A Role-based Infrastructure for the Management of Dynamic Communities

Alberto Schaeffer-Filho, Emil Lupu, Morris Sloman, Sye-Loong Keoh
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

Jorge Lobo, Seraphin Calo
IBM T. J. Watson Research Center
19 Skyline Drive, Hawthorne, NY, 10532
{jlobo, scalo}@us.ibm.com

Abstract*—This paper addresses the problem of specifying and establishing secure collaborations between autonomous entities that need to interact and depend on each other in order to accomplish their goals. 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 start enforcing the policies associated with these roles according to the description of the community. In this paper we describe a basic set of management roles that are needed to provide management and security functions for dynamic communities. Policies are used to specify management protocols as these can be easily modified to reflect different adaptive strategies. We focus on collaboration between nodes in the context of mobile *ad-hoc* networks. Our implementation is based on a surveillance scenario using unmanned autonomous vehicles (UAVs). However our approach to dynamic communities could be applied to emergency services at a disaster site, search and rescue applications or many military scenarios.

I. Introduction

This paper addresses 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. Dynamic communities are applicable to rescue 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 military missions involving teams of vehicles and foot-soldiers. In these examples, the collaboration between a set of autonomous entities is crucial to accomplish the intended goals. Our objective is to create a trusted community for secure collaboration between initially untrusted nodes without manually pre-configuring all nodes which are potential members of the community or providing a shared information base. Instead, a community must autonomously evolve and manage itself without human intervention. The main challenge is to devise a flexible infrastructure for 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] but we now cater for both management and application *roles*. A role specifies how a node assigned to that role interacts with other roles, as well as what services or resources that it offers can be accessed by other roles. The role to which a node is assigned will depend on the node's capabilities in terms of resources it can offer, its credentials or the current mission context. The community thus specifies a dynamic collection of roles to which nodes are assigned either initially or dynamically when discovered or as the mission context changes. Roles define two classes of policies, obligations and authorizations [2], which specify how the roles interact with each other in the scope of the community. The community may also define a set of constraints that indicate separation of duty and cardinality constrains on the number of role instances that make a community viable. Our communities provide a more distributed and extensible set of management and security roles to meet management requirements compared to doctrines which concentrated these functions into a single coordinator. 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.

The implementation of the communities is based on the work on *Self-Managed Cells* (SMCs) developed at Imperial College [3], which provides a policy service, event bus and a service for discovering new potential members of the SMC. The policy approach is itself based on previous work from Imperial College [4], [5].

We use a scenario of a reconnaissance community of *unmanned autonomous vehicles* (UAVs), from a UK/US coalition, 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 [6], [7]. We indicate the working of the security and management of the community in terms of relevant roles. The 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 and the management procedures can be adapted by changing the policies which define the management protocols.

The paper is structured as follows: Section II focus on the

static specification of communities and the abstract model used to represent a collaboration between roles. Section III describes our security requirements, and how protocols for management and security of communities can be specified using a combination of policies representing different steps in such protocols. Section IV describes our prototype, and Section V briefly outlines some future work. Sections VI and VII present related work and concluding remarks.

II. Community description

A community specification describes a set of task-oriented *roles* that need to collaborate in order to achieve the community 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 specific roles in order to perform specific tasks in the community, based on the node's 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 terms of the form:

```
on <event> do
    if  <conditions> then
        <target> <action>;
```

These event-condition-action rules specify what actions must be performed in response to events of interest, provided the conditions for the rules are 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 (e.g. location, time, etc.) 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] There is an implicit role name called the *subject* associated with an obligation which actually enforces the obligation policy, and the action is invoked on a *target* role. Note the target role may be the same as the subject i.e. a role may perform actions on itself. Obligation policies cater for self-adaptation, because they can encode a reactive behavior in response to events of interest.

