A Framework for User-centric Autonomic Management

João Paulo Barraca, and Rui L. Aguiar, *Senior Member, IEEE*

***Abstract*— This paper presents a novel autonomic management framework for user-centric networks. The framework assumes users are directly involved, or have direct interest, on their communication system. Therefore it can place user-related constrains on the behavior of the system. Groups of interest, formed either implicitly or explicitly by users, are considered inside the community concept, which the framework supports. For the realization of this concept it is required a new knowledge and logic layer, which we describe in this work. This framework has been instantiated and tested in a small environment, and provided social aware service access authorization. The results showed the added flexibility of our framework, and its ability to integrate diverse user requirements.**

***Index Terms*—autonomic management, policy-based management, community networks**

I. Introduction

Networks are increasingly becoming user-centric: they cater for user wishes, retain user preferences, and even behave according with user past behavior. Recent community network scenarios have exacerbated this trend – where users become also part of the network provision, with direct or indirect control over the traffic that flows on the network. The *Build Your Own Network* is a concept that is now reaching into market, albeit in a very simplistic way.

Mesh networks and ad-hoc networks are two well-known scenarios where such networks grow into reasonable sizes. For the operation of these networks, leading with the uncertainties of the wireless media and with the moods of the users (turning their equipment on, activating the 3G interface, etc…), self-organization is an essential characteristic. In fact, the challenges now are no longer on the specific technologies to use. Multiple works have already been done both on transport protocols, on wireless spectrum management, on incentives for cooperation [1], [2], [3], [4]. The key challenge is how do we make these networks cooperate amongst themselves.

A network must self-organize in the way it reacts to large topology changes naturally and evolve together with the user base. In our early vision [5], we proposed that self-organization requirements and incentives were entangled and could be jointly managed. Users may feel attracted to participate in spontaneous networks, if they have some control over their participation. Such a vision imposes a radical departure on traditional management challenges. On one hand, the system needs to have autonomic features, being able to operate without human intervention, and interacting with neighbors with essentially zero-hassle to the network owner. On the other hand, these systems are essentially centered on the user interests and desires.

Our work has been addressing this dichotomy. We developed [6] a management framework based on the notion of community of users with similar context. This framework should allow users to express their interest at a very high level, formulating specific policies of what is acceptable behavior (for them). Users can then self-associate in communities of interests, which will be self-regulated. So in these groups we see a balance where users give resources (e.g. in the form of bandwidth, and storage, or even expertise) to others, in the exchange of a desired service (may be similar or a different service), while contributing to a common goal. We see that this would greatly benefit the sprout of community neighborhood networks as proposed in [5].

Nevertheless, we realized that these concepts were not possible to deploy without resort to a high degree of intelligence in the management actions. In fact, our self-organization goals can only be fulfilled if systems self-govern themselves under the external constrains imposed by the community it wants to belong to – effectively if we deploy an autonomic management system, able to extract information on network capabilities, resort to ontologies to agree on community building, and perform reasoning and learning mechanisms to improve overall performance, and the degree of "belonging" to a community.

In section II we will formally describe the concepts associated with our autonomic management proposal. In Section III we describe the architecture developed for our software, focusing in the Logic and Knowledge aspects of this. Section IV briefly presents our prototype and some of our results. Section V then presents a brief overview of previous work in the area, and highlights the advantages of our proposal. Section VI concludes the paper.

II. User-centric Autonomic Management

We base our user-centric management concepts on the establishment of common interests between groups of users (e.g. basic connectivity). For that we rely on the community concept. Formally, a community is composed by a knowledge context that is shared by different entities interacting. Entities

João Paulo Barraca is with Instituto de Telecomunicações – Aveiro and with Universidade de Aveiro, Aveiro, Portugal (email jpbarraca@av.it.pt).

Rui L. Aguiar is with Instituto de Telecomunicações – Aveiro and with Universidade de Aveiro, Aveiro, Portugal (email ruilaa@ua.pt).

can belong to several communities at the same time; so we will focus this paper on the situation that for *Community C*, the existing *Knowledge* K_e which represent the knowledge of each entity, and a set of *Entities E*. A community includes all member entities and their knowledge. That is, a community is simply a subset of the universe of entities and their directly related knowledge.

