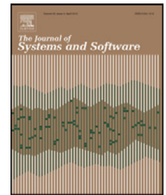
The Journal of Systems and Software 84 (2011) 638–654

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss



A policy-based publish/subscribe middleware for sense-and-react applications☆

Giovanni Russello[a,*], Leonardo Mostarda[b], Naranker Dulay[b]

[a] Create-Net, Trento, Italy
[b] Imperial College London, London, United Kingdom

ARTICLE INFO

Article history:
Received 26 August 2009
Received in revised form 19 October 2010
Accepted 19 October 2010
Available online 29 October 2010

Keywords:
Wireless Sensor Networks
Sense-and-react applications
Separation of concerns
Publish/subscribe middleware
Component-based software engineering
Policy-based composition

ABSTRACT

With the inclusion of actuators on wireless nodes, Wireless Sensor Networks (WSNs) are starting to change from sense-and-report platforms to *sense-and-react* platforms. Applications for such platforms are characterised by actuator nodes that are able to react to data collected by sensor nodes. Sensor and actuator nodes use a variety of interactions, for example, intra-node, inter-node (1-hop to *n*-hop), and global (all nodes). As a result, the functionality that coordinates the activities of the different nodes towards common goals has to be efficiently distributed in the WSN itself. In addition, multiple sense-and-react applications are being deployed within the same WSN, with each application characterised by different requirements and constraints. The design and implementation of these applications is becoming an increasingly complex task that would benefit from new approaches.

In this article, we describe a novel middleware that separates the *interaction* behaviour of sense-and-react WSN applications from the components that implement the basic functionalities (sensing, reacting, computation, storage). This is achieved using *policies* that govern the interaction behaviour of sense-and-react WSN applications. The middleware is composed of a Policy Manager, a Publish/Subscribe Broker, and a set of Extensions that reside on each node. The broker manages subscription information, while extensions provide mechanisms orthogonal to the publish/subscribe core including diffusion protocols, data communication protocols, and data encryption. We conduct a detailed evaluation of the performance of our framework and show that the framework is close to TinyOS in performance but leads to more explicit and flexible application designs.

© 2010 Elsevier Inc. All rights reserved.

1. Introduction

Early applications developed for Wireless Sensor Networks (WSNs) were technically very simple. Because of the limited computational power of the sensors at that time, WSNs were mainly used for collecting data about the sensed environment. The data was then routed to a central sink device with more computational power (e.g., PDAs and laptops) for processing. Therefore, the functionality deployed in each node was dedicated to collect the sensing data and to deliver the data to the sink.

With the advent of actuator nodes and the development of sensor nodes with more computational power, it is now possible to embed more complex functionality. For such WSNs, the *sense-and-react* application paradigm has emerged characterised by the fact that the data gathered by the sensing nodes can be used directly by actuator nodes that can react and change the sensed environment (Deshpande et al., 2005). Because of the more complex interactions between nodes and the stringent physical limitations that characterise WSNs (i.e., in terms of power management) sense-and-react applications are often developed in an ad-hoc manner optimised for the specific environment of deployment. The code of such applications does not only deal with its basic functionality (i.e., collecting data and reacting to it) but it is intertwined with details to deal with other concerns such as data distribution and resource utilisation. Moreover, it is becoming common to have multiple sense-and-react applications deployed within the same WSN sharing some of its nodes (i.e., an air conditioning application and a fire alarm application sharing the same nodes for sensing the temperature). As a result, flexibility, maintainability, and reusability properties of the code for programming such applications are compromised (Intanagonwiwat et al., 2003; Madden et al., 2005).

There is a need for a more rigorous software engineering approach in the development of sense-and-react applications. As advocated by the *Separation of Concerns* (SoC) principle, the core functionality of an application should be specified in isolation from details regarding *extra-functional* concerns (Dijkstra, 1982; Parnas, 1972). Typically, the code for extra-functional concerns is customised and optimised for the specific environment where the overall application is deployed. In separating the application functionality from such details makes the code less prone to errors, and leads to greater reuse and easier maintenance.

☆ A preliminary version of this paper appeared in CBSE 08.

* Corresponding author.
E-mail addresses: giovanni.russello@create-net.org, russello@doc.ic.ac.uk (G. Russello), lmostard@imperial.ac.uk (L. Mostarda), n.dulay@imperial.ac.uk (N. Dulay).

0164-1212/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2010.10.023

The first step in order to realise the SoC principle is to provide a programming unit for encapsulating the application functionality. The component-based approach provides an efficient programming abstraction and encapsulation that is applicable to the WSN domain (Hill et al., 2000). *Application components* can be used as unit of functionality and fulfill the functionality requirements of the node where they are deployed. For instance, a component can be programmed for sensing data from the environment while components deployed on actuator nodes gather the data and react accordingly. The second step required is to add a layer of abstraction that can be used for coordinating the functionality of the application components and to provide the required mechanisms for handling extra-functional concerns. However, the typical abstraction provided in WSN is that of the operation system TinyOS (Levis et al., 2005) to hide to the application layer low level details.

With its loosely coupled, event-driven messaging services, the publish/subscribe paradigm (hereafter referred to as pub/sub) offers to applications simple yet powerful primitives for communication. Although the pub/sub paradigm is well understood and widely used, there are several aspects concerning notification distribution, delivery and security that can be implemented using different approaches. Each approach can be characterised by properties that satisfy the requirements of specific application domains and at the same time each imposes a set of requirements in terms of resource utilisation. For instance, for a critical application it is acceptable to use a reliable delivery protocol even if it requires more resources in terms of energy consumption and computational overhead. To make the implementation of a pub/sub system flexible enough to be used in application domains with different requirements it is necessary that the implementation is able to offer multiple approaches to satisfy the needs of each target domain. Moreover, because several applications from different domains can be deployed at the same time, it is desirable that the same instance of such a pub/sub system can support several approaches to satisfy the requirements of the deployed applications.

In this article, we describe a novel component-based framework for sense-and-react WSN applications realised through a pub/sub middleware where extra-functional concerns are defined separately from application components.[1] In particular, application developers specify the interaction behaviours of components and which mechanisms are needed in their applications in terms of *policies*. Policies are rules that govern the behaviour of a system and are an effective solution for separating the application functionality from low-level mechanisms (Sloman et al., 1993). The framework uses an Event-State-Condition-Action policy model where policies *connect* application components to middleware components orthogonally to the pub/sub paradigm. Policies effectively define *s* tateful interactions among components to *coordinate* their activities and are used to fulfil system-wide goals. Once a specific behaviour is defined in a policy that policy can then be deployed or adapted for other applications with similar characteristics.

The rest of this article is organised as follows. In Section 2, we discuss the motivations and requirements of our approach. Section 3 provides a description of the architecture of our framework. Policy syntax and semantics are described in Section 4. To validate our approach, we present in Section 5 a case study and some of the policies used for its realisation. An evaluation of the implementation of our framework is presented in Section 6. Section 7 presents how our approach fulfils the set of requirements discussed in Section 2. In Section 8, we compare our approach to related research. We conclude in Section 9.

[1] The framework described in this paper was firstly presented in Russello et al. (2008). In this paper, we have extended the description of the execution model, presented an extended case study and provided a more detailed evaluation analysis.

2. Motivations and requirements

Sense-and-react applications represent a class of embedded control systems characterised by the realisation of a feedback-loop between a *sensing apparatus* and a *reacting apparatus*. Some examples of sense-and-react applications are heating, ventilation, and air conditioning (HVAC) (Deshpande et al., 2005), fire alarm systems, and burglar alarm systems. Nodes capable of sensing the environment provide readings of some parameters forming the sensing apparatus. Nodes equipped with actuators react to specific events and change the environment according to user preferences. As a software system, a sense-and-react application consists of two parts: the main *f* unctionality and the *control rules*. The main functionality represents the basic logic that is mapped into application components deployed in each sensor node. For instance, an application component deployed on a temperature node provides the functionality to obtain the readings from the node hardware and make it available to other nodes in the WSN. The control rules map the sensed data to specific actions that should comply with user preferences. When the functionality and control rules are not intertwined then it becomes possible to share the functionality of a node among different applications controlling the same environment. For instance, the functionality of a temperature node could be shared by both a HVAC and a fire emergency application. The two applications have different control rules however the functionality of the temperature node for both applications is the same, e.g. providing readings for the temperature of the environment.

From the above simple example, it emerges that the development of sense-and-react applications for WSNs is challenging not only for the constraints imposed by the physical devices but also for the complexity of the interactions that can be realised among different applications. In the following, we try to identify a set of requirements to define key aspects for facilitating the development of such applications.

R1: Minimise functionality to maximise reuse. In our framework, the node functionality is encoded as an application component deployed on the node. In order to maximise the reuse of such functionality, component functionality should be agnostic of the control rules enforced in the environment.

R2: Coordination through middleware. WSNs can be seen as distributed systems. The publish/subscribe model offers a very powerful abstraction for realising loosed-coupled distributed applications and middleware implementations have been already proposed in WSNs. In particular, the *reactive* style of interaction makes the model attractive for sense-and-react applications. Notifications sent by the sensing nodes can be used to trigger reactions by actuators. The underlying middleware is also an ideal place where the SoC principle can be realised. In particular, the middleware can provide to the application developers mechanisms that would allow the selection of different strategies implementing extra-functional concerns that can be subsequently enforced at runtime. For example, if a particular notification strategy is required, the middleware should offer such a strategy implemented as a component. Application developers specify which particular components have to be used with their applications in terms of policies. If necessary, new mechanisms can be developed and deployed as well, independent of the application functionality implemented by components.