Authorization policies are terms of the form:

```
auth[+/-] <subject> -> 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 policy decisions could be made by one or more specific roles in the community, but our current implementation is based on the target making authorization decisions and enforcing the policy as we assume target nodes wish to protect the resources they provide to the community.

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. Note that there is no fundamental difference between management or application roles in the community.

We currently support only 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. Separation of duty constraints (SC) are defined by a relation which specifies a conflict if a node is assigned to more than one role in the set at the same time.[2] 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.

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

Where $\wp(R)$ denotes the power set of R. A set s in SC indicates that no node in the community can be assigned 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 Figure 1. Although there is some limited similarity with the RBAC model [8], our roles are not just limited to defining authorizations in terms of privileges, but cater for obligation policies and we do not support inheritance of privileges between role instance hierarchies.

The schema of a community specification is illustrated in Figure 2. A community specification is meant to be loaded and instantiated in a node that will be responsible for the community coordination and that will be performing the *coordinator* role. Coordinator is a special management role that must be present in any community specification, and it will be described in more detail in the next section. In addition to the community specification the schema also has a

[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.

[2] Our current implementation limits the separation of duty constraints to sets of cardinality 2.

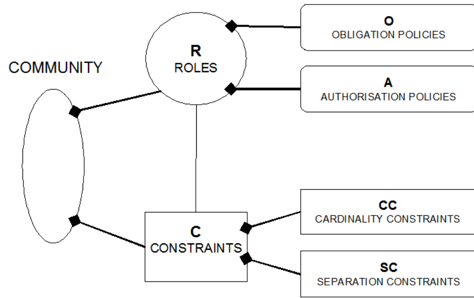

Fig. 1. Abstract community description.

few directives needed by the implementation. It specifies the address of the node that will play the role of *coordinator* and will bootstrap the community, and the set of trusted public-keys that will be used in the authentication of new members that wish to join the community (which will be described later, in Section III-A).

```
community <community_name> (<address>, <public_keys>) {
    constraints {
        cardinality  := (<role_name>, min:<x>, max:<y>)
                        (<role_name>, min:<x>, max:<y>)
                        (<role_name>, min:<x>, max:<y>)
                        (<role_name>, min:<x>, max:<y>)