Elemental objects, with properties and relations are stated in a well-known ontology (as proposed in a previous work [7]), can be arranged in multiple ways, according to the purpose and interaction history, forming complex objects. These elemental objects can be of many types; still all share some common content in the form:

Object : = ParentType::ParentName | Type::Name

Ultimately the concatenation of the object basic properties can be used as a unique identifier (by means of a digest function such as SHA-1) over all the tuples <parent,type::name>.

Entities are able to operate over elemental or complex objects composing the shared knowledge. As interaction takes place, entities will shape knowledge by manipulating its objects. The result is that each object will reflect part of the community history. Each different community will have its unique knowledge, and no two communities share the same exact knowledge.

When interacting, entities may or may not create additional shared knowledge depending of: a) entity wishes, b) interaction nature, c) current knowledge. Entities may refuse to add new knowledge for a specific interaction or may be refused, depending on the information present in the knowledge base. All entities are expected to comply with current knowledge. In order to ensure this rule, other entities can monitor how knowledge evolves, detect violations, and act accordingly. This is facilitated in the wireless medium due to the inherent eavesdropping capabilities not present in many other technologies.

Entities are the actors of a *Community* and represent both systems and users. We opted for such categorization because, at a logical level, when building the ontology they presented many similarities. Other reason is that we wanted to easily support enterprises and other groups as *Entities*. The only constraint imposed is that systems must have a relation to some other *Entity* of a different type acting as an owner. *Entities* may have their capabilities expressed in terms of *Resources, Devices* and *Services*, which may be accessed by others. All (devices, services and resources) may provide several *Attribute* objects, which can be expressed in the form:

Attribute := Object | Name | Value

Actual service and resource consumption is out of the scope of our framework and is executed in the same way it was before by the specific services. At the same time we introduce the means to integrate concepts of user level services as presented by authors such as [4]. Services can be real (e.g. FTP, HTTP) or meta-services (e.g. FileSharing, InternetAccess), which our framework correctly processes. The *Attribute* Name is the identification of the *Attribute* inside the parent object (e.g *NetworkDevice_MediaType* represents the type of medium an interface supports). This is achieved by means of specialization of the root objects through indication of the sub-type. *Attribute*, *Service*, *Device* and *Condition* objects support subtypes (as can be seen in the representation of the *Condition* objects). We explicitly rely on specialization rather than inheritance in order to reduce management complexity.

Rule objects are complex objects defining the action to take when a given condition is met. That is, when the condition is valid, an *Attribute* is set. These objects enable the use of an Event-Condition-Action (ECA) [8] approach, allowing the policy to react to changes in the environment in a reactive manner.

Rule := Condition | Attribute

Role objects are complex objects defining an interaction behavior, which can be associated to *Entities*. They grant access to *Resources, Services,* and other *Roles* (permissions), while also impose constraints on behavior (obligations). *Roles* are a specialization of the *Object* type and add the following elements:

Role := Conditions | Attributes | Rules.

Well known *Attributes* can help describe the *Role (e.g. CreationDate)*, while an optional list states invariable behavioral aspects (e.g. wireless channel or bandwidth sharing). *Condition* objects state limitations to the relations that can be created based to the *Role.* They take the following form:

Condition := Type | Object | Attribute | Operator | Value

Type identifies if the *Condition* is *mandatory* (must be true) or *sufficient* (its result is enough to decide), and constitutes a case of specialization inside the *Condition* object. *Object* identifies the object to require validation, *Value* is an arbitrary block of information relevant to the *Attribute*, and *Operator* is a comparison operator (e.g. EqualOrGreaterThan, Different, or Equal) enhanced with the *Exists* operator. With proper definition of *Condition* objects in *Roles* it is possible to enable duty-separation, or cardinality over any *Attribute* present.