R3: Stateful policies. The control rules in sense-and-react applications typically represent *transitions* through different *states*. For instance, a sprinkler node that receives a temperature reading higher than a certain threshold has to check that smoke is detected before opening the water. This behaviour can be represented as two transitions: (i) from a normal state to a pre-alarm state when the temperature is above a safety threshold; and (ii) from a per-alarm

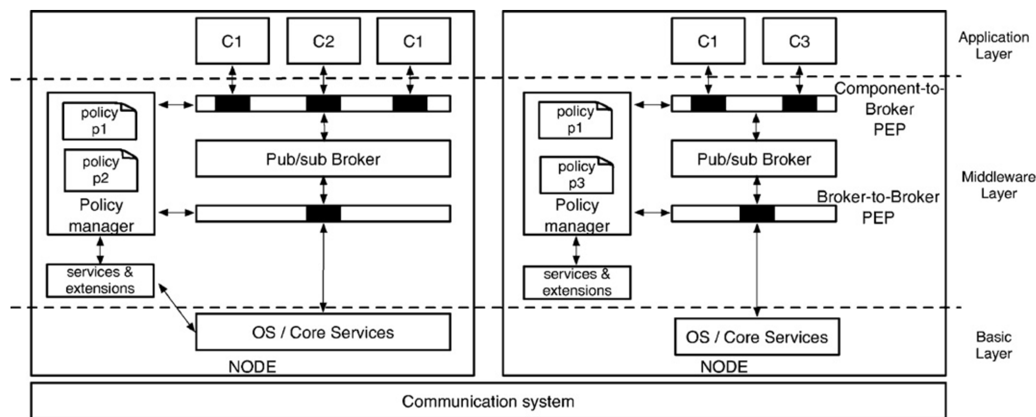


Fig. 1. System architecture.

state to an alarm state when the smoke detector provides a positive reading. To increase reuse, control rules should be specified independently from application components. Policies represent ideal candidates for specifying the control rules of the system. In this case, policies need to be able to capture states and specify how transitions through different states must be executed.

R4: Localised vs distributed computation. Sense-and-react applications are characterised by their capacity of reacting to stimuli coming from the surrounding environment. Because actuators can be in the proximity to where the data is generated, it is not necessary to flood the network with all readings. However, in some cases it is necessary that events have to be spread through the nodes present in the environment, such as fire alarms.

R5: Proactive interaction patterns. Although reactive interactions characterise sense-and-react applications, there are still cases where *proactive interaction* is preferred instead. This type of interaction is common in sense-only applications where data is proactively sensed by the components. We can also save energy on the sensing nodes by requesting data is generated only when needed.

In the following, we describe the architecture of our system to satisfy the identified requirements.

3. Architecture

Fig. 1 shows the architecture of our policy-based approach. It is composed of a set of components deployed on different nodes. In each node components are arranged in the following basic three layers: an *application layer*, a *m* iddleware layer and a *basic layer*.

The application layer contains the application components deployed on a sensor node. Components are used for encapsulating the application functionality. As we show in Fig. 1, different components can be deployed on different nodes depending on the hardware supported by the hosting node. For instance, a temperature application component is deployed on a node sporting a temperature sensor. The sensor is responsible for providing temperature readings to the application components. The application component might act as a publisher for this type of notification. An actuator application component can be deployed on actuator nodes where it can be responsible for controlling the actuator hardware according to current needs. In this case, actuator components act as subscribers of notifications representing the actual conditions of the environment. All applications components interact with the underlying middleware layer through the Policy Enforcement Points (PEP) (a description of an architecture reference for policy-based frameworks can be found in Yavatkar et al. (2000)). They are part of the policy manager and are used to capture all middleware interactions in order to evaluate the related policies.

The middleware layer consists of a *Publish/Subscribe Broker*, a *Policy Manager*, and a set of *Extensions* described as follows.

3.1. Publish/Subscribe Broker

This provides an API to the application layer and manages subscriptions. In our framework, a notification is called a **topic**. A subscriber specifies its interest in a topic by issuing a `subscribe T` where `T` is the tuple representing the topic. Although the subscriber cannot directly express constraints on the content of a topic using the API, constraints can still be expressed in policies. For instance, if a subscriber should be notified only if the temperature value is higher than 50, then it is possible to write a policy that inspects the values of the temperature notifications and discard the notifications with values lower than 50 (more on this in Section 5). This decoupling of component functionality from subscription constraints increases the reusability of the components without sacrificing the expressivity of the publish/subscribe abstraction. In our framework content constraints are used for expressing control rules in the form of policies. A publisher advertises its topic using `advertise T` and it publishes the data using `notify T`. Subscription and advertisements can be withdrawn using `unsubscribe T` and `unadvertise T`, respectively.

3.2. Policy Manager

One of the main features of our framework is that the publish/subscribe core is decoupled from abstractions, semantics and mechanisms related to notification delivery, subscription distribution, and communication protocols. This design decision increases the flexibility of the approach since our middleware is not bound to any specific mechanisms. Application developers can select the appropriate mechanisms that best suit their application needs. In contrast to the approach presented by Hauer et al. (2008) where application components have to explicitly specify the mechanism to be used, in our framework components are completely agnostic of such specifications. Instead, we describe a policy-based approach where policies are used for such specifications. The enforcement of these policies is done by a Policy Manager. Each node contains a Policy Manager to manage and enforce the policies for that node. Policies can be specified to be enforced at specific points representing the component-to-broker and broker-to-broker interactions. These points are monitored via Policy Enforcement Points (PEPs). Each time a message is sent through these points, the corresponding PEP intercepts the message and sends an event to the Policy Manager. The Policy Manager uses the events to trigger the policies available in its repository defined for that PEP (more on how policies are specified and enforced in Section 4). An important

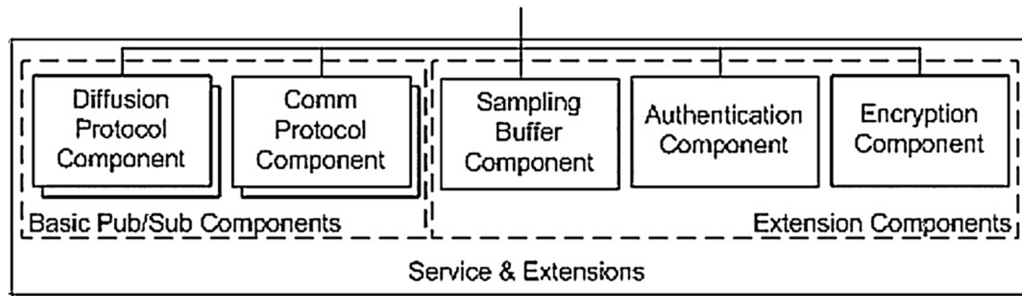


Fig. 2. Overview of Extensions.

feature of our policy environment is that, the Policy Manager supports the loading of new policies even at run-time without the need of taking the running application off-line. This feature increases flexibility of our framework and it makes particular appealing for WSN applications that require a high degree of availability.

3.3. Extensions

These provide hooks to the framework for invoking components that implement protocols and services outside the publish/subscribe core. Each extension component is responsible for providing and consuming information required for fulfilling their tasks and if necessary to perform specific actions. Fig. 2 shows some of the extensions currently available. Components are organised into two sets: *Basic Pub/Sub Components* and *Library Components*.

The Basic Pub/sub Components provide services that are necessary for realising the publish/subscribe paradigm and are described as follows:

- A *Diffusion Protocol Component* (DPC) is responsible for routing data between publishers and subscribers. Early work on diffusion protocols used a *two-phase pull* model (Intanagonwiwat et al., 2000), where subscriptions are distributed for the seeking of matching advertisements. Once a matching advertisement is found, the notifications are sent to the subscribers trying to find the best possible paths. This type of protocol is not suitable for all classes of application. In applications with many publishers that produce data only occasionally, the two-phase pull model is inefficient since it generates a lot of control traffic for keeping the delivery route updates. For this class of applications, the *push* diffusion protocol is more suitable in Heidemann et al. (2003). In the push diffusion protocol, the subscriptions are kept locally and the notifications seek subscribers. Other diffusion protocols have also be proposed, such as a *one-phase pull* protocol (Heidemann et al., 2003) (an optimised version of the two-phase pull), geographically scoped protocols (Yu et al., 2001), and rendezvous-based protocols (Braginsky and Estrin, 2002; Ratnasamy et al., 2002). Our current implementation provides a Push DPC (PushDPC) and a Pull DPC (PullDPC) implementing the push and one-phase pull protocols, respectively.
- A *Communication Protocol Component* (CPC) implements the delivery protocols of the messages generated by the publishers and subscribers with certain delivery guarantees. For instance, in certain cases subscriptions need to be updated frequently then a CPC that implements unreliable delivery is acceptable. On the other hand, if an application requires a more reliable subscription distribution then a CPC that offers reliable delivery can be used (at a higher costs in terms of resources). The former protocol is implemented by an unreliable CPC (UnreliableCPC), while the latter is implemented by a Reliable CPC (ReliableCPC).

Library Components provide extra features. In the following we describe the components that are used in our case study discussed in Section 5.

- The *Sampling Buffer Component* (SBC) provides functionality for storing data samplings and computing certain predicates on the stored values. For instance, it can be used for calculating if a recent sampling differs more than a specified delta value from a stored sampling. Similarly, it could just be used as a buffer that stores samplings frequently accessed or that require a longer time to be collected (i.e., audio signals).
- The *Authentication Component* (AC) and the *Encryption Component* (EC) are used for implementing *Secure Groups* of sensors. Secure groups are similar to secure multicasting groups (Rafaeli and Hutchison, 2003). Each secure group is associated with a secret key K_g. In this way, members of the same secure group are able to perform encryption and authentication within the group. The distribution of K_g is done using an out-of-band bootstrapping process. For the encryption/decryption the component uses the Skipjack algorithm provided by the TinyOS2 core.