        separation   := (<role_name_1>, <role_name_2>)
                        (<role_name_1>, <role_name_2>)
                        (<role_name_1>, <role_name_2>)
    }
    role <role_name_1> {
        obligation {
            <list_of_obligation_policies>
        }
        accessControl {
            <list_of_authorization_policies>
        }
    }
    .
    .
    .
    role <role_name_n> {
        obligation {
            <list_of_obligation_policies>
        }
        accessControl {
            <list_of_authorization_policies>
        }
    }
    initialization {
        <initial_variable_assignments>
        <initial_role_assignments>
    }
}
```

Fig. 2. Community specification schema.

The schema also has an initialization block, which performs initial variable and role assignments that are meant to be executed upon community instantiation. As an example, the node whose address was passed as one of the community instantiation arguments may be assigned to a specific set of roles upon community instantiation. However, initial assignments may change as the community evolves and new members are discovered. This block is interpreted and used only by the coordinator.

The community will dynamically evolve and grow as other nodes are automatically discovered and assigned to one or more roles specified in the community, provided the constraints are satisfied. Whenever a node is assigned to a role, the set of policies associated with that role is dynamically loaded into the node. By interpreting the policies associated with their respective roles, nodes will perform the tasks that are meant to accomplish the community goals.

III. Secure community management

Based on the community specification model presented in the last section, we describe here how a community is instantiated and how it evolves during runtime when new members join the community. In a community there can be nodes performing management roles that are independent of the application and other roles that are application dependent. In this section we will focus only on an application independent basic set of management roles and their interactions which could easily be extended with new roles and/or new policies relating to their interactions.

A. Security requirements and management roles

We consider a set of basic security mechanisms that we regard as essential for the construction of secure dynamic communities. These are *authentication*, *membership management* and *access control*. In addition, there is a set of management procedures required for the *coordination* of dynamic communities. We claim that these are the essential mechanisms because they will ensure that all members are authenticated before joining the community, that the community keeps track of all participants and the roles they are playing, that access control restrictions apply to all resources and services offered by nodes performing roles, and that the vital management procedures for community maintenance are performed. We describe in this section how the basic security and management requirements for dynamic communities are fulfilled by the collaboration between a set of management roles.

As mentioned before, the *coordinator* is a special role that *must* be part of any community specification. It performs the overall management of the community, such as tasks related to the community bootstrapping and assignment of new members to roles, as well as the validation of the community constraints. Based on the list of current members participating in the community and the set of cardinality constraints, the coordinator will check whether the minimum requirements for the community are met (similarly, the coordinator will check separation of duty constraints upon role assignment). Additionally, if the coordinator detects that the constraints in the community are not being met, it may try to reassign roles in order to keep the community running. If this is not possible the coordinator may decide to dissolve the community. Initially, the node performing the coordinator role may be also assigned to several other management roles. However, the coordinator may delegate one or more of these roles as the community evolves and new members join.

A second management role is the *authenticator*. Authentication is required in order to validate the identity of nodes that wish to join the community. A typical and fairly simple approach for authentication is based on the use of public-key certificates. The public-keys of the certification authorities (CAs) that are relevant to the community must be pre-loaded in the community specification. We assume that only a node that possesses a valid certificate signed by one of such CAs is able to join a community (other restrictions may apply as we will see later, but having a valid certificate is the minimal requirement). The public-keys loaded in the community specification will avoid access to a centralized certification authority as there may not be access to a network infrastructure for this. The initial simple authentication implementation does not cater for key revocations. We may integrate non-PKI based authentication in future work.

The *membership manager* role is responsible for keeping track of the current members in the community. Because our application area, and pervasive environments in general, often involves mobile and highly dynamic resources, the list of members must be constantly updated. New nodes may join the community or current members may leave at any time. The membership manager notifies other members in the community about changes in the members list, which also specifies the roles each member is performing.

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 II). However, if an entity is not able to enforce its own access control policies, it may outsource these to a specific role in the community or to its own trusted agent.

The community is not limited to these management roles; new roles can be specified to perform additional security or management procedures as required. Other security mechanisms that might be worth investigating are: threshold cryptography, which could be used in conjunction with multiple authenticators to prevent a compromised authenticator from accepting rogue members into the community; access control based on location information, which could use information regarding the location of a member as conditions within policies; intrusion detection, which could be used to monitor attack attempts, log the information and generate alerts; and finally, encrypted communication, which could be used among the members of the community if necessary, depending on the risk context or other specific characteristics of a mission. More on security on *ad-hoc* networks can be found in [9].

B. 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, according to the community specification. 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 the interaction between roles in a community by defining a finite state machine, where arrows represent the generation of events and states are actions that represent steps of the protocol. We only focus on modeling interactions between management roles, but the same approach can be used for application-specific interactions.

For example, the finite state machine in Figure 3 illustrates an interaction protocol in a community that supports the three management roles described previously: 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 will trigger the authentication step; if the node is successfully authenticated, a list of potential roles for assignment must be selected; after that, constraints must be checked; the node is then assigned to the roles that satisfy both the set of maximum cardinality and the set of separation of duty constraints, provided the node possesses the required capabilities for a specific role (the assignment policies will be described in detail in the next section); at this point, if the minimum cardinality constraints specified by the community are satisfied, the community changes to the state *"established"*, otherwise it remains in the state *"broadcasting/waiting"*, where it waits for more nodes to reply. The sequence of policies that represents these steps in the protocol is presented in Figure 4.

The protocol illustrated in Figure 3 has 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 and show how this protocol can be changed by modifying policies. Each step in the protocol represents 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. Typically when an event triggers a policy it contains parameters that can be used in the condition or in the action of the policy. When the events used to trigger the next step are generated by policies, they must also specify such parameters. To simplify the presentation, the parameters are omitted in the events generated by the policies in Figure 4.

The chaining of policies provides a flexible approach that allows the modification of the protocol by simply changing policies. For example, after selecting a set of potential roles for assignment, we can completely skip the validation of maximum cardinality and separation of duty constraints by using the *selected_potential_roles* event to trigger the assignment of the node to the role instead of triggering the validation of maximum cardinality and separation of duty constraints policies. Similarly, we could add a new type of constraint validation before going to the assigning step – this would require a new policy specifying the validation action to be performed, the triggering event would come from the *"selecting potential roles"* state and this step would generate an additional event that would be used as input of the *"assigning"* state. This policy would be similar to the policies illustrated above that perform the validation of maximum cardinality constraints or

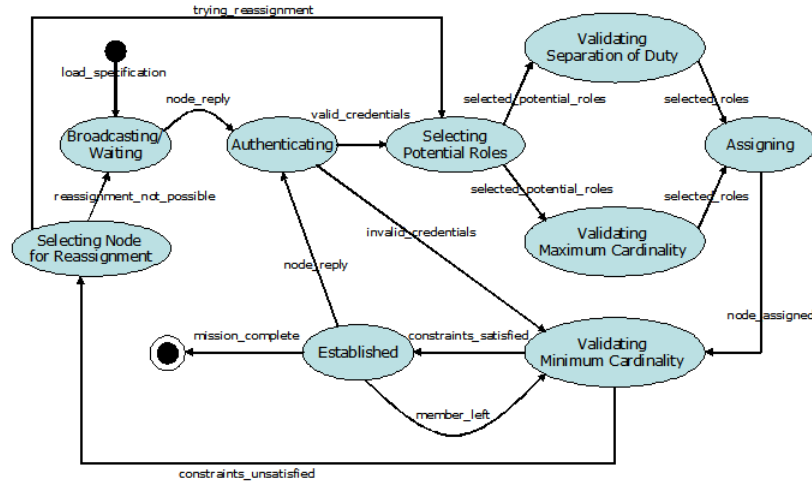

Fig. 3. Modeling community management.