Creating a relation between *Entities* and *Roles (or other Objects)* involves issuing a *Delegation* object, effectively creating a relation between the two entities (issuer and receiver) and the object. *Delegations* are comprised by the following data:

D := Issuer | Receiver | Object | Counter | Flags | Dates | Sig

Delegation D is cryptographically signed (Sig) by the issuer *A* and may allow the receiver *B* to further delegate it (or not). We assume a PKI is available to nodes if security requirements require secure *Delegation* issuing. However for the sake of simplicity, and due the large number of standard solutions, the PKI it is out of scope for this work. The *Counter* field states the permission to further delegate (when greater than 0). Issuers must provide *Delegation* object with a value lesser than the value in their *Delegation* until 0 is reached. The

flags field states if the *Delegation* is revoked. Other fields like *startDate*, *expireDate* and *updateDate* are present to restrict *Delegation* duration, or refresh its status. Considering that *A* has a delegation with right to delegate, it can create *Delegation* D_{br} for *B* over *Role R,* with $C_{Rs}(B)$ *and* $C_{Rm}(B)$ being the list of *Conditions* of *R* evaluated for *Entity B* if:

$$\left[\left(\exists s \in CRs(B) : s = true\right) \cup \left(\forall m \in CRm(B) : m = true\right)\right] \neq \varnothing$$

Delegating a *Role R* to an *Entity B* effectively entitles, *B* to act according to the premises stated in *R*. It also gives *B* permissions to access *Services* and *Resources*, otherwise restricted to it. That is, delegations state a trust relation between issuer and subject for a given object, at the same time associating obligations and providing rights. Membership is stated by the existence in the knowledge base of a delegation of the "Member" *Role*. Besides this, each community can build their set of *Roles* and evolve freely.

Also important is the verification of the *Delegation* objects present, as they may allow access to resources which otherwise would not be granted. Proper validation of a *Delegation*, thus require verification of the *Delegation* chain up to the community creator (both logical correction, and cryptographic signatures). It's up to the *Entities* to decide when, and which chain to verify, constituting a tradeoff between security and network overhead.

III. System Architecture

This section describes the software architecture required to support an autonomic management system able to support the user-centric view detailed in the previous sections. Our architecture is designed as a generic framework for managing information of entities forming groups or interests. We intended to design a system with a generic core, able to provide policy based self-management mechanisms to a multitude of scenarios. In this aspect, except the Interface Layer, all modules are independent of the actual instantiation of the community. As an example, it could be a set of hosts and employees using it as a distributed access control and management tool. Still, critical design solutions are oriented towards spontaneous networks of low power devices such as ad-hoc and community mesh networks.

Our architecture is divided in four layers, namely: logic, knowledge, interfaces and communications. The division aims at building a modular system that while capable of supporting a multitude of scenarios, still remains efficient to be applied in low power devices. A representation of this architecture is depicted in Figure 1.

A. Logic Layer

The core components of our system are the ones responsible for managing information and reasoning. Without these components it is impossible to devise a system capable of processing (distributed) policies. We identify the need for three core components: *Operations*, *Reasoning Engine* and *Monitoring*.

The *Operations* component is required in order to actually manage content. That is, create, change or delete information and compose elements into more complex and useful objects. Most objects will be created due to human interaction when users input their preferences or explicitly create a group. Also, it will operate over the knowledge and perform simplifications, without discarding knowledge or distorting the resulting obligations and permissions.

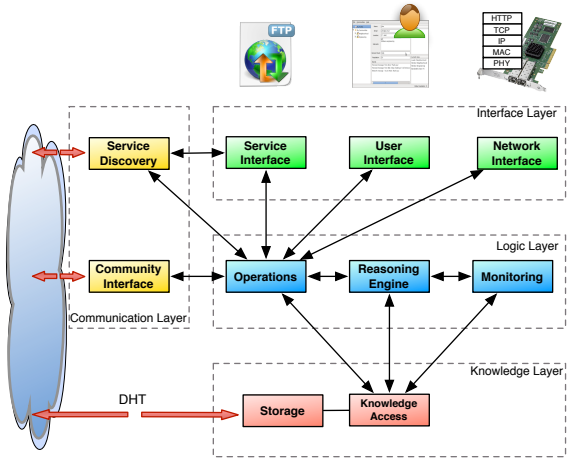

Figure 1 - System Architecture of the management system

Simplifications to knowledge are required as a form of purging the knowledge base from duplicate or invalid objects. Moreover, groups can be created, destroyed or, more importantly, merged if logically similar. This is all handled by the *Operations* component.