4. Event-State-Condition-Action Policy Environment (ESCAPE)

In this section, we define the syntax and the semantics of our policy language. The syntax is defined by using a Backus–Naur form (BNF) while the semantics is described by defining the runtime behaviour of our Policy Manager. We will use as an example throughout this section the ESCAPE policy in Fig. 3. This policy is also represented as a state machine using our graphical tool (see Fig. 4). This policy captures humidity variation with the following

```
1  humidityVariationPolicy
2    int prevValue, threshold=10;
3
4    on C-B notify Humidity(h_value)
5     0-1: true ->
6               prevValue = h_value;
7               discard;
8
9     1-1: abs(prevValue,h_value)<=threshold ->
10              prevValue = h_value;
11              discard;
12
13    1-1: abs(prevValue,h_value)>threshold ->
14              prevValue = h_value;
15              notify Excess;
16              accept;
17
18   on timeout(5000)
19    0-2: true -> sendAlert();
20    1-2: true -> sendAlert();
```

Fig. 3. An example of policy for detecting humidity variations.

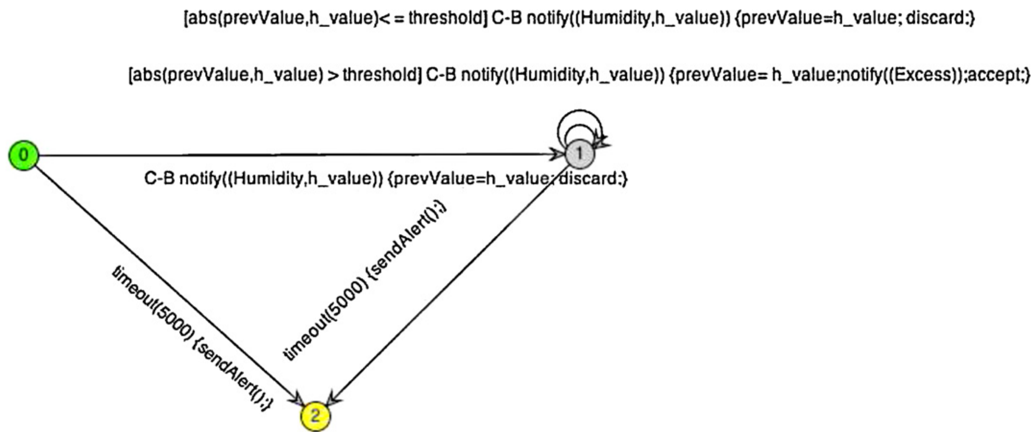


Fig. 4. Graphical view of the humidity variation policy.

requirement: *when a variation of humidity exceeds a given threshold it should be detected and a notification sent.*

4.1. The ESCAPE Policy syntax

In Fig. 5, we define the grammar used to define Event-State-Condition-Action (ESCAPE) policies. Terminals are enclosed within quotes or boldface. Square brackets are used to denote optional choices while parenthesis denotes repetition.

A policy is composed of a name, variables and an `ESCA list`. `PolicyVariables` denote variable declarations that can be used inside `Condition` and `Action` definitions. Our language supports C-style declarations with a wide range of primitive types (e.g., int, float, double, char and so on) and different type constructors (e.g., vector and record). For instance, the policy of Fig. 3 declares the integer variables `prevValue` and `threshold`. The former is used to remember the last humidity reading while the latter the threshold to represent the maximum difference allowed between two consecutive humidity readings. It is worth mentioning that variables defined inside a policy are not shared with other policies unless the modifier `global` is prefixed to the variable declaration. In this case, the modifier also specifies the policies that share such a variable.

An `ESCAlist` is a list of (`Event`, `SCAList`)-pairs each starting with the keyword **on**. When the event defined in the `ESCAList` is generated, then the action defined in the corresponding `SCAList` is triggered if `condition` is true. In the following, we describe in more detail the specification of `Event` and `SCAList`.

4.1.1. `Event`

Two types of events can be defined: `PubSubEvent` and `Timeout`. An `PubSubEvent` event defines pub/sub operations that are executed both by the components and the brokers. A `Timeout` event id of the form **timeout** `(t)` and is generated by the Policy Manager itself when no pub/sub events are observed within the time interval `t`.

```
1  Policy = PolicyName PolicyVariables ESCAlist
2
3  ECSAlist = on Event SCAlist [ ECSAlist ]
4
5  SCAlist= CurrentState'-'NewState':' Condition '→' {Action;} Outcome; [ SCAlist ]
6
7  Event = PubSubEvent |  Timeout
8
9  PubSubEvent = Direction PubSubOp Topic
10
11 TimeOut = timeout '(' unsignedint ')'
12
13 Direction = B-C | C-B | B-extB | extB-B
14
15 PubSubOp = notify | advertise | subscribe | unadvertise | unsubscribe
16
17 Topic = Topicname [ '(' ActualParameters ')' ]
18
19 Outcome = ( discard | accept ) [ '(' ActualParameters ')' ]
20
21 Action = PubSubOp  Topic  with Extension {',' Extension} | C style statement
22
23 PolicyName = Topicname = Extension = identifier
24
25 PolicyVariables = C style variables
26
27 Condition = C style expression
28
29 CurrentState = NewState = unsignedint | string
```

Fig. 5. The syntax to our language for specifying ESCAPE policies.

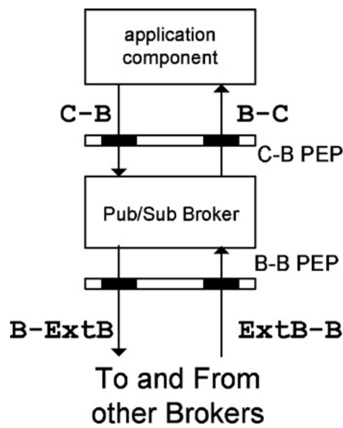


Fig. 6. Events sent by PEPs for each pub/sub interaction type.

PubSubEvent events are generated by the PEPs that intercept all the executed pub/sub operations. Fig. 6 shows the placement of the PEPs in our approach. The PEP placed between the application components and their local pub/sub broker is called a Component-to-Broker PEP. This PEP generates events tagged as **C-B** if the direction of the operation is from the component to the broker (e.g., the component requests the local broker to send a notification to the corresponding subscribers); when the direction of the operation is from the broker to the component then the event is tagged as **B-C** (e.g., a notification is delivered by the local broker to the component). Similarly, the Broker-to-Broker PEP captures pub/sub operations between the local broker and the brokers deployed on other nodes. If the operation is executed from the local broker to an external one (e.g., a subscription that is sent to the brokers where publishers for that notification are registered) then the event is tagged as **B-extB**; on the other hand, if the executed operation represents an incoming message from an external broker (e.g., a notification that arrives from another broker) then the event is tagged as **extB-B** tag.

These tags are used for defining the `Events` in the `ESCAList` using the non-terminal `Direction` as specified in Fig. 5. `PubSubOp` defines the type of pub/sub operations: `notify`, `advertise`, `subscribe`, `unadvertise`, `unsubscribe`. A `PubSubOp` is followed by the `Topic` composed of a `TopicName` and an optional list of `ActualParameters`. For instance, the humidity Topic with its `h_value` in Fig. 3 is notified each time a component sends a humidity notification to its local broker.

4.1.2. `SCAList`

A `SCAList` is always associated with an `Event` and defines the action that is executed when the corresponding event is generated by the PEPs. A `SCAList` can be defined as shown in Line 5 of Fig. 5: before the `Action` is executed the system has to check that the policy is in the `CurrentState` and that the `Condition` is true; when the `Action` is completed then the policy goes to state `NewState`.

States can be identified using either strings or numbers and together with policy variables are used to define the current policy state. Actions and conditions are written in a C-based platform independent language and can refer to policy variables, event parameters, execute any pub/sub operations, and call external libraries. Pub/sub operations are specified using the syntax `PubSubOpTopic`**with**`Extension` where `Extension` specifies protocols and services outside the pub-sub system (see Section 3 for details). Procedures and functions are specified by using our language that provides assignment, if-then-else and while statements. An action must always end either with the outcome **accept** or **discard**. The value **accept** specifies that the pub/sub operation that triggered the policy can be completed after the action execution terminates. When the policy specifies the value **discard** it means that the pub/sub operation that triggered the policy cannot be completed after the action.