```
on load_specification(specification) do
    /role/Coordinator broadcastAndWaitForReplies();

on node_reply(node, credentials) do
    /role/Authenticator authenticate(node, credentials);
    /role/Coordinator notify(valid_credentials);

on valid_credentials(node, capabilities) do
    /role/Coordinator selectPotentialRoles(node);
    /role/Coordinator notify(selected_potential_roles);

on selected_potential_roles(roles)
    /role/Coordinator validateSeparationConstraints(roles);
    /role/Coordinator notify(roles_separation);

on selected_potential_roles(roles)
    /role/Coordinator validateMaximumConstraints(roles);
    /role/Coordinator notify(roles_maximum);

on roles_separation(rs) + roles_maximum(rm)
    if hasCapabilities(node, <cap>) and
       isRoleValid(<role_name>, rs + rm) then
        /role/<role_name> assign(node);
        /role/Coordinator notify(node_assigned);

on node_assigned()
    /role/Coordinator validateMinimumConstraints();
    /role/Coordinator notify(constraints_satisfied);

on constraints_satisfied()
    /role/* establishCommunity();
```

Fig. 4. Policies for community management protocol transitions.

the validation of separation of duty constraints.

We are therefore specifying the community management in terms of policies that perform steps in the protocol; the policies generate new events that trigger other policies that perform other steps; in turn, these policies also generate new events and so on. This is similar to the approach used in PDL [10], where internal events are used to link the execution of policies. There however, only local events were considered, while here events can be sent to remote nodes performing a given role. The flexibility of our approach is even clearer when we consider the addition of entirely new management roles to the community, with a whole new set of management policies. These can be used then to enhance the protocol, by adding new management steps.

The duality between the finite state machine and the policy representation would facilitate automatic generation of the set of policies from the corresponding FSM. However, this is future work.

Based on the set of management roles outlined above, the next sections give examples of how the community could evolve during runtime, and how new members join and leave the community.

C. Authentication and role assignment

Initially, the node assigned to the coordinator role is responsible for broadcasting messages advertising the community to nearby nodes. This task, as everything else, is specified in terms of a policy. An event, such as a clock tick every 30 or 60 seconds, is used to trigger the broadcasting action. Additionally, the broadcast could be sent over a special channel frequency decided and controlled by policies. The advertisement message contains a reply address, which corresponds to the address of the node performing the authenticator role (which can be the same node performing the coordinator role, or some other node). When a nearby node replies to the advertisement, its reply message typically contains a description of the capabilities of the remote entity and its credentials. The authenticator needs to validate the credentials of any node before it is allowed to join the community.

The credentials of a node are validated using the public-keys of the relevant certification authorities (CAs) for a given community. For example, in our reconnaissance scenario involving US and UK troops, typical relevant CAs would be US DoD and UK MoD. These public-keys were previously loaded into the community specification.

If the discovered node is successfully authenticated, an event is sent to the coordinator. This event may trigger one or more

assignment policies. Assignment policies are one of the types of obligation policies enforced by the coordinator role. The roles to which a new member can be assigned are application-specific roles (surveyor, aggregator, etc), management roles or both. The coordinator may also delegate management tasks to other nodes, assigning these nodes to the respective roles.

An assignment policy defines the assignment of a node to a role as a function of the capabilities the node possesses and the requirements of the role. A typical assignment policy is shown below. It assigns the discovered node to a surveyor role provided that the node possesses the required capabilities for performing that role (in this case, the node must provide the *video* capability).