Connected to the *Operations* component is the *Reasoning* component. While the first actively manipulates knowledge, the *Reasoning* component only processes existing objects and outputs decisions. No alteration of knowledge is done by the *Reasoning* component, as it behaves as the engine processing logical objects. In the same manner, other components do not perform any logical inference, delegating this task to the *Reasoning* module.

Objects can be manipulated by *Entities* according to their wishes and access limitations imposed. However it can be possible to perform malicious changes, which go beyond the accesses or violate obligations. Moreover, community systems are dynamic. Delegations previously issued may need to be revoked if the destination user doesn't meet the conditions for the *Role*. The Monitoring module pools the knowledge and the delegations issued and validate their correctness.

B. Knowledge Layer

Core components can manipulate objects and reason over the logic created. Still, these objects are not persistent and only available to the current process. The knowledge layer provides means to make the knowledge created persistent and disseminate it to all agents. In some way it can be considered as an out-of-band communication plane as content added to the knowledge layer by *Entity A* will affect *Entity B* if they share the same knowledge instance.

This layer is composed by two components: *KnowledgeAccess* and *Storage*. The first provides an interface

enabling all actors sharing the same knowledge base to access and operate over the existing objects. Effectively the *KnowledgeAccess* component abstracts location of knowledge elements, allowing transparent access, independently of the actual physical location of an information object. While knowledge is available to all actors, the *KnowledgeAccess* component will also limit access (using the *Reasoning* component) based on the existing objects and the *Roles* of the accessing peer.

Data in our model is inherently relational, and standard SQL databases are appropriate to provide the functionalities of the Storage component. However, this solution is limited to scenarios where all members can reach a common SQL server. Distributed scenarios, such as the case of Wireless Mesh Networks, may require the deployment of a more distributed Storage, mainly due to issues related to overhead and reliability. This is the approach we followed in our prototype, as it enables spontaneous ad-hoc networks, and the *Build Your Own Network* concept. In particular we focused in developing a Storage layer supported by a DHT and following P2P principles, allowing data dissemination and retrieval among an undefined number of participants.

C. Communication Layer

With the previous layers, information objects can be processed, made persistent, and disseminated transparently among participants of a Community. However this is not enough as there is no direct communication between entities.

The Communication Layer provides mechanisms, which allow information to be discovered and exchanged between entities. The *ServiceDiscovery* component advertises key information regarding the current *Entity* capabilities and active communities. In a previous preliminary work [9] we described and evaluated an instantiation of this discovery component. It relies on an improvement over mSLP suited for mesh and ad-hoc scenarios. It makes possible to discover entities, services and communities in the neighborhood or through direct contact as specified by an URI. The *ServiceDiscovery* component effectively increases the reach of our system in order to process location information (neighborhood based).

The Community Interface provides an interface allowing direct exchange of information between instances running in different nodes. It is required in order to support mechanisms such as: requesting and issuing delegations, probing capabilities to validate roles, or exchange knowledge private to the entities.

D. Interface Layer

The Interface Layer is connected to the other layers and provides additional objects to the knowledge space. Each interface enriches the knowledge space by providing elemental objects, thus shaping the future knowledge created. If a node has no support for a particular object supported by an unknown interface, it will just be incapable of processing part of the knowledge. Ultimately its absence may limit enrolling in particular roles or limit access to a service. The interface components available at each node will thus tailor the use and capabilities of the system. In our specific case we aim at user centric autonomic behavior in wireless mesh networks. For this purpose we defined three interfaces: *User*, *Services* and *Network*. Other scenarios can require a different set of interfaces. These interfaces have full responsibility for interfacing services, users or systems to the internal language of the management system (and vice-versa) and should be as expressive as possible.