To make things more concrete, let us refer to the example of the policy defined in Fig. 3. Fig. 7 provides a message sequence chart of the steps executed from the event generation to the policy activation. When the application component executes the operation `notify Humidity(h_value)`, the Component-to-Broker PEP intercepts the operation and generates the corresponding event that is sent to the Policy Manager. The Policy Manager triggers the humidity policy using the `execute` routine (that will be discussed in more detail in the following section). This policy defines three `SCAList` statements associated with the event. The first is applied when the policy state is 0 (line 5). The action specifies that the variable `prevValue` is set to the humidity readings `h_value`. The second is applied when the policy state is 1 and the absolute value of the difference between the current and the previous humidity readings does not exceed the threshold (line 9). In this case, the value of the `prevValue` variable is updated. The third is applied when the policy state is 1 and the difference between the current and the previous reading exceeds the threshold (line 13). In this case, in line 15 the action executes a notification with a topic that represents the excess in humidity (**notify** Excess). When the Policy Manager completes the action **discard** or **accept** reply is sent back to the PEP. If the reply is a **discard** then no

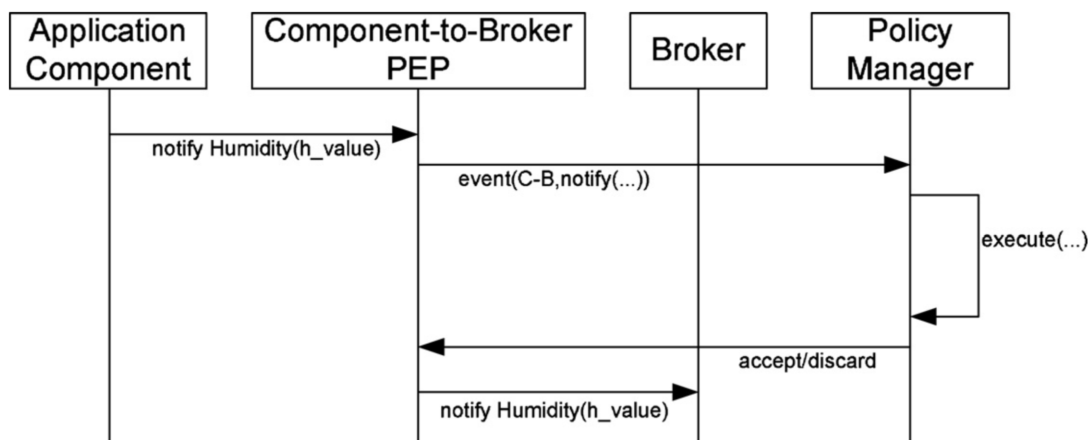


Fig. 7. Message sequence chart of the event generation and policy activation.

```
1  void execute(event currentE)
2   Outcome[]={}; //the set of outcomes for each executed action
3   if no policy defines currentE then
4     accept;
5     return;
6   for each policy p such that p.e = currentE do
7     let p.CS be the current state of the policy p
8     for i = 1 to length(p.e.SCAlist) do
9       if (p.e.cs_i == p.CS) and (p.e.condition_i) then
10        execute p.e.action_i;
11        add p.e.action.outcome_i, p.name to Outcome[];
12        p.CS = p.e.ns_i ;
13        break;
14  if Outcome[] contains no conflicts
15     return decision;
16  else return ConflictManager.takeDecision(currentE,Outcome[]);
```

Fig. 8. Policy execution procedure of the Policy Manager.

further operation is executed. If the reply is an **accept** then the pub/sub operation that triggered the policy is resumed. In this case, the notification operation is forwarded to the broker and executed.

4.2. Policy execution model

In the following, we describe our policy execution model, that is the Policy Manager run-time behaviour. We denote with *P* the set of all policies and $p_1, \ldots, p_n$ are elements in *P*. A policy *p* in *P* defines a set of events $\{e_1, \ldots, e_m\}$. Each event *e* has related a *SCAlist* containing a sequence of elements of the form (cs_i, ns_i, $condition_i$, $action_i$), representing the current state, the new state (after the action is executed), the condition, and action, respectively. In order to refer to one of these elements we prefix it with the event name followed by the symbol ".". For instance if *e* is an event defined in a policy *p* then $p.e.condition_i$ and $p.e.action_i$ denote the action and the condition related to *i* th element in the *SCAlist* of *e* as defined in *p*.

When the Policy Manager receives from the PEPs an event *currentE*, it invokes the `execute` procedure shown in Fig. 8. The procedure takes as an input an event *currentE* and defines a local list `Outcome` that will contain the outcomes of all policies evaluated on the event *currentE*.

In the case that no policies define the event *currentE*, then the related pub/sub operation is performed and the procedure terminates (line 4). On the other hand, for all policies *p* that define an event *e* equal to the current event *currentE* the Policy Manager executes the following steps. First of all, it retrieves the current state *CS* of the policy *p* and finds the first state-condition-action (in the following referred to as an *element*) related to *p.e* such that the state $p.e.cs_i$ is equal to the current policy state *CS* and the condition defined in the element $p.e.condition_i$ is satisfied (line 9). If this is the case, then the action $p.e.action_i$ is executed (line 10). The outcome of the action $p.e.action_i$ together with the identifier of policy *p* (*p.name*) is inserted on the `Outcome` vector (line 11). Finally, the current state of the policy *p* is update to the state $p.e.ns_i$ (line 12).

If all the actions output the same value (either **accept** or **discard**) then no conflict is present in the vector and the decision is returned to the PEP (lines 14 and 15). On the other hand, if both the statements are present in the vector then a conflict is present. To solve such conflicts, our framework provides a *Conflict Manager* that is invoked by the `execute` procedure (line 16). Fig. 9 provides a message sequence chart of the steps executed for solving conflicts.

The resolution of conflicts can be done using several strategies. We allow the policy writer to specify which strategy should be used according to the event that triggered the policy execution defining (event,strategy)-pairs. During execution, the Conflict Manager selects the strategy associated with the event that matches the *currentE*. The framework offers two strategy implementations but it is possible to implement other strategies and make them available in the framework.

The *Default Strategy* is associated with all events that are not associated with any strategy. This strategy always outputs a **discard** when at least one **discard** is present in the vector `Outcome`. A simple variation of this strategy could be one where the percentage of tolerated **discard** values is provided. For instance, if the value is set to 30% and the percentage of **discard** values present in the vector is below that threshold then an **accept** is output. Otherwise the strategy outputs a **discard**. The second strategy available in our framework is the *Table Strategy* that utilises a table where a policy writer associates events with the value that should be returned each time a conflict is detected.

More complex strategies can use Extension Components described in Section 3. For instance, it is possible to specify a *resource-oriented* strategy that takes into account the actual level of battery charge of the sensor when a decision is taken. The battery charge can be calculated by an Extension Component. In order to reduce power consumption, each time a conflict is detected and the remaining charge is below a giving threshold then the messages are not transmitted and hence discarded.

The Policy Manager is responsible not only for enforcing policies but also for *loading* new policies, *disabling* policies, and *updating* existing policies at run-time.

Because we compile policies and because of the restrictions imposed by the TinyOS2 implementation, it is not possible to download new code while a sensor is in execution mode. This means that the Policy Manager can dynamically load during run-time only policies that are already compiled into the code bundle and available in the sensor. When a new policy is loaded, then the Policy Manager can execute its actions when the associated events are generated. On the other hand, when a policy is disabled its actions are no longer executed. The granularity of code enabling/disabling

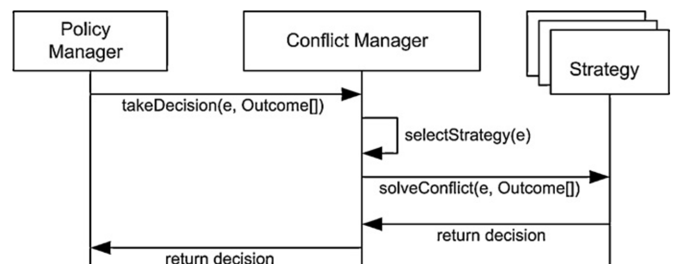


Fig. 9. Message sequence chart of the conflict resolution procedure.

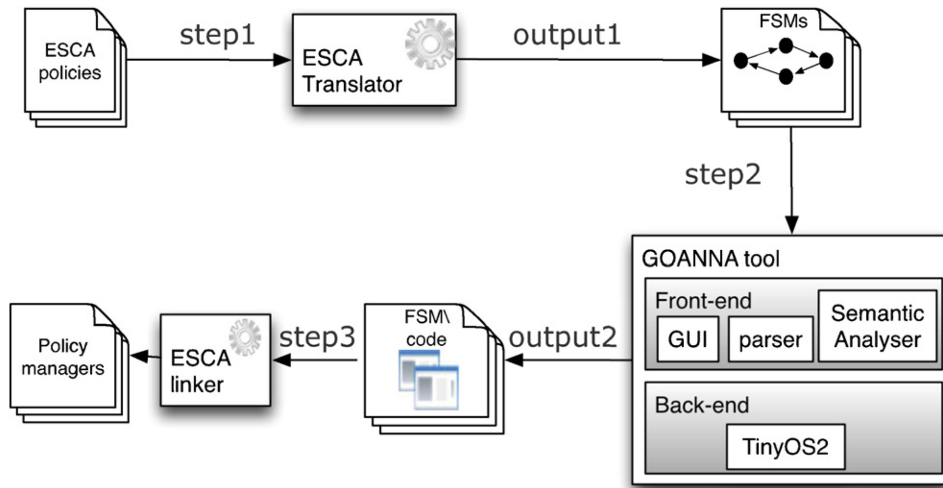


Fig. 10. The ESCAPE tool chain.

```
1 TemperaturePolicy
2   on C-B advertise Temperature(t_value)
3     0-1: true -> accept;
4
5   on C-B notify Temperature(t_value)
6     1-1: t_value<50 -> discard;
```

Fig. 11. An example Policy.

can be extended to the level of each ESCA rule defined in a policy. In this way, our framework can support the updating of policy definitions during run-time.

4.3. Tools and policy analysis

The ESCAPE framework provides tools that take as input policy definitions and generate code. There are 3 main steps performed these tools depicted in Fig. 10.

In the first step, the *ESCAPE policy translator* is applied to perform both syntactic and semantic checks and it translates the ESCAPE policies into a finite state machine to perform semantic checks strictly related to the pub/sub operations such as *correct operation ordering* and *well-formed policies*. A correct operation ordering check analyses the correct ordering among operations, i.e., a **notify** is preceded by a **advertise**. The well-formed policy check verifies that each action always ends with either an **accept** or **discard**.

