2013 IEEE 27th International Conference on Advanced Information Networking and Applications

Implementing Adaptation and Reconfiguration Strategies in Heterogeneous WSN

A. Di Marco, F. Gallo
Dip. di Ingegneria e Scienze dell'Informazione e Matematica
University of L'Aquila
L'Aquila , Italy
Email:{antinisca.dimarco,francesco.gallo}@univaq.it

O. Gemikonakli, L. Mostarda, F. Raimondi
School of Science & Technology
Middlesex University
London, UK
Email:{o.gemikonakli,l.mostarda,f.raimondi}@mdx.ac.uk

***Abstract*—Wireless Sensor Networks are becoming one of the most successful choices for the development and deployment of applications in a range of scenarios, from intelligent homes to environment monitoring. Nowadays, there is a growing demand for programming large-scale wireless sensor networks. New programming paradigms should ease the task of building WSN applications that adapt at run-time to changes in the context, in the available resources, and also in user requirements.**

In this paper we describe PROTEUS, a platform to manage adaptation and reconfiguration, with the aim of supporting the development of WSN applications. After introducing PROTEUS, we show how it can be used to program a dynamic clustering algorithm, where clusters are created and destroyed at runtime, and nodes need to adapt and reconfigure accordingly. We provide a prototype implementation using TinyOS. Some remarks on the work are also presented.

***Index Terms*—WSN, Clustering, Middleware, adaptation.**

I. Introduction

A Wireless Sensor Network (WSN) consists of spatially distributed autonomous sensors that cooperate in order to accomplish a task. Sensors have unique *characteristics*: they are small, low-cost, wireless and battery-powered devices. They can be easily deployed to monitor different environmental parameters and create large-scale flexible architectures. Sensors can be distributed on roads, vehicles, hospitals, buildings, people and enable different applications such as medical services, battlefield operations, crisis response, disaster relief and environmental monitoring. The unique *characteristics* of sensors also complicate the development of applications. Traditionally, WSN applications have been developed programming each sensor by using low level primitives. Although different programming abstractions and middlewares have been proposed, various research challenges are still open [1], [2].

New programming approaches should not only consider that a WSN is composed of limited resource devices but must take into account that the population of sensors can be quite large. In this case, the WSN should be programmed as a whole, without programming each sensor individually. For instance, programmers could be more interested in specifying the following behaviour: *when all the sensors inside the Area East have a high temperature, then some sprinklers must be activated.* Programming approaches should also consider that WSNs are often deployed in a dynamic environment, and thus there is need to *adapt* and *reconfigure* the WSN behaviour at run time. For instance, a WSN could adapt its sensing rate in response to some events or reconfigure its topology as consequence of failures.

In this paper, we propose PROTEUS, a novel approach for the implementation of adaptive WSN applications. The key features of PROTEUS inherit the characteristics of a biologically inspired paradigm previously presented in [3], [4]. PROTEUS is composed of a language and a middleware. The language provides primitives to group nodes into so-called membranes and to adapt the behaviour of nodes belonging to a membrane (see below). Properties can be assigned to a membrane so that only nodes verifying those properties can be part of the membrane. Effectively, membranes can be used to program the WSN (or a subset of its nodes) as a whole. Communication primitives (i.e the PROTEUS middleware) can be used to multicast a message to all the nodes in a membrane while PROTEUS adaptation primitives can be used to adapt the behaviour of nodes at run time (e.g., by changing the code of nodes).

For the purposes of this paper we have implemented PROTEUS for various platforms [5], thereby enabling the development of applications in heterogeneous WSNs. To show the validity of our work we have performed a case study implementing the clustering algorithm EECS [6]. We have discussed the implementation obtained and the overhead introduced in terms of memory and time.

The rest of the paper is organised as follows. In Section II we present the key features of PROTEUS paradigm, the PROTEUS framework architecture(Section II-A) and the primitives implemented in the provided prototype implementation (Section II-B). In Section III we describe EECS, a clustering protocol for WSN, and its corresponding implementation in PROTEUS. In Section IV we report an assessment of the performance of our approach using TinyOS. We review related work in Section V and we conclude the paper in Section VI.

