rates to different community requirements.

3.3 A Methodology for Modeling Community Management

The interaction between the roles in the community is defined in terms of the obligation policies each (subject) role enforces. These policies specify actions that must be performed in response to events, and such actions can be seen as steps in the protocol that defines the interaction between roles. We can model such interactions in a community by defining a *finite state machine* (FSM), where arrows represent the generation of events and states are actions that represent

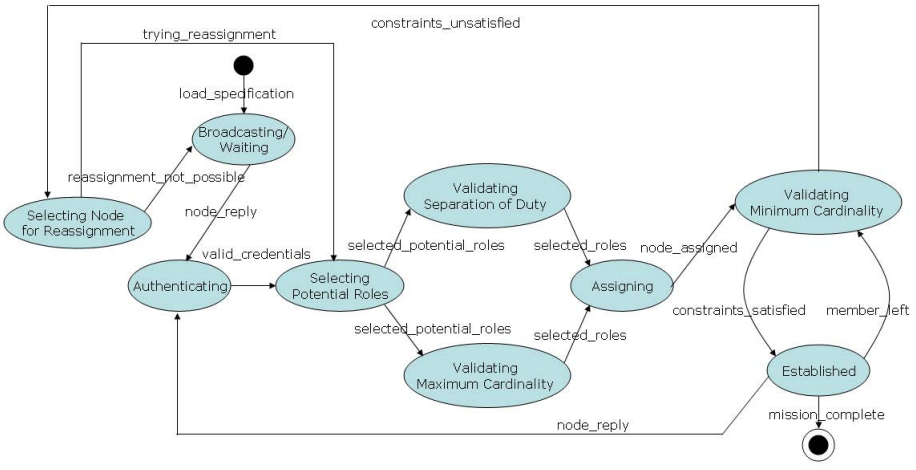

Fig. 2. Modeling community management

protocol steps. We only focus on modeling interactions between management roles, but the same approach can be used for application-specific interactions.

The FSM in Fig. 2 exemplifies an interaction protocol in a community that supports the three management roles previously described: coordinator, authenticator and membership manager. The protocol specifies that after the community specification is loaded into the coordinator, the community broadcasts its presence and waits for node replies. A reply triggers the authentication step; if the node is successfully authenticated, the potential roles for assignment are selected; then, constraints are checked and the node is assigned to the roles that satisfy both maximum cardinality and separation of duty constraints, provided the node possesses the required capabilities for the roles. At this point, if the minimum cardinality constraints are satisfied, the community changes to the state *"established"*, otherwise it remains in a *"broadcasting/waiting"* state. The protocol may have other steps, but our intended contribution is not in terms of defining a specific management protocol, but to illustrate the methodology for modeling community interactions: each step in the protocol can be seen as an action (or set of actions) performed by a policy triggered by the event that corresponds to the incoming arrow – if a step also generates the event required to trigger the policy which specifies the next step, we can *"chain"* the steps of the protocol.

We are therefore specifying the community management in terms of policies that perform steps in the protocol. This is similar to the approach used in PDL [10], where internal events are used to link the execution of policies. However, in PDL only local events were considered, whereas here events can be sent to remote nodes performing a given role. This flexibility is clearer if we consider the addition of entirely new management roles to the community. These can be used then to enhance the protocol, by adding new management steps to it in terms of additional obligation policies.

4 Implementation and Evaluation

The work on dynamic communities was implemented in Java, relying on the infrastructure provided by *Self-Managed Cells* (SMCs) [3], which uses the Ponder2[2] policy framework. An SMC consists of hardware and software components forming an autonomous administrative domain which supports both *obligation* and *authorization* policies. Policies can be added, removed, enabled and disabled to change the behavior of an SMC without interrupting its functioning. We assume the nodes assigned to roles within a community are SMCs.

The evaluation described in this section intends to show how our infrastructure for community management scales-down and can be deployed in resources with limited computational power and memory, which are likely to be found in search and rescue applications such as collaborations of autonomous robots or rescue personnel carrying small computing devices in the field. We deployed our prototype in two classes of lightweight, constrained devices: Gumstix[3] and Koala robots[4] (Fig. 3). The Gumstix has a 400 MHz Intel XScale PXA255 processor with 16 MB flash memory and 64 MB SDRAM, running Linux and Wi-Fi enabled. The Koala robot has a Motorola 68331, 22 MHz onboard processor, 1 MB ROM and 1 MB RAM. The robot is extended with a KoreBot module which has a 400 MHz ARM PXA255 processor, 64 MB SDRAM and 32 MB flash memory, running Linux and also Wi-Fi enabled. In addition, the robot has 16 infrared proximity sensors around its body, and a video camera. Both run the lightweight JamVM[5].