The *UserInterface* is vital as it allows users to express their interests to the system by creating a *persona* in the form of an *Entity* object. By using the User Interface, users explore the entire knowledge base available in the communities they belong to. In particular they are able to discover neighbor users (both in terms of location and in terms of interest) and interact with them, or create new communities. More importantly they are able to build a social network (by issuing *Delegation* objects), which can later affect network fabric.

The *ServiceInterface* provides a bridge between services existing at nodes (e.g. FTP, HTTP, BitTorrent) and the management system. Interaction is bidirectional: services can use the management system to condition or limit access (e.g. only members can access FTP with full speed); the system will take in consideration the existence of a service when reasoning over an object (e.g. Membership requires service BitTorrent). It should be noticed that Users can always express their will and may allow access to their services by issuing a *Delegation*.

The final interface, and of utmost importance, is the *NetworkInterface*. This component effectively bridges different networking technologies by providing an abstraction to the management system, while enforcing rules to the host system. It allows generalized cross layer management, using user centric policies. It also exports native network interfaces, their capabilities and attributes to the system, and allows manipulation of the network stack state and attributes (e.g. changing wireless channel, or SSID). To further enrich the *NetworkInterface*, networking functions such as Routing, Forwarding, Shaping, and Packet Filtering are mapped to *NetworkService* objects, which are a sub-type of the Service object. This allows communities to enforce the configuration of its participants, and restrict access to higher *Roles* based on the value of almost any network attribute.

IV. Prototype and Results Obtained

A prototype implementation of the management system we propose was developed under the Linux environment, and using the C++ language. While used to validate the architecture we devised, it aims to support user centric autonomic network management in urban Wireless Mesh Networks (WMN). All modules were implemented in our prototype except the *Monitoring* module, which we expect to evaluate in a future work.

Because the target scenario of our prototype (WMN) is comprised of many low power embedded devices, we implemented most of the system using C++ on a Linux environment. Other characteristic of the WMN scenario is that

neighbor users living in the same street or city block can form groups of interest spontaneously, and in an uncontrolled manner. Moreover, existence of a reliable fixed infrastructure is not guaranteed, which invalidates centralized storage approaches, thus imposing the use of a distributed knowledge base.

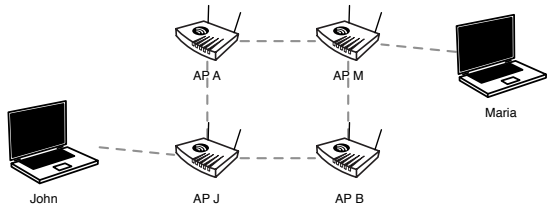

Figure 2 - Schematic of our test topology

Several hosts, emulating the topology depicted in Figure 2, composed the testbed used to evaluate the prototype implementation. All hosts use Ubuntu 11.04 as their operating system. The hardware is comprised either by actual PCs (Maria, John), or by virtualized hosts. Access Points have their memory limited to 64MB. All equipments run an instance of the management system and are allowed to write to a small permanent storage (which keeps local objects and configuration across different runs). No wireless medium is emulated in the testbed. While the wireless medium is relevant for our scenario, it is not that relevant to validate the concept and its logical effectiveness. Future work will focus in evaluating the impact of the prototype in terms of additional overhead, and efficiency over a real wireless testbed, which is currently being deployed.

Considering the distributed nature of WMN, the Storage module is comprised of a Distributed Hash Table (DHT), running at a subset of the nodes (in this case, at the APs). Information objects are automatically replicated to nodes, and accessed in a completely transparent manner. This is actually the purest form of a spontaneous, autonomic community, sharing a common knowledge base. The DHT overlay follows some principles of Kademlia [11], and each node can allocate a configurable amount of space to storage. Objects are *put* to the storage, and are identified by a digest computed over the concatenation of *name*, *type* and *parent*. Caching much increases storage performance during *get* operations. We still follow a write-through approach to *put* operations. Automatic object locking, and *put* serialization is supported by designating master peers (determined by the *XOR* metric), which are responsible for coordinating *put* operations to an object, and invalidate other peers cache.