II. The PROTEUS Paradigm

PROTEUS is a novel paradigm for adaptive systems that embeds the new concept of *virtual membrane* inspired by the biological membrane. The purpose of PROTEUS is to provide a level of abstraction to confer to a resource and the context it

1550-445X/13 $26.00 © 2013 IEEE
DOI 10.1109/AINA.2013.102

belongs to, the ability of reconfiguration and adaptation, after a change occurrs in the system, in its context or in the system user requirements.

PROTEUS provides a language for reconfiguration, that is a set of constructs that allow to "program" a reconfiguration plan [4]. Each reconfiguration plan refers to a virtual membrane that represents the scope of the reconfiguration/adaptation.

The basic elements that define PROTEUS are: the virtual membranes, the properties and the resources.

A *virtual membrane* groups several system resources satisfying a given property. Hence, the notion of virtual membrane is strongly linked to the concept of property.

With *property* we define a logic property used to select all the system resources that verify it. We call this set of resources a virtual membrane.

For example, an application may need to update/remove all the resources for which a certain state variable x has value less than 100. For this, we define a property P as follows:

$$P(r_i) = r_i.x < 100$$

Whereas, VM(P) is the corresponding virtual membrane (see Figure 1) defined as:

$$VM(P) = \{r_i \in System : P(r_i)\ is\ true\}$$

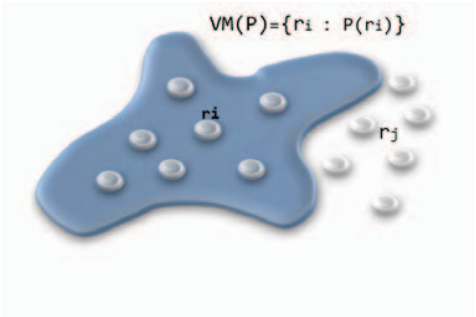

Figure 1. PROTEUS Membrane

Virtual membranes can be created and modified at run-time according to the system (status) evolution. This implies that, once built, the membrane itself is a new system resource that must be managed.

The power of the virtual membrane is that, once created, all the resources it aggregates can be simultaneously accessed (in a manner very similar to multicast) and adapted. This feature is fully exploited in the WSN domain and opens to a new approach that allows the programming of a WSN as a whole.

Moreover, it is possible to create concurrent (see Figure 2) and nested virtual membranes (see Figure 3). The concurrent nature of the virtual membranes enables the simultaneous adaptation of several portions of the system in accordance to the associated reconfiguration plan.

For what concern the creation of *nested* virtual membranes, this feature permits to create virtual membranes that respond to certain properties within a bigger membrane. This is particularly convenient and useful if there are resources inside

Figure 2. Concurrent Membranes

a virtual membrane that define specific characteristics. For example, external membranes can correspond to invariant properties, whereas their nested ones correspond to properties of the system status.

In Figure 3, we show an example of two nested membranes. The external one is defined by the property $P1 = \{r_i.x \leq 100\}$ (the membrane VM(P1)) and the internal one by $P2 = \{r_i \in VM(P1) \wedge r_i.x < 90\}$ (the membrane VM(P2)).

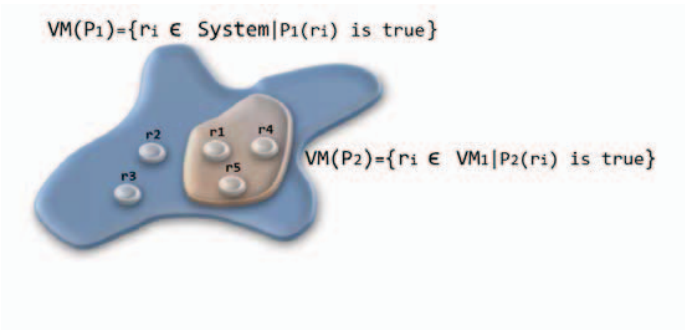

Figure 3. Nested Membrane

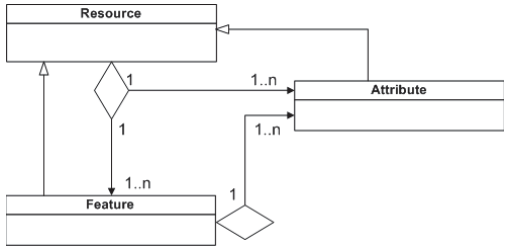

Figure 4. Resource

In PROTEUS all the system hardware and software modules are considered as *resources* to be managed. As already said, virtual membranes are themselves considered resources. In PROTEUS, in accordance to the conceptual model in Figure 4, resources are composed of *features* and *attributes* that

are again resources manageable from PROTEUS. Features and attributes define the behaviour and state of the resource, respectively. This permits that features and attributes can be added, modified or removed dynamically to respond to specific changes in the context of the system, in the available resources and in the user requirements.