```
on valid_credentials(node, credentials, capabilities) do
    if hasCapabilities(capabilities, [video]) then
        /role/Surveyor assign(node);
```

Note that this policy is simpler than the assignment policy illustrate in Figure 4 (sixth policy). Here, we are assuming that the assignment is triggered just after the authentication step, while in Figure 4 the assignment is triggered by a correlation of events after the two steps for validation of constraints are completed. The event that triggers the policy may change according to the management protocol and the steps used. However, the assignment action must be conditional on the node possessing the capabilities required by the role.

After a node is assigned to a role the relevant obligation and authorization policies it must enforce are downloaded to it.

D. Membership management

For membership management we have implemented a policy that causes members of the community to periodically notify the membership manager that they are still present. This is required due to the high mobility of entities in ad-hoc networks and in pervasive environments in general. The community must be able to deal with nodes which move out of communication range, run out of battery or disconnect. In the situation that a member does not renew its membership within a given time period, the entity is not considered a member anymore and the membership manager informs the other members that a node has left. This also causes the constraints of the community to be reevaluated by the coordinator – the departure of a member may for example violate the minimum cardinality constraints, which would require the community to be dissolved.

IV. Implementation

The work on dynamic communities relies on the infrastructure provided by self-managed cells (SMC) to represent autonomous entities and the Ponder2 policy framework for the specification of policies [3]. An SMC consists of hardware and software components which form an autonomous administrative domain. Components include resources (such as sensors and cameras), devices (such as PDAs, Gumstix and mobile phones) as well as software services and components within those devices.

An SMC comprises a dynamic set of management services that are integrated through a common publish/subscribe *event bus*. Of particular importance is the SMC *policy service*, which is based on the Ponder2[3] policy framework. Ponder2 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. Ponder2 provides an object management system where *managed objects* (*MOs*) are kept in a domain structure similar to a hierarchical namespace. Policies specify obligations or authorizations in terms of managed objects, which can be internal SMC resources, adapters for external services or policies themselves. In addition, a *discovery service* is used to detect new devices which are capable of interacting with the SMC, such as other SMCs in the vicinity.

The SMC event bus, the policy and the discovery services support the basic functionality of a community and enable policy actions to be performed in response to context changes. We assume the nodes assigned to roles within a community are SMCs.

Although Ponder2 comes with a set of built-in basic managed objects used to represent common concepts (*Event MO*, *Domain MO*, *ObligationPolicy MO*, etc), a set of additional managed objects were required in order to support the concepts described in this paper. For example, a *Community MO* and a *Role MO* were created to represent the community specification schema described in Section II. The community object defines the roles that are part of the community and the policies associated to each role. An instance of a Community MO is kept in the domain structure of the node that will be playing the coordinator role. When new members are assigned to roles, the members will receive part of the policies contained in the community specification, according to the roles they will be playing. The self-managed cells, Ponder2 and the framework described in this paper were implemented in Java.

In order to support a minimum set of management and security requirements, most of the functionality required by the coordinator, the authenticator and the membership manager roles was implemented. Therefore we could test our system using real policies which specify actions using these management roles. For example, the authentication of new members performed by the authenticator role was implemented using *X.509 digital certificates* using the *java.security* package. The coordinator already implements the management procedures required to perform the community broadcast, assignment of new members to one or more roles, and to load the respective Ponder2 obligation and authorization policies onto the remote node.

The dynamic validation of constraints and reassignment of roles if the constraints are not satisfied still remain to be implemented. Additionally, the representation of policies in terms of the finite state machine as illustrated in Section III-B