In the second step, the *GOANNA tool* generates the state machine implementation. The GOANNA tool is composed of a front-end and a set of back-ends. The front-end provides three main components: a *GUI*, a *parser* and a *semantic analyser*. The GUI implements a graphical tool to view the policy as a state machine in a graphical form. For instance, Fig. 11 shows an example of a ESCAPE policy and Fig. 12 shows the same policy loaded in the GOANNA GUI as a state machine. The semantic component performs semantic checks on state machine specifications, such as *state reachability* and *recursive event detection*. State reachability ensures that each event-state-condition-action can be applied, i.e., all states inside a state machine definition can be reached. Recursive event detection avoids policies leading to livelock. In the simplest case, we can have a policy in which an event *e* is defined, the new state is equal to the current one (a transition that enters and exits in the same state) and its action defines a **notify** of the same event *e*. In this case, the policy can generate an infinite number of events *e* without making any *real* progress. Generally speaking, a policy can define a chain of events that produces livelock. The tool tries to visit each state machine, detect possible livelock conditions and produce warnings. The back-end is used to translate the state machine specifications into platform-specific code for different implementations. For instance, we have developed a TinyOS2 back-end and Java back-end. The former produces a state machine implementation that can be loaded by the TinyOS2 operating system while the latter by the Java VM.

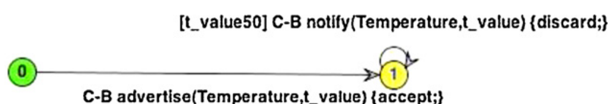


Fig. 12. GUI provided by the GOANNA tool.

In the third step, we apply the *ESCAPE linker* in order to produce the code bundle comprising the Policy Manager and the policies (in the form of state machine implementations). The ESCAPE linker links together state machine implementations, the conflict manager, all pub/sub services and the third-party libraries. Note that, this step is not automatic. For instance legacy libraries may need to be included by the users during the generation process.

5. Case study

This section presents a case study related to cultural asset transportation service used to securely move cultural assets from one venue (museum) to another. The service was developed as part of the EU CUSPIS project (CUSPIS, 2007).

In the transportation service, a lorry transports a set of packages each containing a cultural asset. As shown in Fig. 13, the lorry is equipped with sensors and actuators on which several sense-and-react applications are deployed. During transportation, the lorry is

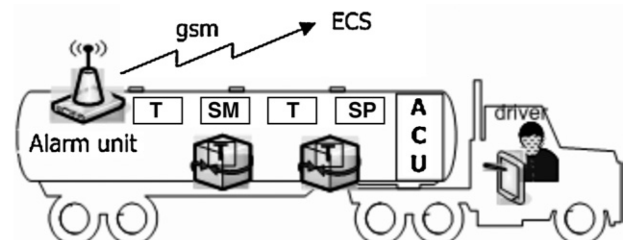


Fig. 13. Deployment of sensors and actuators for the CUSPIS case study.

```
1 FATempPolicy
2   on C-B notify TempForFireAlarm(t_value)
3     0-0: t_value<50 -> discard;
4
5   on B-extB notify TempForFireAlarm(t_value)
6     0-1: t_value>50 ->
7               notify TempForFireAlarm(t_value) with ReliableCPC, PushDPC;
8               accept;
```

Fig. 14. Temperature policy for the fire alarm application.

monitored by the Emergency Central Station (ECS) that is in contact with police and emergency units (such as fire fighter stations). To send alarms to the ECS, the lorry is equipped with multiple alarm units that include a GSM transmitter and GPS sensor. The lorry driver can use a portable wireless computer, such as a PDA, to check sensor readings and be notified in case of any alarms.

The following sense-and-react applications are deployed:

- *Fire Alarm Application* is responsible for detecting and taking initial actions against the fire inside the lorry. Temperature sensors provide readings for the actual temperature and smoke detectors are used for sensing the presence of smoke. If the temperature rises over a given threshold and the smoke detectors provide positive smoke readings then the water sprinklers must be activated and a fire alarm sent. Issuing a fire alarm activates the alarm unit that informs the lorry driver through the PDA and sends an alarm message to the ECS.
- *Air Conditioning Application* is responsible for maintaining temperature and humidity within the lorry to given values. Temperature sensors (shared with the Fire Alarm Application) and humidity sensors provide readings of the air quality in the lorry. An air conditioning unit (ACU) uses the readings from the sensors to increase or decrease the temperature and humidity to keep those values within the target values set by the driver.
- *Package Tampering Monitor* is responsible for the integrity of the packages containing the artefacts and to raise an alarm in case the packages are tampered with. Each package contains sensors that collect readings for temperature, humidity and light. An indication that a package was opened can be signalled when a reading deviates significantly from the previous values. For instance, when the package is opened the amount of light and temperature inside the package increases and such variation can be captured by the sensor. If this is the case, then the sensor notifies the driver's PDA and the alarm unit. The latter sends an alarm (together with the GPS position) to the ECS to summon the intervention of the police.

In the following, we elaborate on the application requirements and policies showing how such requirements are met with our approach.

5.1. Fire alarm application

This application makes use of the following sensing and actuator devices to detect fire in the in lorry. Temperature sensors provide readings for the actual temperature and smoke detectors are used for sensing the presence of smoke. If the temperature rises over a given threshold and the smoke detectors provide positive smoke readings then the water sprinklers must be activated and a fire alarm sent. Issuing a fire alarm activates the alarm unit that informs the lorry driver (through his/her computer) and sends through the GSM link an alarm message combined with the actual GPS position of the lorry.

The components deployed on the temperature sensors provide readings of the actual temperature. Each component is a publisher of a notification of type `TempForFireAlarm(t_value)`. Subscribers of the temperature notifications for the fire alarm application are the components running on the sprinklers. Instead of flooding the network with every temperature sampling, only notifications with meaningful values should be allowed to leave the publisher node. In particular, for the fire alarm application, only samplings with values over 50 should be allowed. By filtering out notifications we minimise energy consumption. This behaviour is captured by the policy in Fig. 14. When the component sends a notification, the policy checks whether the value of the temperature is less than 50 (line 3). If this is the case, the notification is discarded. However, when the sampled temperature is over 50 (line 6), the notifications should be disseminated as quickly and reliably as possible. In this case, the notifications are associated with the `PushDPC` component that implements the push-model (line 7) using the `ReliableCPC` for a reliable communication protocol.

The sprinkler nodes are responsible for operating the water sprinklers when a fire alarm is detected. Two conditions have to be verified before a fire alarm is sent: first the temperature must be over 50 and second a smoke reading must be collected to confirm the presence of flames in the lorry. This is important to avoid false fire alarms that could be sent because the lorry is exposed to the direct heat of the sun and the air conditioning unit is not working properly. To make the code of the sprinkler component as simple as possible in line with the philosophy of our approach, the verification of the high temperature value and the presence of smoke is specified in the `SprinklerPolicy` deployed on the sprinkler nodes and shown in Fig. 15. The sprinkler component is simply a subscriber of fire alarm notifications: when the component receives such a notification the water sprinkler is opened. When the sprinkler component subscribes for the notification of type `FireAlarm`, the policy captures such an action (line 2) and advertises the node as a publisher of two notification types: `FireAlarm` (in this way the node is both a publisher and a subscriber of this notification type) and `GetSmokeReading` (lines 4 and 5). Moreover, the policy subscribes to notifications of types `TempForFireAlarm(t_value)` and `Smoke(s_value)` (lines 6 and 7). When the actions are completed the policy goes from state 0 to 1. If a notification for the temperature value reaches the node (line 10), the policy is activated and a notification of type `GetSmokeReading` is sent using push mode and a reliable delivery protocol (lines 12). This notification proactively triggers the smoke detectors to provide smoke samplings (more details on this type of interactions are given below). When a notification from the smoke detectors arrives (line 15), the policy checks whether the smoke is detected (indicated by `s_value` set to true). If this is the case, a `FireAlarm` notification is sent (line 17). When this notification is delivered to the sprinkler components the sprinklers are activated.

In this scenario, the smoke readings are necessary only when an action is about to occur. Generating this data continuously would result in wasting energy. On the other hand, when the data is required, fresh data needs to be generated. This type of *proactive interaction* where the sensing device generates data on-demand is typical of sense-only scenarios but they are still useful in sense-and-react scenarios since data can be generated more efficiently. To support this type of interactions maintaining all the advantages of our approach, a policy as the one shown in Fig. 16 can

```
1  SprinklerPolicy
2    on C-B subscribe FireAlarm
3      0-1: true ->
4              advertise FireAlarm;
5              advertise GetSmokeReading;
6              subscribe TempForFireAlarm(t_value);
7              subscribe Smoke(s_value);
8              accept;
9
10   on B-C notify TempForFireAlarm(t_value)
11     1-2: true ->
12             notify GetSmokeReading with ReliableCPC, PushDPC;
13             discard;
14
15   on B-C notify Smoke(s_value)
16     2-3: s_value ->
17             notify FireAlarm with ReliableCPC, PushDPC;
18             discard;
19
20     2-1: !s_value -> discard;
```

Fig. 15. Policy deployed on the sprinkler nodes.

```
1  SmokePolicy
2    on C-B advertise Smoke(s_value)
3      0-1: true ->
4              subscribe GetSmokeReading;
5              accept;
6
7    on B-C notify GetSmokeReading
8      1-1: true ->
9              s_value = getData();
10             notify Smoke(s_value) with ReliableCPC, PushDPC
11             discard;
```

Fig. 16. Policy deployed on smoke detectors.

be used. The component running on the smoke detector registers itself as a publisher of `Smoke(s_value)` notifications. This advertisement triggers the activation of the `SmokePolicy` (line 2) that subscribes the node to notifications of type `GetSmokeReading` (line 4). When a `GetSmokeReading` notification arrives (sent by the `SprinklerPolicy` policy described above), the policy generates a fresh sampling of the data via the `getData()` operation (line 9) and publishes this data through a notification (line 10).