In a simple authorization test case Maria discovers a user community where John provides a FTP service. When Maria accesses John FTP service, the DHT is searched for a valid delegation with key *<Maria, Member, community>,* and the access is authorized or denied based on the existence of a Member *Delegation*. The sequence of modules invoked is depicted in Figure 3. This approach is valid for almost all applications making use of the Linux PAM library. It should be noticed that, Maria is not running any additional software besides the FTP client and our management software.

In this case, and when Maria provides her username, in the background at John laptop, the FTP server (through the PAM infrastructure) queries the *ServiceInterface* module for authorization.

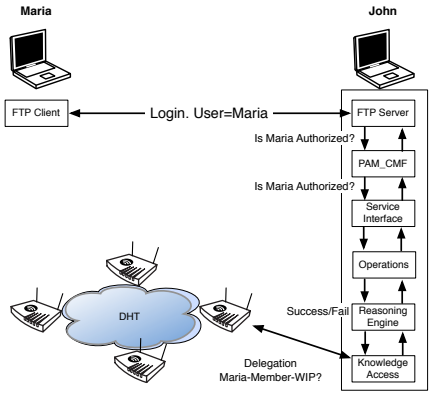

Figure 3 - Schematic of the agents and messages exchanged while authorizing an FTP session

Considering a community named WIP, Maria has no delegation in this community and therefore cannot access WIP resources. When she tries to access the FTP service she will get a negative answer. Figure 4 depicts the messages exchanged by the PAM module and the management system running at John laptop. In this case the response can be obtained, from the reasoning logic, after 415*ms*. During this time, other queries could be made in parallel as the delay is related to the process of searching the DHT for an object, until all relevant nodes provide a negative answer.

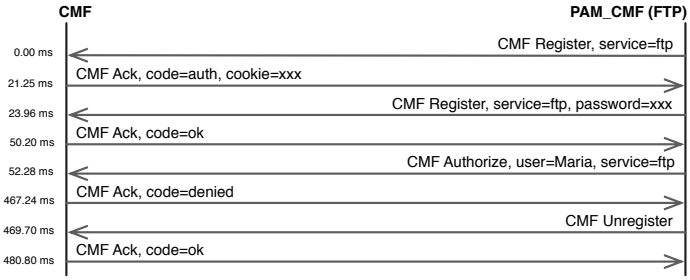

Figure 4 - Messages related to service registration and user verification, with unsuccessful authorization, captured by a monitoring application.

```
fcmf@cmf-maria:~$ ftp john
Connected to john.
220 cmf-john FTP server
Name (john:cmf): Maria
331
Password:
CMF: NO Delegation found for Maria as Member in WIP: DENIED
Login failed.
ftp> _
```
Figure 5 - Output of a denied FTP login

In the case of a failure accessing a service, the PAM module provides feedback to the user detailing the reason why the authorization failed. As shown in Figure **5**, the user is informed that no *Delegation* object exists for Maria in the WIP community. Therefore the Maria cannot access John FTP Server.

In our test case, AP-M creates a delegation for Maria as Member. The same FTP access process is repeated (see Figure 6), but this time a *Delegation* is found and service access is

granted (Maria can access FTP server owned by John). The message flow is similar to the previous case; with the difference in the result obtained (Authorized), and the time the query takes, which is lower. The process of searching for an existent object will always be faster than the remaining cases due to caching and automatic dissemination of information. In this case the query takes a mere 52*ms* in contrast to the 415*ms* for an unknown object.

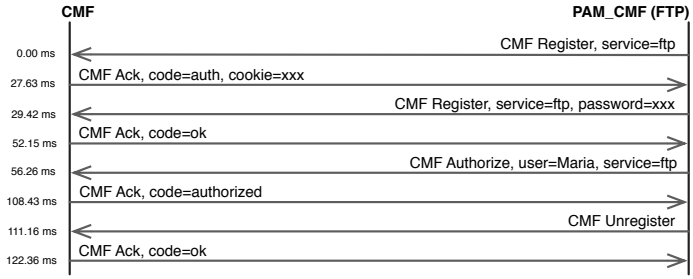

Figure 6 - Messages related to service registration and user verification, with successful authorization, captured by a monitoring application.