Either a Gumstix, which is a very portable device, or a robot can discover other Gumstix or robots, assign them to roles, and deploy the policies pertaining to the role on them. New members are authenticated using *X.509 digital certificates* before a policy-based decision on their admission to the community and role assignment is made. Assignment policies, enforced by the *coordinator* role,

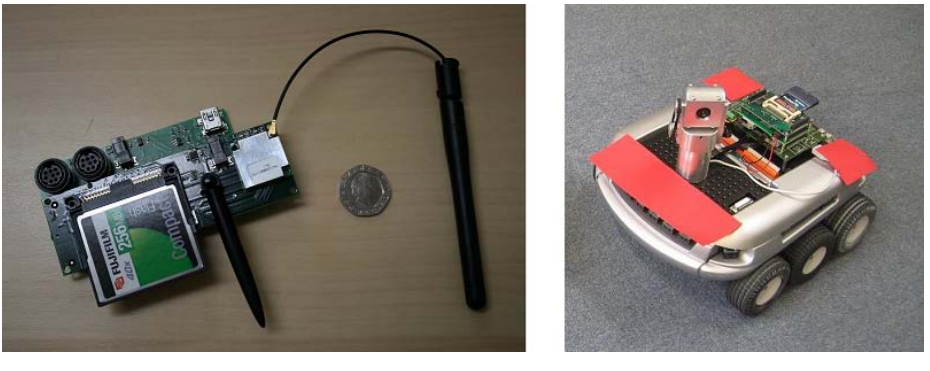

Fig. 3. Gumstix (left) and Koala robot with video capability (right)

[2] http://www.ponder2.net
[3] http://www.gumstix.com
[4] http://www.k-team.com
[5] http://jamvm.sourceforge.net

```
obliCoord := root/factory/ecapolicy create.
obliCoord event : root/event/nodeAuthenticated.
obliCoord condition : [: cap | interface hasCapabilities : "video" from : cap ].
obliCoord action : [: name : address | role/coordinator assign : name
                    from : address to : "surveyor" community : reconnaissance ].
```

Fig. 4. Policy specifying assignment rule for nodes possessing video capability

are used to specify preferences for the assignment of nodes to roles. Membership is also managed. Members of the community must periodically signal their presence with the *membership manager* and should a node become disconnected from the community (e.g. a robot runs out of battery) its role can be re-assigned to one of the existing devices, provided it has the capabilities to fulfil that role.

We show in Fig. 4 an obligation policy, to illustrate the kind of policies loaded across SMCs participating in a community. The policy is a typical assignment policy, specifying that nodes possessing the capability *"video"* must be preferably assigned to the role *surveyor* (and would normally belong to the *coordinator* role specification).

A discussion on the Ponder2 syntax is out of the scope of this paper, but essentially this snippet creates an *event-condition-action* policy (*ecapolicy*) named *obliCoord*, which is triggered by an event of the type *nodeAuthenticated*. The condition verifies if *"video"* is among the set of capabilities *"cap"* provided as argument of the event. If the condition evaluates to true, the action to be executed is the assignment (action *assign*) of the node whose *name* and *address* were provided as parameters of the event to the role *surveyor* with respect to the object *reconnaissance* (which is an instance of *Community*). The target of the *assign* action is the *coordinator* role.

The size of the bytecodes required for running the prototype, including Ponder2 and necessary libraries, is 710 KB. The size of a typical policy written in Ponder2 syntax is about 620 bytes (but this obviously depends on the complexity of the policy). The size of a typical community specification (with 5 roles, each role specifying 5 policies) written in Ponder2 is about 20.4 KB (but it is also subject to the complexity of the policies, number of policies, and number of roles in the specification). In terms of memory usage during runtime, we observed that a Gumstix running the *coordinator* role, and keeping the community specification loaded in memory, required 15 MB for the Ponder2 process and 9224 KB for the *rmiregistry* process[6] (*RMI* is one of the communication protocols supported by SMCs). On the other hand, a Koala robot running an application role (containing 5 policies) required 8384 KB for the Ponder2 process and 4492 KB for the *rmiregistry* process. Increasing the number of policies loaded in the robot from 5 to 10 caused a negligible overhead in terms of memory consumption. The small

[6] By comparison, an empty JamVM and rmiregistry uses about 3200 KB and 5900 KB respectively, and a JamVM running an empty Ponder2 instance and rmiregistry uses about 8200 KB and 5900 KB respectively.