[3]http://www.ponder2.net

requires the ability of event correlation, in the sense that we have states (policies) that must be triggered by a combination of two or more events (such as the assigning step, in Figure 3, which is triggered by multiple events generated in the validation of constraints). The basic event service in the SMC does not support event correlation, but an event correlator can be implemented as a role which receives simple events, performs event correlation and generates new events representing the correlation of other events. This has been implemented in previous test systems but not yet integrated into the Ponder2 system.

V. Future work: cross-community interactions

The ability to cater for communities that interact with other communities will be required to extend the dynamic community approach to larger scale scenarios. We could have for example *federation* representing peer relationships between communities, where one rescue team could provide data (such as surveillance data, maps, etc.) or services and resources that it possesses to another rescue team. This is illustrated in Figure 5.

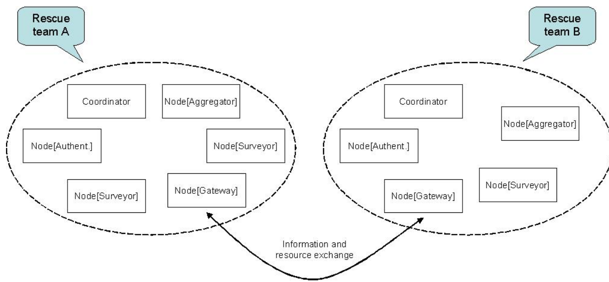

Fig. 5. Peer federation of communities.

Alternatively, we could think of *composition* of communities, where a rescue team could have a medical team as one of its components, that would be a community itself. This is illustrated in Figure 6. In this case, the rescue team could have the ability to bootstrap or dissolve the medical team community (in fact, the sub-community may be part of the specification of the outer community). In both peer federations and compositions, the interactions between communities could be facilitated by a node performing the role of *gateway* in each community.

Composition would allow a sub-community to be a member of an outer community. Composition would also allow inner and outer communities to share information, however typically an inner community would be encapsulated in the outer community and the outer would have a tighter control over the inner community compared to the control that peer federated communities have over each other. The outer community 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 to larger environment, with self-managed, encapsulated communities, but future work still has to investigate the abstractions required to support cross-community interactions.

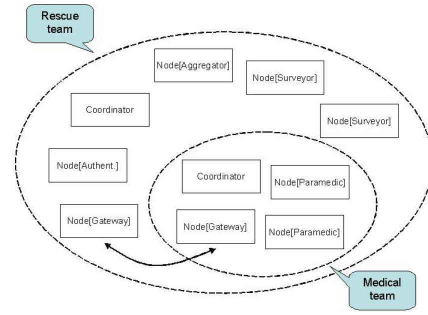

Fig. 6. Composition of communities.

VI. Related work

There are a number of frameworks for realizing pervasive spaces [11], [12]. In general, these frameworks tend to focus on spaces of relatively fixed size such as a room or a house. In addition, whilst much of the literature focuses on the architecture of pervasive spaces and their supporting services, less attention is paid to the interactions and collaboration between such spaces. In contrast, our work not only supports communities to discover new members and evolve internally, but they are also expected to support cross-community interactions. In cross-community interactions not only a single entity but an entire community can become a member of the community. By encapsulating the management of communities and hiding its complexity, we can support large-scale applications providing collaborations between entire communities.

The industrial work on autonomic computing, led primarily by IBM [13] but also addressed by Motorola [14] and HP [15], 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, for example in communities of ad-hoc unmanned autonomous vehicles. The control-loop performed by self-managed cells 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.

Research on policies has been active for several years, especially regarding policies for network and systems management. Some examples include PCIM [16], PDL [17] and PMAC [18]. Although they use similar event-condition-action rules for encoding adaptation actions, these approaches are targeted for management of large-scale and networked systems only, and do not scale-down for managing small devices.

Mobile UNITY [19] 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.

Finally, the management of dynamic communities will be enhanced as we include more mechanisms beyond the basic set of security and management procedures we already have. Threshold cryptography [9] for preventing a compromised authenticator from accepting rogue members and intrusion detection [20] for monitoring potential risks and attacks to the

community are among the additional mechanisms mentioned in Section III-A, 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.

VII. Concluding remarks