5.2. Air conditioning application

The air conditioning application uses temperature sensors (shared with the fire control application) and humidity sensors for sampling the quality of the air in the lorry. The ACU controls the quality of the air using readings from the sensors to increase or decrease the temperature and humidity to keep those values within the target values set by the driver's GUI. The publishing of the readings from these sensors can use a "send-on-delta-or-zero" (SDZ) approach. According to this approach, a notification is published only if the difference between the actual value measured by the sensor either is more than a delta from the target value or it is equal to zero. This approach is more efficient in terms of energy consumption since a notification is published only if it is really required. In fact, in the first case it means that the ACU has to be activated to bring the monitored values within the desired target; while in the second case the ACU can be switched off since the desired target is reached.

This content-based filtering can be specified by a policy as shown in Fig. 17. When the component sends a notification (line 3), the policy uses the predicate `deviateOrZero` (provided by the Sampling Buffer Component (`SBC`)) to check whether the temperature value needs to be published (either deviates more than a delta from or is equal to the target (line 4)). If not, then it is simply discarded (line 5). When the notification has to be sent the pull model with the unreliable communication protocol is used (implemented by the `PullDPC` and `UnreliableCPC`, respectively) (line 9).

The target and delta values are set by the driver's GUI application. To this end, the temperature sensors need to be subscribers for notifications from this application. This notifications are used for setting these values. The policy in Fig. 18 takes care of subscrib-

```
1  ACSTempPolicy
2    global int target_t, delta_t;
3    on C-B notify ACSTemp(t_value)
4      0-0: !SBC.deviateOrZero(target_t, delta_t,t_value) ->
5              discard;
6
7    on B-extB notify ACSTemp(t_value)
8      0-0: SBC.deviateOrZero(target_t, delta_t,t_value) ->
9              notify ACSTemp(t_value) with UnreliableCPC, PullDPC
10             accept;
```

Fig. 17. Temperature policy for the air conditioning application.

```
1  SettingPolicy
2    global int target_t, delta_t;
3    on C-B advertise ACSTemp(t_value)
4      0-1: true ->
5              subscribe TempControl(target_t_value, delta_t_value);
6              accept;
7
8    on B-C notify TempControl(target_t_value, delta_t_value)
9      1-1: true ->
10             target_t=target_t_value;
11             delta_t=delta_t_value;
12             discard;
```

Fig. 18. Policy for setting the target and delta values for the temperature readings used by the air condition application.

ing the sensors and updating the global variables accordingly. The variables `target_t` and `delta_t` (line 2) are shared between all the policies deployed in the same node. The policy registers the node as a subscriber for the temperature control values (line 5). When this notification is delivered to the node, the target and delta variable are updated accordingly (lines 10 and 11).

5.3. Package tampering application

This application is responsible for monitoring the integrity of the packages containing the artefacts and for raising an alarm in case the packages are tampered with. Each package contains sensors that collect readings for temperature, humidity and light. An indication that a package was opened can be signalled when a reading deviates significantly from the previous values. For instance, when the package is opened the amount of light and temperature inside the package increases and such variation can be captured by the sensor. If this is the case, then the sensor sends an alarm notification. The alarm unit and the driver's GUI respond to this notification by sending out alarm information to the driver and the police.

However, care must be taken to avoid notification of false alarms. For instance, if the temperature in the package goes up it could be the effect due to the increase of temperature inside the lorry (i.e., the air conditioning unit is not working properly). If this is the case, then the sensors in the other packages should also register an increase in temperature. Therefore, before sending the alarm notification, the sensor that first registers an increase of temperature sets off a timer waits for notifications from the sensors in other packages that signify an increase in temperature. If these notifications from other sensors arrive before the timer timeouts then no alarm is sent. Otherwise, the alarm notification is sent.

This behaviour can be codified using a policy as shown in Fig. 19 (note that to improve readability we removed all details related to diffusion and communication protocols). This policy captures only the case for variations in temperature readings. The policy starts registering the node as a publisher for the `PackageAlarm` notification and as subscriber of the `PackageTemperature(t_value, node_id)` notifications. The publishing of the temperature readings uses a "send-on-delta" approach, where a notification is published only if it varies more than a given delta from the previous published notification. The predicate `deviateDelta` is used for checking whether the difference between the actual reading and the previous published one is greater than delta. If the difference in not more than delta, the notification is discarded (line 10). Otherwise, if the notification deviates more than delta, the notification is sent (line 12). At this stage, the following can happen:

- a notification arrives but is the one that was just sent by the node itself (line 15). In this case, the state of the policy is not changed.
- a notification arrives from other nodes (line 18). This means that the increase of temperature is not local to this package but other sensors are registering it as well. Therefore this is a false alarm and should be ignored.

```
1  TamperingPolicy
2   on C-B advertise PackageTemperature(t_value,node_id)
3     0-1: true ->
4             advertise PackageAlarm;
5             subscribe PackageTemperature(t_value,node_id);
6             accept;
7
8    on C-B notify PackageTemperature(t_value,node_id)
9      1-1: !deviateDelta(t_value, delta_value) ->
10             discard;
11     1-2: deviateDelta(t_value, delta_value) ->
12             accept;
13
14   on B-C notify PackageTemperature(t_value,node_id)
15     2-2: node_id == this.node_id ->
16             discard; \\ignore: this is my notification
17
18     2-1: node_id != this.node_id ->
19             discard; \\false alarm: increase in temp in other sensors
20
21   on timeout()
22     2-3: true ->
23             notify PackageAlarm;
24             discard;
```

Fig. 19. The policy for setting off alarm notifications when the packages are tampered with.

```
1  SecureGroupPolicy
2    on B-extB notify PackageTemperature(t_value,node_id)
3      0-0: true ->
4                e_temp = EC.encrypt(t_value,K_g);
5                e_node_id = EC.encrypt(node_id,K_g);
6                accept (PackageTemperature, e_temp, e_node_id);
7
8    on extB-B notify PackageTemperature(e_t_value,e_node_id)
9      0-0: EC.test(e_t_value,K_g) && EC.test(e_node_id,K_g) ->
10               t_value = EC.decrypt(e_t_value,K_g);
11               node_id = EC.decrypt(e_node_id,K_g);
12               accept (PackageTemperature, e_temp, e_node_id);
13
14     0-0: !(EC.test(e_t_value,K_g) || EC.test(e_node_id,K_g))->
15               discard;
```

Fig. 20. The policy for the crypto operations on notifications sent within the secure group.

- the timer expires and a timeout event is sent (line 21). This means that no other sensors registered the increase in temperature. In this case a `PackageAlarm` notification is sent (line 23).

5.4. Secure zoning

In WSNs, it is possible to realise a secure group communication where the participant nodes can protect the confidentially of their communication. The secure group communication scheme is implemented by means of symmetric encryption. A group is composed of different members that share a secret key, e.g. each secure group G has related a secret key K_g. The distribution of the K_g is done as an out-of-band bootstrapping process to the sensors where the participant components are deployed. Policies control the use of encryption/decryption operations provided by the *Encryption component* (EC) (see Section 3).

As an example of a policy for creating a secure zone of communication, let us consider the case of the sensors inside the cultural asset packages discussed in the previous section. An attacker that discovers the values transmitted by the sensors inside the package can put the package in an environment with the same values avoiding the detection of package unwrapping. In each sensor that forms the package tampering monitoring application, the `SecureGroupPolicy` shown in Fig. 20 is deployed together with the `TamperingPolicy`.

This policy executes the encryption/decryption operations on the notifications that are sent/received by the sensors of the group. The group key `K_g` is provided to each sensors before their deployment in the packages. When the notification containing the temperature of the package is going to be sent to other sensors, it is intercepted by the policy and the values of the temperature and the node id are encrypted with the group key (line 4 and 5). The notification tuple with the last two fields encrypted is accepted. On receiving a `PackageTemperature` notification, the policy first tries to test whether the values where encrypted with the key `K_g` (line 9). If this is the case, the action is executed where the values are decrypted (line 10 and 11) and the notification accepted (line 12). If the values are not encrypted with the key of group G, then the condition in line 14 would held true and the notification is discarded (line 15).

6. Evaluation

The ESCAPE programming model is based on the component-based approach where the extra-functional concerns of an application are separated into components with well-defined interfaces. Components are then integrated into larger assemblies and complete applications. In ESCAPE, application components implement the basic sensing and control functionality of sensors and actuators. For instance, a temperature sensor would include an application component that provides temperature readings. Extra-functional concerns such as the protocols used for temperature dissemination, and secure communication are specified separately. The integration of components is performed by middleware. Although affording greater clarity and flexibility, separation and integration of concerns can also lead to additional overheads in terms of CPU-time and memory usage. In this section, we elaborate on the advantages of our programming model in comparison with other two approaches (Section 6.1) and also quantify the overheads that the approach carries (Section 6.2).

6.1. ESCAPE programming model

In this section, we discuss the advantages of our approach compared to TinyOS and the TeenyLIME middleware (Costa et al., 2007). We selected TeenyLIME because it was specifically designed for sense-and-react applications in WSNs. TeenyLIME provides a programming model based on the tuple space abstraction where application components communicate by means of tuples. TeenyLIME and other related approaches are also discussed in Section 8. Fig. 21 summarises the properties offered by ESCAPE compared with TinyOS and TeenyLIME.