A. Architecture of PROTEUS middleware

In this section, we describe the architecture of the PROTEUS middleware and how it can be used to develop applications that can reconfigure themselves automatically.

As showed in Figure 5, the implementation of a PROTEUS-based application is structured into four layers:

- *Application Layer* is composed of a set of resources and the reconfiguration manager. The resources are the PROTEUS abstractions for the hardware and software modules of the application. For instance resources can be temperature and humidity components. The reconfiguration manager can execute reconfiguration commands (see below). For instance the reconfiguration manager can add new software components at run time or update already existing ones. Both resources and reconfiguration manager can receive and invoke PROTUES primitives through the Proteus interpreter.
- *PROTEUS Layer* is the layer we define and for which we provide the prototype implementation in three different programming languages (that are Java, Pyton and NesC) downloadable from [5]. It is composed of three components: *PROTEUS Interpreter* that is able to interpret the PROTEUS primitives, the *Membrane/Property Capabilities* component that manages the virtual membranes and the *Communication Manager* that takes care of the communications. For instance the PROTEUS interpreter can receive a call from a resource to create a new membrane. The PROTEUS interpreter will interact with the Membrane/Property Capabilities component in order to create the membrane. It will also interact with communication layer in order to notify the addition of the new membrane to other PROTEUS interpreters running on a different devices.
- *Communication Layer* that implements point-to-point, multicast and broadcast communications;
- *Operating System* that can be of several types. In the implementation of the framework for WSN we assume to rest on TinyOS, Contiki and Linux (for both Intel and ARM architectures) operating systems.

In order to explain the use of PROTEUS for application development suppose we need to implement the support for the EECS protocol on a Raspberry PI device. This is an ARM device with the capability of running Linux.

First, the PROTEUS middleware needs to be installed on the actual device. This correspond to the "PROTEUS Layer" in Figure 5; we provide various implementations of the PROTEUS middleware using different languages. For instance, the Python implementation of our middleware can be installed directly on the Debian image of the Raspberry PI. Alternatively, one can choose to install a Java runtime environment and deploy our Java-based implementation for the middleware. The "PROTEUS Interpreter" in the PROTEUS Layer is responsible for accepting requests from a Reconfiguration Manager (see below) and for forwarding these messages to / from the Communication Manager.

Second, a developer needs to implement the logic of the application in the "Application Layer" of Figure 5. More specifically the developer needs to specify the resources. For instance a resource can be some software components executing the system functionality. Resources can call PROTEUS primitives in order to reconfigure the system or communicate with other PROTEUS resources. For instance when a resource requires the addition of new code into the system the configuration manager will perform it.

B. PROTEUS commands

In this section, we describe the set of commands that can be used in order to implement our biologically inspired paradigm. These commands have been implemented in our toolset and are available on line at [5]. For the sake of presentation, we group the PROTEUS commands into three sets: commands for the management of the virtual membranes, commands for the system and resources reflection and commands for system and resources adaptation.

1) Virtual membrane management commands: A membrane aggregates a set of resources. It has related various attributes such as a label and properties. A label is a string that uniquely identify the membrane inside the system. A property is a boolean formula defined over the state of the resources. A virtual membrane can be considered an adaptive structure since new resources are added and removed at run time when they validate the property. The following are the commands that our prototype provides to manage and interact with membranes.

SETM(P,M). This command allows a programmer to create a virtual membrane M with a property P. All resources that verify the property P will be aggregated inside the membrane M. Notice that a conflict can be generated when two different membranes with the same label but different properties are created. In this case the conflict will be detected by the command $SETM(P, M)$ and reported to the user. The default action is to consider the *OR* of the properties that is to perform the union of the two membranes.

CALL(Service,M). This command can be used to call a service on all resources of a membrane. We assume that a service call is reliable and services have no returned values.

2) System/Resources reflection commands: In the following we describe various PROTEUS commands that allow a programmer to have information about the resources (including the defined virtual membranes) of the system.