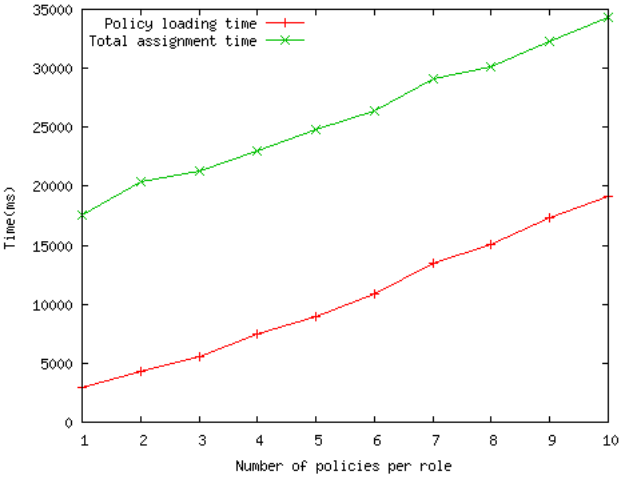

Fig. 5. Total assignment time *versus* policy loading and deployment time

footprint needed for our role management infrastructure highlights that other devices with a similar configuration and capacity could also have been used.

The graph in Fig. 5 depicts some initial performance measurements running our prototype on devices with very constrained computational power. The tests consisted in measuring the time taken for a Gumstix running the *coordinator* role to assign a nearly discovered Koala robot to another role, containing a variable number of policies. We have measured both the time taken to transfer and deploy only the policies, as well as the whole assignment process The latter involves the transfer of the policies, the transfer of additional community information such as event definitions, the creation of role placeholders in the remote node, sending an event informing that a new node has joined the community, and the attribution of the discovered node to the role in question by the *coordinator*.

Our results show that for roles with a small number of policies the total cost of assignment is dominated by the cost of tasks not related to policy transfer, but as we increase the number of policies per role, this fixed cost tends to become negligible in comparison to the cost of loading and deploying policies (which increases linearly). This suggests that the prototype is able to support more complex roles where the only significant cost is the policy transfer, because the residual component of the assignment time remains constant. We also observed that most of this time (about 97% on average) is spent on RMI serialization and network delay when transferring data from the Gumstix to the robot, and only a small part corresponds to the time that is actually spent by the robot to instantiate the policies. We expect that Ponder2's ability of supporting alternative communication protocols will mitigate this overhead. The evaluation of other aspects of the community strategy, in particular the cost of role replacement when a node fails, remains to be done as future work.

5 Related Work

Although related work exists in the area of ad-hoc communities, we are not aware of any that similarly addresses structural community issues based on dynamic roles and assignment policies. The industrial work on autonomic computing, led primarily by IBM [11] but also addressed by Motorola [12] and HP [13], usually tends to focus on network management of large clusters and web servers. Self-managed cells are suitable for more dynamic and mobile pervasive settings, e.g. communities of ad-hoc unmanned autonomous vehicles. The control-loop performed by SMCs is much simpler than the control-loop used by those projects, as it does not depend on planning techniques or ontologies in order to support self-management. Mobile UNITY [14] provides a notation system for expressing the coordination among mobile unities of computation. It focuses on the formalization of coordination schemas, and not on the management of communities.

Research on policies has been active for several years, especially regarding policies for network and systems management. Examples include PCIM [15], PDL [16] and PMAC [17]. Although they use similar event-condition-action rules for encoding adaptation, these approaches are targeted for management of large-scale and networked systems, and do not scale-down for managing small devices.

Finally, the management of dynamic communities may be enhanced with the inclusion of additional security and management mechanisms: threshold cryptography [18] for preventing a compromised authenticator from accepting rogue members and intrusion detection [19] for monitoring potential risks and attacks are some of the options, but their inclusion in our dynamic communities still requires further investigation. Our focus however is not on developing such mechanisms but instead on the management infrastructure they require.

6 Discussion and Concluding Remarks

As well as application-specific roles, a community infrastructure must define a flexible framework for the management of the community itself. This is most flexibly achieved by: *(a)* identifying roles corresponding to the community management functions, *(b)* defining the community operation in a higher level FSM based model, and *(c)* dividing and deploying the management steps as dynamically replaceable policies.

Our strategy of *splitting* the management tasks in several different roles, which will be then assigned to different nodes, caters for the distributed management of a community. Typically, in search and rescue missions, the *coordinator* is assigned based on chain of command and on capabilities. It may constitute a single point of failure, however a community is not under threat if the coordinator fails – the community is stable and continues to operate, but new members cannot join until a new coordinator is assigned. This is mitigated by *assigning a replacement node to the role*. Replicating the coordinator (or any management role) would require replica consensus and would significantly increase messaging (with power consumption and security implications), and therefore the role replacement strategy is preferable. Our model caters for an extensible infrastructure for management of dynamic

communities, where new roles can be added, according to the management and security requirements of each mission-oriented dynamic community.