This paper presented an approach for specification of policy-based dynamic communities of autonomous entities that need to collaborate in order to accomplish their goals. We have used self-managed cells as the implementation platform for the autonomous entities that constitute our dynamic communities. Based on the resources and devices an SMC possesses, it may be assigned to different roles in the community and thus dynamically receive the policies that define the role.

More important though is our focus on management and security of dynamic communities. We specified a basic set of management roles that provide the minimum management and security functionality for dynamic communities. Based on the combination of policies provided by such roles, we can specify a different range of management protocols. By extending the set of management and security roles, one can enhance the community management protocol in a flexible manner. The protocol is represented by the chaining of policies provided by the management roles, and it can be illustrated by a finite state machine. Just by rewriting the policies, one can dynamically include or remove steps to or from the protocol. This cater for an extensible infrastructure for management and security of dynamic communities, where new management procedures or even entire roles can be added, according to the management and security requirements of each community.

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 heron. The authors also wish to thank Eskindir Asmare for his ideas on defining the reconnaissance scenario used in this paper.

References

[1] S. L. Keoh, E. Lupu, and M. Sloman, "Peace: A policy-based establishment of ad-hoc communities," in *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 386–395.

[2] M. Sloman and E. Lupu, "Security and management policy specification," *IEEE Network*, vol. 16, no. 2, pp. 10–19, Mar.-Apr. 2002.

[3] E. Lupu, N. Dulay, M. Sloman, J. Sventek, S. Heeps, S. Strowes, K. Twidle, S.-L. Keoh, and A. Schaeffer-Filho, "AMUSE: autonomic management of ubiquitous systems for e-health," *J. Concurrency and Computation: Practice and Experience, John Wiley*, no. Special Issue: Selected Papers from the 2005 U.K. e-Science All Hands Meeting (AHM2005), May 2007.

[4] N. Damianou *et al.*, "The Ponder policy specification language," in *Proc. IEEE Workshop on Policies for Distributed Systems and Networks*. Bristol, U.K.: IEEE-CS, Jan 2001, pp. 18–39.

[5] M. Sloman, "Policy driven management for distributed systems," *Journal of Network and Systems Management*, vol. 4, no. 2, pp. 333–360, 1994.

[6] E. Asmare, N. Dulay, H. Kim, E. Lupu, and M. Sloman, "A management architecture and mission specification for unmanned autonomous vehicles," in *1st SEAS DTC Technical Conference*, Edinburgh, Scotland, 2006.

[7] E. Asmare, N. Dulay, E. Lupu, M. Sloman, S. Calo, and J. Lobo, "Secure dynamic community establishment in coalitions," in *MILCOM*, Orlando, FL, 2007.

[8] R. Sandhu, "Rationale for the rbac96 family of access control models," in *RBAC '95: Proceedings of the first ACM Workshop on Role-based access control*. New York, NY, USA: ACM Press, 1996, p. 9.

[9] L. Zhou and Z. Haas, "Securing ad hoc networks," Cornell University, Ithaca, NY, USA, Tech. Rep., 1999.

[10] R. Bhatia, J. Lobo, and M. Kohli, "Policy evaluation for network management," in *INFOCOM*. Tel-Aviv, Israel: IEEE CS-Press, March 2000, pp. 1107–1116.

[11] M. Roman *et al.*, "A middleware infrastructure for active spaces," *IEEE Pervasive Computing*, vol. 1, no. 4, pp. 74–83, Oct.-Dec. 2002.

[12] D. Garlan, D. P. Siewiorek, A. Smailagic, and P. Steenkiste, "Project Aura: toward distraction-free pervasive computing," *IEEE Pervasive Computing*, vol. 1, no. 2, pp. 22–31, Apr.-Jun. 2002.

[13] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, Jan 2003.

[14] J. Strassner, N. Agoulmine, and E. Lehtihet, "Focale a novel autonomic networking architecture," in *Latin American Autonomic Computing Symposium*, Campo Grande, MS, Brazil, July 2006.

[15] HP, "Hp utility data center: Enabling enhanced datacenter agility," http://www.hp.com/large/globalsolutions/ae/pdfs/udc enabling.pdf, May 2003.

[16] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen, "Policy core information model, version 1 specification. request for comments 3060, network working group," 2001, available at: http://www.ietf.org/rfc/rfc3060.txt.

[17] J. Lobo, R. Bhatia, and S. Naqvi, "A policy description language," in *Proceedings of the 16th National Conference on Artificial Intelligence*, Orlando, FL, July 1999, pp. 291 – 298.

[18] D. Agrawal, S. Calo, J. Giles, K.-W. Lee, and D. Verma, "Policy management for networked systems and applications," in *Proceedings of the 9th IFIP IEEE International Symposium on Integrated Network Management*. Nice, France: IEEE CS-Press, May 2005, pp. 455 – 468.

[19] G.-C. Roman and J. Payton, "Mobile unity schemas for agent coordination," pp. 126–150, March 2003.

[20] T. F. Lunt, "A survey of intrusion detection techniques," *Computers and Security*, vol. 12, no. 4, pp. 405–418, 1993.

View publication stats