GETV(). This command returns the list of membranes

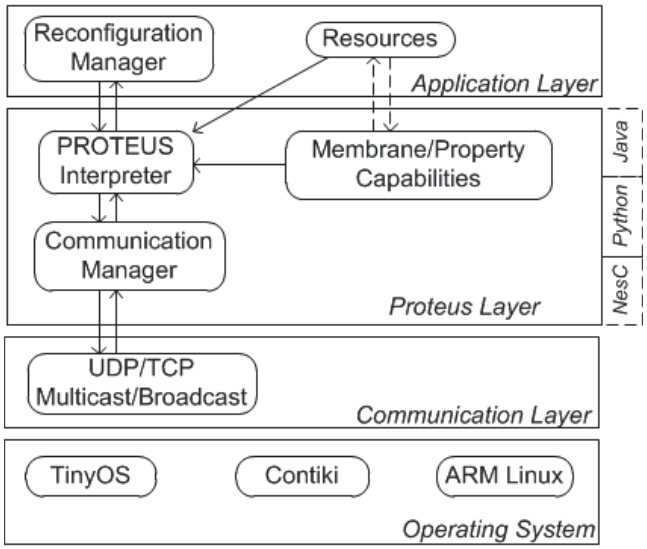

Figure 5. PROTEUS-based Software Architecture Elements

$\{VM_1, \ldots, VM_j\}$ that have been defined inside the system. A membrane is a structure that contains different information such as a label, a property and so on.

GETV(M): This command returns the list of nodes $\{n_1, \ldots, n_i\}$ belonging to the Membrane M.

GETC(NodeID): This command returns the list of all the attributes and features of the resource NodeID.

3) System/Resources adaptation commands: The PROTEUS language also defines a group of **Actions** that allow the modification of the code of system resources. These actions are defined as.

add((code,type),M). This command is used to add a piece of code to all resources inside the membrane M. Type is used in order to describe the type of code to be added. It can have two values that are attribute and features. For instance $add((int\ x;, attributes), M)$ can be used to add the variables x to all nodes inside the membrane M. The call $add((sum(int\ x, int\ y), features), M)$ can be used to add the method sum to all nodes inside the membrane M.

remove((code,type),M): to remove a piece of code to all resources inside the membrane M.

update((code,type),M): to update a piece of code in all resources inside the membrane M.

III. Case study

Prolonging the network lifetime, scalability and load balancing are the most important requirements when implementing and designing a wireless sensor network. Clustering sensor nodes is a technique that has been applied for achieving these goals. Clustering organises the WSN into disjoint subsets (clusters). A sensor from each subset is elected as cluster head (CH). A CH coordinates and aggregates data of nodes within its clusters (intra-cluster communication). CHs communicate with each other and/or with external observers on behalf of their nodes. There are various protocols that implement different clustering strategies. Here we present a high level description of the clustering protocol EECS [6]. In the following section we show how PROTEUS can encode this protocol.

A. EECS Protocol Overview

In this section we present a high level description of the Energy Efficient Clustering Scheme (ECCS) proposed in [6].

EECS protocol is composed of three steps, namely: *Node Synchronization*, *Cluster Head Election* and *Cluster Formation phase*.

Node Synchronization phase: is executed during the network deployment phase when the Base Station (BS) sends an *hello* message to all the nodes so that they can compute an approximate distance from the BS. This allows each node to set an acceptable transmission power in order to reach the BS. The distance from the BS is also used in the cluster formation phase.

Cluster Head Election phase: this phase aims at determining the cluster head. A node can become CANDIDATE with probability T (*a threshold between 0 and 1*) and then broadcast a COMPETE_HEAD_MSGs message within the radio range $R_{compete}$ (*broadcast radius of candidate nodes*). In Figure 6, we show the pseudo-code for the EECS leader election. Between lines 6-16 each CANDIDATE receives COMPETE_HEAD_MSGs from other CANDIDATEs. When another CANDIDATE has more energy, the node will give up and set its state to PLAIN (a non cluster head node). Otherwise it will be elected as a cluster head.

```
1
2  /*Timer_phase1 must be enough to complete the Cluster Head
       Election*/
3
4  state = PLAIN
5  μ= Random(0,1)
6  if μ < T
7    state = CANDIDATE
8    broadcast COMPETE_HEAD_MSG
9    while Timer_phase1 not expired
10     msg= receive COMPETE_HEAD_MSG
11     senderId = msg.getSenderID()
12     if E_residual < E_sender
13     or (myid<senderId and E_residual == E_sender)
14       state=PLAIN
15       break;
16 if state==CANDIDATE
17   state=HEAD
```