V. Previous Work

There is a reasonable amount of solutions in the area of management of interacting actors. Still they are either constrained to particular applications (thus too simple), or too complex to be realizable in user-centric distributed environments. In [12], the authors define a solution based on previous work on the area of Self-Managed Cells [13], targeting military scenarios for unmanned autonomous vehicles, forming an ad-hoc network. While similar to our work, the rigid scenario of coalition military cooperation influences (and simplify) key design and implementation decisions, which are not suited to a generic network management framework. PEACE [14] targets ad-hoc communities, but severely lacks flexibility and redundancy due to the importance of the coordinator role.

On the other side of problem, RFC 3060 [15] defines a policy information model to manage generic information, yet it is highly focused in scenarios with centralized data repositories such as using LDAP, and enterprise environments. PDL [16] defines a generic policy language and schema, supporting explicit delegation, and object specific rules like our scheme. However it is focused in the management of corporations and positions, and is unclear how it could be applied to a completely distributed system to manage shared knowledge. Nevertheless, it has proven to be a somewhat flexible language as it drove large attention from the research community. SOUPA [17], targets pervasive applications and correctly identifies the need of ontologies in order to cope with the heterogeneous nature of pervasive environments. While also considering ontologies, systems like [18], and [15] lack adaptability and expressiveness because of the reduced semantic relations supported. Our solution further extends SOUPA by exploring semantic reasoning over ownership, location, interactions, and other relations between objects (even social). The *ReasoningEngine* module in our system is able to manage the policy in a completely cross layer manner, and even considers social relations between users, past interactions, location, or high level interests, which could be used to promote a more comprehensive management approach.

Applied to the networking environment, CoRaL [19], proposes a language and system to manage radio spectrum. While very useful (and inline with our vision [5]), by binding spectrum management to ontology based reasoning, it is limited to a very specific application. We extend this solution by supporting cross-layer management and means of pluggable interfaces. From all these solutions, KAoS [20] seems to be the most flexible solution yet developed, with applications over many fields. However, it is highly complex and requires well-defined Distributed Directory Servers (DDS). Moreover reasoning incurs in a high number of complex queries over many objects of the DDS, thus raising scalability issues. One example of this complexity is CoRaL that explicitly aims to develop a solution aside from KAoS due to its rigid architecture, and complexity.

VI. Conclusion

In this work we present a user-centric, policy-driven, autonomic management framework, which is a natural evolution of distributed management systems for community networks. However, the complexity of such systems is such that a knowledge layer has been found to be essential for effective cooperation between distinct users.

This framework has been instantiated in an environment with multiple users. It provides users to express their interests, and can bootstrap the network, retaining overall consistency, while simultaneously providing a simpler user interface for configuration of the system. The results proved our concepts are feasible for basic service authentication, and open the prospect for more complex, cross layer management capabilities.

Future work will aim to evaluate the performance of our system over a more realistic wireless mesh testbed, and consider scenarios making use of complex cross layer, social aware, management functionalities.

References

[1] B. Cohen. “Incentives build robustness in BitTorrent”, In Workshop on Economics of Peer-to-Peer Systems, Berkeley, CA, USA, June 2003.

[2] João Paulo Barraca, Susana Sargento, Rui L. Aguiar, “The Polynomial-assisted Ad-hoc Charging Protocol”, IEEE International Symposion on Computers and Communications - ISCC2005, Cartagena, Spain, 2005.

[3] Buttyan, L. 2006. “Nash Equilibria of Packet Forwarding Strategies in Wireless Ad Hoc Networks”, IEEE Transactions on Mobile Computing 5, 5 (May. 2006), 463-476.

[4] A. Satsiou, L. Tassiulas, “A Trust-Based Exchange Framework for Multiple Services in P2P Systems”, in Proc. of the 7th IEEE International Conference on P2P Computing, Galway, Ireland, 2007.