Both ESCAPE and TeenyLIME achieve a clear separation of application functionality by means of application components. For instance, the functionality for providing temperature readings can be encapsulated in a temperature component while humidity readings can be encapsulated in another dedicated component. As such, components are units of reusability that can be redeployed *as is* to compose more complex application functionalities. Additionally, ESCAPE provides two more units of reusability – ESCAPE policies and Extension Components. For example, the Push Diffusion Protocol Component can be reused in any deployments where a

Approach	Reusability	Extendability	Flexibility	Multi-hop
ESCAPE	Very High	Very High	Very High	Fully Supported
Teeny LIME	High	High	High	Partially Supported
TinyOS	VeryPoor	Poor	Poor	Unsupported

Fig. 21. Comparison between the ESCAPE, TeenyLIME and plain TinyOS approaches.

fast delivery of the events is required. In TeenyLIME, such aspects are more intertwined with the actual implementation of the middleware. If a specific diffusion protocol has to be used then the actual implementation of the TeenyLIME middleware needs to be changed. Although TinyOS provides a component model, the model fails to provide support for separating and reusing extra-functional concerns (Costa et al., 2007).

In terms of extendability, adding new application functionalities in both TeenyLIME and ESCAPE can be achieved by adding new application components. For instance, if the level of light needs to be measured an application component that senses and reports such values can easily be deployed in both approaches. However, if new extra-functional support is needed, then ESCAPE framework provides a cleaner approach by using ESCAPE policies. For instance, if events need to be encrypted then an encryption component can be simply added in ESCAPE by implementing the appropriate Extension Component. Then, the encryption component can be enabled by means of an ESCAPE policy. In TeenyLIME, new extra-functional concerns can be added in the middleware without modifying the application components. However, the code of the TeenyLIME middleware does not provide specific hooks for such extensions as in ESCAPE. Therefore, adding code to enable the encryption of the tuples requires extensive recoding of the middleware. As for TinyOS, matters are even worst since the new code has to be added directly into the application components.

As for flexibility, if an application needs to be re-deployed in a new environment with different requirements then support is needed for application adaptation. Application functionality can be changed in all approaches by changing the corresponding application components and/or configuration. ESCAPE however, goes further by its inclusion of a policy-driven layer that can be used to tailor and dynamically reconfigure extra-functional concerns.

Finally, an important consideration in WSNs is support for multi-hop WSNs. ESCAPE was designed from the onset as a multi-hop framework where events could be disseminated by dedicated mechanisms through all or part of the network of nodes. It allows for the design of applications that can easily gather global, regional and subscribed-only information. TeenyLIME's one-hop architecture can automatically forward a tuple inserted into a node to a nodes one-hop. To globally distribute a tuple it is necessary to explicitly program every node to keep forwarding such a tuple. In a pure TinyOS implementation, a multi-hop mechanism can be implemented by flooding the network with every message and coding directly within the application code the routing control. As a result, such a multi-hop mechanism is not efficient because can waste resources of the nodes that are not interested in the message.

6.2. Run-time evaluation

In this section we evaluate the overheads of ESCAPE in terms of memory, CPU-time, message exchange and power consumption. We perform our experiments using the fire alarm application (see Section 5) running on Tmote Sky nodes and simulations of up to 120 sensors.

In Fig. 22, we show the overall system architecture and the standard ESCAPE components: the Policy Manager, the Broker, the Push and Pull diffusion protocols. Additional ESCAPE components can be added as needed by application or system requirements. The Broker, the Push and Pull diffusion protocols are essential to implement the reliable message passing infrastructure required by the publish/subscribe service. The Policy Manager evaluates policies and manages their life cycle (i.e., it loads, executes and deletes policies). Effectively the only memory overhead introduced is by the Policy Manager since the other parts implement the system requirements and would be required in equivalent approaches.

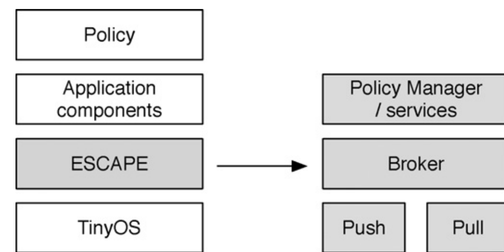


Fig. 22. ESCAPE architecture.

6.2.1. Memory overhead

In Fig. 23 we show information about the code and data size of application components, some policies related to our case study and the ESCAPE components. We emphasise that the Policy Manager size is independent from policy specifications and all other components. In other words the policy manager is a container that manages the policy life cycle so that its size is always the same. In our case the temperature components is bigger than the tampering policy since the temperature component must embed all code needed to sense and to manage the timer (the sensing is performed at each tick) while the policy only embeds a few if statements and a few variables to implement state machines.

Overall our framework requires a small amount of additional memory. In fact the Policy Manager only requires 1120 bytes while the reliable communication infrastructure (the Policy Manager and PushDPC) requires 2530 bytes, which is less than the Temperature component.

6.2.2. Message exchange overhead

Message exchange overhead is a consequence of the diffusion protocol required to route data between publishers and subscribers. More specifically in the pull model, subscriptions must be distributed and must be updated while delivery routes are updated. In the push model subscriptions are kept locally but there is still the additional overhead needed for the notifications to seek the subscribers.

Our pull mode is implemented by an optimised flooding protocol that requires for each subscription n messages, with n equal to the number of nodes. More specifically when a new subscription is detected the local broker broadcasts a **pull** message containing the address of the local node (2 bytes) and the ID of the subscription (2 bytes). This message is forwarded by all brokers (no more than once) in order to reach all potential publishers. After a publisher receives the message it uses both the address and the interest of subscribers in order to notify new events. In the case of dynamic systems (publishers added at run time) the broker keeps sending pull messages in order to maintain the table of the new subscribers updated. Various optimisations are also performed. For instance our compiler works out when different subscriptions can be sent together in a single message. In this way the message overhead of the pull mode is reduced to a single message that is sent to all nodes for each new subscriber.

Our push mode is implemented by a different flooding protocol that requires for each new notification, n messages, where n is equal to the number of nodes that subscribe to the message. More specifically the new tuple is forwarded by all brokers in order to reach all potential subscribers. Again various optimisations are performed. For instance different notifications can be sent in the same message.

In Figs. 24 and 25 we show the message overhead related to our case study. More specifically, Fig. 24 shows the over-

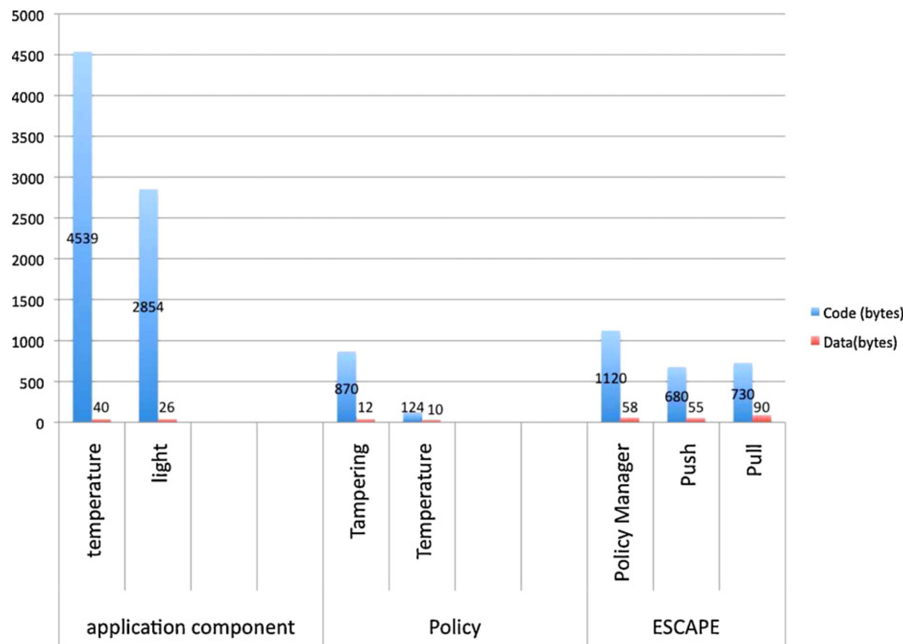


Fig. 23. Code and data size information.

Message type	Message	Mode	Messages per node
Advertise	`TempForFireAlarm`	Push	0
Notify	`TempForFireAlarm`	Push	1 broadcast
Advertise	`FireAlarm`	Push	0
Notify	`FireAlarm`	Push	1 broadcast
Subscribe	`Smoke`	Push	0
Notify	`Smoke`	Push	1 broadcast
Subscribe	`GetSmokeReading`	Push	0
Notify	`GetSmokeReading`	Push	1 broadcast

Fig. 24. Sprinkler message overhead.

head of messages related to the Sprinkler policy. All notifications are performed using the push mode so that subscriptions and advertisements have no cost in term of messages while notifications cost exactly one message per node. Fig. 25 shows the overhead of messages related to the SettingPolicy policy (related to the air conditioning system). In this case a pull strategy is adopted so that the initial subscription requires 1 broadcast message for each node but the notification requires a single message.

6.2.3. *Execution overhead*

The execution overhead is a consequence of the Policy Manager and the diffusion protocols. The Policy manager is executed every time the instrumentation point detects a new pub/sub event. More specifically the instrumentation locally calls the Policy manager which performs three main functions: (i) loads all policies; (ii) verifies the policy conditions (whether or not the policy must be executed); (iii) executes policy actions and changes policy states. Conditions and actions can be arbitrary code, but are typically simple and implement some aspect of the system requirements. Therefore the overhead introduced by the Policy manager is consequence of the time required to call the PolicyManager procedure and load all policies.