Figure 6. Cluster Head Election

```
1
2  if state==HEAD
3    broadcast HEAD_AD_MSGs;
4    wait for JOIN_CLUSTER_MSGs
5  if state==PLAIN
6    receive all HEAD_AD_MSGs
7    compute cost for each cluster head
8    choose the cluster head CH with min{cost}
9    broadcast JOIN_CLUSTER_MSG
```

Figure 7. Cluster formation

Cluster Formation phase: in the cluster formation phase each CH broadcasts a HEAD_AD_MSG message. This is received by the PLAIN nodes that decide which cluster to join. The algorithm for cluster formation is show in Figure 7.

Synchronisation for the first phase is performed choosing a proper time interval $Timer_{phase1}$. This is done according to the system parameters and wireless channel quality. In the cluster formation phase, each CH broadcasts a Time Division Multiple Access (TDMA) schedule [7]. This allows different members of a cluster to share the same frequency channel by dividing the signal into time slots. Members transmit, one after the other, by using the slot assigned by the CH and turn off the radio in the other intervals to save energy. The BS should periodically synchronise the nodes over the network to mitigate any time drift effects.

B. Case study implementation: EECS in Proteus

In this section we describe the implementation of EECS in PROTEUS. The cluster head election is showed in Figure 8.

Each node n that becomes a CANDIDATE defines the membrane *candidates*. This contains all nodes that have their state set as CANDIDATE and are neighbours of the node n (line 9). Each membrane has a label that uniquely identifies it. For instance our membrane *candidates* has the label defined by the string candidates followed by the node id. This membrane is used by the node to multicast a $COMPETE\_HEAD\_MSG$ call to all other candidates in the same area (line 10). A node that receives this call compares its energy with the energy of the caller. When the caller has more energy the node sets its state to PLAIN (line 22).

```
1  Membrane candidates,inRange;
2
3  void cluster_head()
4  state = PLAIN
5  μ= Random(0,1)
6  if μ < T
7    state = CANDIDATE
8    candidates.label=strcat("candidates",myNode.Id)
9    SETM(state==CANDIDATE and isNeighbour(myNode.Id),
         candidates)
10   CALL (COMPETE_HEAD_MSG(myNode),candidates)
11
12 void timeout()
13   if state==CANDIDATE
14     STATE=HEAD
15     inRange.label=strcat("candidates",myNode.Id)
16     SETM(state==PLAIN  and isNeighbour(myNode.Id),inRange)
17     CALL (HEAD_AD_MSGs(myNode),inRange)
18
19 void COMPETE_HEAD_MSG(Node n)
20   if E_residual < E_sender
21   or (myNode.Id<senderId and E_residual == E_sender)
22     state=PLAIN
```

Figure 8. Cluster election in PROTEUS

```
1  Membrane myMembers
2  Node myCH
3
4  void Timeout1()
5    if state==HEAD
6      myMembers.label=strcat("myMembers",myNode.Id)
7      SETM(state==PLAIN and myCH.id==myNode.Id,myMembers)
8      CALL (tdmaSchedule(data),myMembers)
9
10 void HEAD_AD_MSGs(Node n)
11   if state==PLAIN
12     if computeCost(n)<mincost(myCH)
13       myCH=n
```

Figure 9. Cluster formation in PROTEUS

After the call $COMPETE\_HEAD\_MSG$ is completed, the timeout function will be called (line 12). This function is called when a timeout expires. This timeout must be long enough so that the node receives all $COMPETE\_HEAD\_MSG$ calls from other nodes. The timeout function performs the $HEAD\_AD\_MSGs$ call on its PLAIN neighbours in order to start the cluster formation that is described in the following.

The $HEAD\_AD\_MSGs$ starts the cluster formation (see Figure 9). It allows each PLAIN node to compute the CH with the least cost (line 12). The $Timeout1()$ method is called when a timeout (timeout 1) expires; this must be long enough so that all PLAIN nodes receive a $HEAD\_AD\_MSGs$ message from all cluster heads. The $Timeout1()$ method is used by the cluster head CH to define the membrane $myMembers$ (line 7). This is composed of all nodes that set myCH as the cluster head. The membrane $myMembers$ is used by the cluster head in order to send the TDMA schedule.

```
1  Membrane allCH
2
3  SETM(state==HEAD,allCH)
4  CALL(HELLO(),allCH)
```

Figure 10. HELLO message from BS

In Figure 10, we show how the base station sends the HELLO message. This is used by the BS to synchronise the CH over the network to mitigate any time drift effects. The cluster head will start a new cluster head election after a certain amount of time.