The work presented in this paper significantly extends our past results and shows how Ponder2 and Self-Managed Cells offer a flexible infrastructure for self-management and autonomy in such MANETs. The overall implementation overhead shows that our prototype scales well and can be deployed in constrained resources with limited computational power and memory, which are likely to be found in mobile ad-hoc communities.

To apply our model in larger scale scenarios, we will require the ability to cater for communities that interact with other communities. For example, we can think of *hierarchical composition* of communities, where a rescue team has as one of its members a medical team, which is a community itself. The inner community would encapsulate its management and the outer would not be concerned with the details of the management in the inner community. This architecture of hierarchical communities allows the management to scale-up, with self-managed, encapsulated communities, but future work still has to investigate the abstractions required to support cross-community interactions.

Acknowledgments

Research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. We also acknowledge financial support in part from the EC IST EMANICS Network of Excellence (#26854). Finally, the authors wish to thank Eskindir Asmare for his contributions in defining the reconnaissance scenario for UAVs used in this paper.

References

1. Keoh, S.L., Lupu, E., Sloman, M.: Peace: A policy-based establishment of ad-hoc communities. In: Proc. of the 20th Annual Computer Security Applications Conference (ACSAC), Washington, DC, pp. 386–395. IEEE Computer Society, Los Alamitos (2004)
2. Sloman, M., Lupu, E.: Security and management policy specification. IEEE Network 16(2), 10–19 (2002)
3. Lupu, E., Dulay, N., Sloman, M., Sventek, J., Heeps, S., Strowes, S., Twidle, K., Keoh, S.L., Schaeffer-Filho, A.: AMUSE: autonomic management of ubiquitous systems for e-health. J. Concurrency and Computation: Practice and Experience (May 2007)

4. Schaeffer-Filho, A., Lupu, E., Dulay, N., Keoh, S.L., Twidle, K., Sloman, M., Heeps, S., Strowes, S., Sventek, J.: Towards supporting interactions between self-managed cells. In: 1st International Conference on Self-Adaptive and Self-Organizing Systems (SASO), Boston, USA, pp. 224–233. IEEE Computer Society, Los Alamitos (2007)
5. Asmare, E., Dulay, N., Lupu, E., Sloman, M., Calo, S., Lobo, J.: Secure dynamic community establishment in coalitions. In: MILCOM, Orlando, FL (2007)
6. Asmare, E., Dulay, N., Kim, H., Lupu, E., Sloman, M.: A management architecture and mission specification for unmanned autonomous vehicles. In: 1st SEAS DTC Technical Conference, Edinburgh, Scotland (2006)
7. Clark, D.D., Wilson, D.R.: A comparison of commercial and military computer security policies. In: IEEE Symposium on Security and Privacy (1987)
8. Sandhu, R.: Rationale for the rbac96 family of access control models. In: RBAC 1995: Proceedings of the first ACM Workshop on Role-based access control, p. 9. ACM Press, New York (1996)
9. Lupu, E., Sloman, M.: A policy based role object model. In: Proc. 1st Int. Enterprise Distributed Object Computing Workshop, Gold Coast, Queensland, Australia, pp. 36–47. IEEE, Los Alamitos (1997)
10. Bhatia, R., Lobo, J., Kohli, M.: Policy evaluation for network management. In: INFOCOM, Tel-Aviv, Israel, pp. 1107–1116. IEEE CS-Press, Los Alamitos (2000)
11. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–50 (2003)
12. Strassner, J., Agoulmine, N., Lehtihet, E.: Focale a novel autonomic networking architecture. In: Latin American Autonomic Computing Symposium, Campo Grande, MS, Brazil (July 2006)
13. HP: Hp utility data center: Enabling enhanced datacenter agility (May 2003), `http://www.hp.com/large/globalsolutions/ae/pdfs/udcenabling.pdf`
14. Roman, G.C., Payton, J.: Mobile unity schemas for agent coordination (March 2003)
15. Moore, B., Ellesson, E., Strassner, J., Westerinen, A.: Policy core information model, version 1 specification. request for comments 3060, network working group (2001), `http://www.ietf.org/rfc/rfc3060.txt`
16. Lobo, J., Bhatia, R., Naqvi, S.: A policy description language. In: Proceedings of the 16th National Conference on Artificial Intelligence, Orlando, FL, July 1999, pp. 291–298 (1999)
17. Agrawal, D., Calo, S., Giles, J., Lee, K.W., Verma, D.: Policy management for networked systems and applications. In: Proceedings of the 9th IFIP IEEE International Symposium on Integrated Network Management, Nice, France, pp. 455–468. IEEE CS-Press, Los Alamitos (2005)
18. Zhou, L., Haas, Z.: Securing ad hoc networks. Technical report, Cornell University, Ithaca, NY, USA (1999)
19. Lunt, T.F.: A survey of intrusion detection techniques. Computers and Security 12(4), 405–418 (1993)