[5] Antoniadis, P.; Le Grand, B.; Satsiou, A.; Tassiulas, L.; Aguiar, R.L.; Barraca, J.P.; Sargento, S., "Community Building over Neighborhood Wireless Mesh Networks," Technology and Society Magazine, IEEE, vol.27, no.1, pp.48-56, 2008.

[6] João Paulo Barraca, Rui L. Aguiar, "Managing Community Aware Wireless Mesh Networks", IEEE International Symposium on Computer and Communications (ISCC), Marrakech, Morocco, 6-9 July 2008

[7] João Paulo Barraca, Rui L. Aguiar, "Ontology driven Framework for Community Networking Management", International Conference on Telecommunications, St. Petersburg, Russia, 2008.

[8] Klaus R. Dittrich, Stella Gatziu, Andreas Geppert: The Active Database Management System Manifesto: A Rulebase of ADBMS Features. Lecture Notes in Computer Science 985, Springer 1995, ISBN 3-540-60365-4, pages 3-20.

[9] João Paulo Barraca, Pedro Fernandes, Susana Sargento, Rui Rocha, "An Architecture for Community Mesh Networking", IEEE International Symposium on Personal, Indoor and Mobile Radio Communications - Social and Mesh Networking Workshop, Cannes, France, 2008.

[10] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. "The cricket location-support system", In Mobile Computing and Networking, pages 32–43, 2000.

[11] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the XOR metric", in Proceedings of IPTPS'02, (Cambridge, MA), 2002.

[12] Schaeffer-Filho, A., Lupu, E., Sloman, M., Keoh, S., Lobo, J., and Calo, S. "A Role-Based Infrastructure for the Management of Dynamic Communities", In Proceedings of the 2nd international Conference on Autonomous infrastructure, Management and Security: Resilient Networks and Services (Bremen, Germany, July 01 - 03, 2008). D. Hausheer and J. Schönwälder, Eds. Lecture Notes In Computer Science, vol. 5127. Springer-Verlag, Berlin, Heidelberg, 1-14., 2008

[13] 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), 2007.

[14] Keoh, S.L.; Lupu, E.; Sloman, M., "PEACE: a policy-based establishment of ad-hoc communities" Computer Security Applications Conference, 2004. 20th Annual, vol., no., pp. 386-395, 2004

[15] RFC3060 - Policy Core Information Model -- Version 1 Specification

[16] Nicodemos Damianou, Naranker Dulay, Emil Lupu, Morris Sloman "The Ponder Policy Specification Language", Proc. Policy 2001: Workshop on Policies for Distributed Systems and Networks, Bristol, UK, 29-31 Jan. 2001, Springer-Verlag LNCS 1995, pp. 18-39, 2001

[17] Chen, H.; Perich, F.; Finin, T.; Joshi, A., "SOUPA: standard ontology for ubiquitous and pervasive applications" Mobile and Ubiquitous Systems: Networking and Services, 2004. MOBIQUITOUS 2004. The First Annual International Conference on, vol., no., pp. 258-267, 2004

[18] A. Roy, S. K. D. Bhaumik, A. Bhattacharya, K. Basu, D. J. Cook, and S. K. Das. "Location aware resource management in smart homes", In First IEEE International Conference on Pervasive Computing and Communications (PerCom'03), 2003.

[19] Elenius, D.; Denker, G.; Stehr, M.-O.; Senanayake, R.; Talcott, C.; Wilkins, D., "CoRaL--Policy Language and Reasoning Techniques for Spectrum Policies", Eighth IEEE International Workshop on Policies for Distributed Systems and Networks, 2007. POLICY '07, pp.261-265, 2007.

[20] Uszok, A.; Bradshaw, J.M.; Lott, J.; Breedy, M.; Bunch, L.; Feltovich, P.; Johnson, M.; Hyuckchul Jung, "New Developments in Ontology-Based Policy Management: Increasing the Practicality and Comprehensiveness of KAoS" IEEE Workshop on Policies for Distributed Systems and Networks, 2008. POLICY 2008, pp.145-152, 2008.