IV. DISCUSSION AND REMARKS

PROTEUS allows a programmer to implement a WSN as a whole by using membranes. PROTEUS adds / removes nodes to a membrane at run time. This is done to make sure that nodes inside the membrane verify the properties assigned to the membrane.

Membranes ease the task of programming the WSNs and reduce the size of the code to be written. All synchronisation code required in order to build and keep synchronised a set of nodes verifying a property is factorised inside the $SETM$ PROTEUS primitive. The programmer can multicast method calls to all nodes inside a membrane with high level primitives. For instance in the function of Figure 10 the BS needs two lines ($SETM$ and $CALL$) in order to multicast an $HELLO$ message to all CHs in the network. The same code in TinyOS or CONTIKI would require the BS to perform the following steps: (i) use low level broadcast primitives; (ii) define the message to be sent; (iii) implement a reliable communication (if none is implemented). Each node must also have some code in order to receive the broadcasted HELLO message and relay it (if needed).

PROTEUS allows a programmer to add and/or remove code at run time without stopping the application. This can be needed when the WSN has to adapt its behaviour to a changing environment. For instance in our EECS case study the programmer could need to update the random number generation function. This is used in Figure 8 (line 5) to calculate the probability of a node to become cluster head. In Figure 11 we show the code executed by the base station in order to update the Random function. We get all the membranes defined inside the system and we store them in a list. This is done by using the $GETV()$ primitive (line 3). For each membrane we update the random function (line 4-6) by using the *update* primitive provided by PROTEUS.

```
1 MembraneList *membraneList;
2
3 membraneList=GETV();
4 while(membraneList!=null)
5     update((Random(int x,int y),feature),membraneList->
          membrane)
6     membraneList=membraneList->next
```

Figure 11. Update of the random function

A programmer can also use PROTEUS to add the EECS code into WSN nodes. This can be needed when new sensors are added into the systems.

In Figure 12 we show the memory overhead of our PROTUES prototype implementation. The code of the EECS component is 2797 bytes while the PROTEUS component is 1213 bytes. The data part of EECS is 40 bytes while the PROTEUS is 45. It is worth mentioning that the data part is dependent on the size of the membranes (the number of members).

Component	Code bytes	Data bytes
EECS	2797	40
PROTEUS	1213	45

Figure 12. PROTEUS memory overhead

Overhead in term of messages is introduced in order to keep the membranes synchronised but this is an inevitable cost if the application requires synchronisation among sensors. In our preliminary prototype each node periodically broadcast PROTEUS control messages to its neighbours. A control message contains the membranes locally defined and info about the node state. Other nodes receive the control message and update their local membranes with the new membranes (if any) and the node state. In our experiments the bandwidth requirements for control messages are negligible when compared to the bandwidth required by the actual application.

V. RELATED WORK

In this Section we describe various middlewares and platforms that can be used in order to build WSN applications.

The TeenyLIME middleware [8] is specifically designed to implement WSN applications. It provides a programming model based on the tuple space paradigm. A tuple space in TeenyLIME represents a memory that is shared among sensors within a one-hop region. While TeenyLIME offers a simple but powerful abstraction it lacks some of our abstractions. A virtual membrane can be defined over the entire WSN thus a node using a membrane is effectively interacting with nodes outside the one-hop region. The members of a membrane are automatically updated at run time so that the membranes properties are validated. This provides a layer of abstraction and avoids the user to implement subtle and error prone distributed algorithms that are needed to keep all the membranes synchronised.

ESCAPE [9] is a policy-based platform that can be customised for the needs of WSN applications. While application components are used to implement basic functionalities (sensing and reacting) ESCAPE supports the specification and enforcement of policies that cater for application-domain requirements. Policies can be updated at run time. ESCAPE has the concept of sets that is closely related with membranes. However sets are not automatically updated when nodes change at run time.

TinyCOPS [10] is a pub/sub middleware that uses a component-based architecture for decoupling the pub/sub core from choices regarding communication protocols and subscription and notification delivery mechanisms. The middleware can be extended with components that provide additional services such as notification caching, extra routing information. The Mires middleware [11] is also a pub/sub system that is based on the component architecture of TinyOS. It

uses a topic-based naming scheme. In Mires it is possible to introduce new services (like aggregation) using extension components, the choice of the communication protocols is fixed. While the aforementioned middlewares support event-based programming we believe our approach provides higher level primitives such as membranes and reliable multicast synchronous calls on membranes. These can make the program flow easier to understand.

MiLAN [12] is a middleware for WSNs that provides application QoS adaptation at run-time. The middleware continuously tracks the application needs and optimises network usage and sensor stacks for an efficient use of the energy. As such, MiLAN focuses more on a class of resource-rich wireless networks that can support well the impact of the monitoring overhead. In our approach, we concentrate more on sensors with limited resources, where optimisations are mainly performed at compile-time.

VI. Conclusions and future work

In this paper we have described PROTEUS, a platform for the development of applications in WSN with a particular emphasis on supporting adaptation and reconfiguration. We have described the architecture of our platform and provided primitives that can be used in actual code. To enable the management of heterogeneity, we have provided implementations of our system for various platforms (Java, Python, NesC). We have validated our approach by implementing the EECS clustering protocol. PROTEUS command and reconfiguration primitives have been used to implement the reconfiguration of the clusters when new CH are elected. We have assessed PROTEUS by discussing the EECS implementation produced and by analysing the memory overhead introduced.

As a future work we are planning to optimise the implementation of PROTEUS and further validate the approach through the acquisition of data from the use of PROTEUS in real-world scenarios. We are considering security issues such as authentication and confidentiality. We are also extending the range of platforms we support. We are planning to implement PROTEUS in Contiki [13] and other operating systems.

VII. Acknowledgements

This work has been partially supported by VISION ERC project (ERC-240555).

References

[1] J. A. Stankovic, "Research challenges for wireless sensor networks," *SIGBED Rev.*, vol. 1, pp. 9–12, July 2004.

[2] K. Römer, O. Kasten, and F. Mattern, "Middleware challenges for wireless sensor networks," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 6, no. 4, pp. 59–61, Oct. 2002.

[3] A. Di Marco, F. Gallo, and F. Raimondi, "Proteus: a language for adaptation plans." Nice, France: The Fourth International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE 2012), July, 2012.

[4] ——, "Proteus language," University of L'Aquila, Tech. Rep., 2012.

[5] A. Di Marco, F. Gallo, L. Mostarda, and F. Raimondi, "Proteus language: Proteus implementation code," University of L'Aquila, Italy and Middlesex University, London, U.K., Tech. Rep., 2012.

[6] M. Ye, C. Li, G. Chen, and J. Wu, "Eecs: an energy efficient clustering scheme in wireless sensor networks," in *Performance, Computing, and Communications Conference, 2005. IPCCC 2005. 24th IEEE International*, april 2005, pp. 535 – 540.

[7] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison-Wesley, Jul. 2005.

[8] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco, "Programming wireless sensor networks with the teenylime middleware," in *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, ser. Middleware '07. New York, NY, USA: Springer-Verlag New York, Inc., 2007, pp. 429–449.

[9] G. Russello, L. Mostarda, and N. Dulay, "A policy-based publish/subscribe middleware for sense-and-react applications," *Journal of Systems and Software*, vol. 84, no. 4, pp. 638–654, 2011.

[10] J.-H. Hauer, V. Handziski, A. Köpke, A. Willig, and A. Wolisz, "A component framework for content-based publish/subscribe in sensor networks," in *Proceedings of the 5th European conference on Wireless sensor networks*, ser. EWSN'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 369–385.

[11] E. Souto, G. Guimarães, G. Vasconcelos, M. Vieira, N. Rosa, C. Ferraz, and J. Kelner, "Mires: a publish/subscribe middleware for sensor networks," *Personal Ubiquitous Comput.*, vol. 10, no. 1, pp. 37–44, Dec. 2005.

[12] W. B. Heinzelman, A. L. Murphy, H. S. Carvalho, and M. A. Perillo, "Middleware to support sensor network applications," *IEEE Network*, vol. 18, no. 1, pp. 6–14, 2004.

[13] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, ser. LCN '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 455–462.