In Fig. 26 we show the overhead added by our approach with respect to a basic TinyOS implementation. We consider the requirement implemented by the FATempPolicy policy, that is, an alarm is sent when the temperature is high. In our TinyOS solution this has been implemented by linking together the application components, the policy code and the communication layers. In the ESCAPE solution we have added the extra layer of indirection implemented by the Policy Manager. The same experiment was also performed for the ACSTempPolicy policy.

Message type	Message	Mode	Messages per node
Advertise	`ACSTemp`	Pull	0
Notify	`ACSTemp`	Pull	1
Subscribe	`TempControl`	Pull	1 broadcast

Fig. 25. ACSTemp policy message overhead.

Operation	TinyOS	ESCAPE	Overhead
FATempPolicy	2.5 ms	2.7 ms	8%
ACSTempPolicy	2.6 ms	2.9 ms	11.5%

Fig. 26. Policy manager overhead.

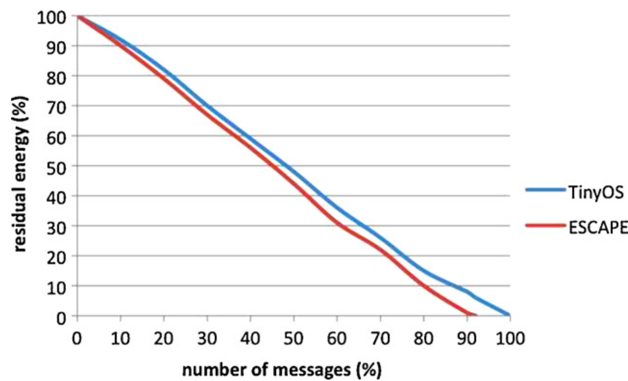


Fig. 27. Average of the residual computational power for configuration.

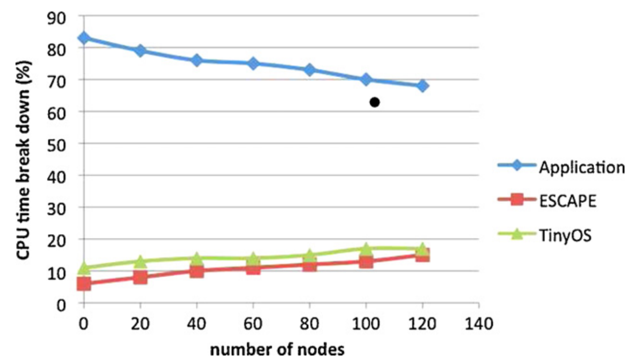


Fig. 28. CPU time breakdown.

6.2.4. *Power consumption*

The diffusion protocol requires control messages to be sent in order to support the implementation of global policies, i.e., policies that can correlate data related to distributed sensors. For instance in the fire alarm application no matter which sensors detect the high temperature and the presence of smoke the sprinkler must be activated. Our approach allows the description of a distribution independent policy where the sprinkler can get event notifications that can require low level multi-hop routing. This is an abstraction with respect to the plain TinyOS implementation which does not support multi-hop communication. In this section we relate the overhead introduced by our diffusion protocol to the power consumption.

We ran systems simulating up to 118 smoke and temperature sensors and 2 sprinklers. More specifically each smoke and temperature sensor sent a random reading every 400 ms.

In Fig. 27 we show the comparison of our implementation and the TinyOS one. This is performed on the percentage of the draining of the sensors batteries, with respect to the percentage of messages sent. Note that transmission and reception operations are much more expensive than local computations. According to the consumption values expressed in Heinzelman et al. (2000) transmitter and receiver electronics consume an equal amount of energy per bit, namely 5 nJ/bit. The experiment shows that our approach only has a small message exchange overhead since the TinyOS implementation still requires to broadcast the event to deliver notifications. For instance Temperature and Smoke policies have no message overhead with respect to the TinyOS implementation since they use of the push mode, i.e., they broadcast the message only when needed. The ACSTemp requires only a small amount of control messages to setup the subscriptions but this results in a later increase since, once the addresses of the subscribers have been delivered, notifications are sent using point-to-point communications instead of broadcasts.

6.2.5. *CPU time breakdown*

We have measured the CPU times of our implementation. More specifically we have partitioned our code in three layers: application, ESCAPE and TinyOS. The application layer contains all application components, the ESCAPE layer contains the policy manager and part of the diffusion protocol implementation all core services and the reliable communication infrastructure. We have simulated an increasing number of sensors (from 5 up to 118 temperature and smoke sensors plus 2 sprinkler) and obtained the results shown in Fig. 28. This emphasises how the extra time spent inside the ESCAPE implementation is a small with respect to the rest of the logic.

7. Discussion

In this section, we will discuss how ESCAPE addresses the requirements listed in Section 2. In order to maximise reuse of application code, our framework makes a clear separation between the application functionality and its control rules. The code of application components fulfils basic functional requirements of an application: for example, a temperature component simply provides temperature readings. Such a simple component code has the advantage to be easily reused in other applications because it does not contain details related to the actual deployment. Details regarding the environment in which the application components are deployed will instead be specified using ESCAPE policies. ESCAPE policies can impose how the functionality of a component can be used by the applications. For instance, in order to save energy the temperature reading can only be published when its value is greater then a threshold. The threshold value depends on type of application is going to do with that information. For instance, a threshold value used for an air conditioning application is lower than the threshold for a fire control application. Ultimately, an ESCAPE policy represents a unit of reusability, they provide a straightforward abstraction for specifying details related to the deployment environment instead of having them scattered through the application code fulfilling requirements *R1* and *R3*.

The publish–subscribe middleware provided in our framework makes use of ESCAPE policies as a mechanism for weaving extra-functional concerns into the application (requirement *R2*). For instance, diffusion strategies, communication protocols, the use of cryptographic primitives represent typical examples of extra-functional concerns that can be implemented using several strategies. Instead of having our middleware statically bound to a specific strategy, we provide a set of different strategies implemented as extendable plug-ins. During the execution of an operation, ESCAPE policies are used for selecting the appropriate strategy. For instance, a temperature notification can be delivered using an optimistic protocol when it is used for the air conditioning application. On the other hand, when the temperature reading is used by a fire control application a more reliable delivery strategy should be used. Because the strategies are not hard coded into the middleware code, new strategies can be developed and deployed making the middleware easily extensible. Moreover, the use of different strategies for distributing the notifications supports both localised and distributed computations (requirement *R4*).

Although the publish–subscribe abstraction is more suitable for reactive interactions, there are still cases in which proactive interactions are more suitable (requirement *R5*), for example, saving energy in a sensing node and requesting it to generate the data only

when it is needed. ESCAPE policies address this requirement without the need to change any of the publish–subscribe primitives. The policy for the smoke sensor described in Fig. 16 is a typical example.

8. Related work

The TeenyLIME middleware (Costa et al., 2007) was specifically designed to address the requirements of sense-and-react applications for WSNs. TeenyLIME provides a programming model based on the tuple space paradigm where sensors communicate through a shared memory. TeenyLIME offers a simple but powerful model but is limited in various aspects. The extra-functional mechanisms that are provided by TeenyLIME are fixed to specific hard-coded modules. For instance, in TeenyLIME tuples are distributed according to a communication protocol that supports only one-hop communications. In our case, notifications can be distributed using several diffusion protocols, according to the needs of the applications. See Section 6.1 for a more detailed comparison.

TinyCOPS (Hauer et al., 2008) is a publish–subscribe middleware that uses a component-based architecture for decoupling the publish–subscribe core from choices regarding communication protocols and subscription and notification delivery mechanisms. The middleware can be extended with components that provide additional services (e.g., caching of notifications, extra routing information). The specification of which particular mechanism has to be used is done by means of metadata information that the application components have to provide through the publish–subscribe API. In our case, application components are agnostic of such extra-functional concerns since policies are used for specifying which mechanisms have to be used.

The Mires middleware (Souto et al., 2005) is also a publish–subscribe service. It uses the component architecture of TinyOS 1.x. Like our approach, it uses a topic-based naming scheme. However, differently than in Mires, we can support content-based filtering by means of policies. Although it is possible to introduce new services like aggregation using extension components, the choice of the communication protocols is fixed.

MiLAN (Heinzelman et al., 2004) 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.

9. Conclusions and future work

In this article, we have described the ESCAPE component-based framework for WSNs based on publish–subscribe paradigm that is used to support Event-State-Condition-Action policies. Component applications implement the basic functionality of the wireless nodes (sense and reaction capabilities) while policies govern the extra-functional concerns of the application. Policies are specified using a finite state machine language that includes variables and functions in order to define complex policies. Policies are compiled by a translator that performs semantic checks and generates all code needed to execute them. We have applied our approach to a number of larger case studies, including a sensor network that is deployed in lorries that transport cultural assets between museums. ESCAPE currently runs on Tmote Sky motes running the TinyOS2 operating system.

Acknowledgments

This research was supported by the UK EPSRC, research grants EP/D076633/1 (UBIVAL) and EP/C537181/1 (CAREGRID). The authors would like to thank our UBIVAL and CAREGRID collaborators and members of the Policy Research Group at Imperial College for their support.

References

Braginsky, D., Estrin, D., 2002. Rumor routing algorithm for sensor networks. In: Proceedings of the First ACM Workshop on Sensor Networks and Applications, ACM, Atlanta, GA, USA, October, pp. 22–31.
European Commision 6th Framework Program - 2nd Call Galileo Joint Undertaking. Cultural Heritage Space Identification System (CUSPIS), 2007. www.cuspis-project.info.
Costa, P., Mottola, L., Murphy, A.L., Picco, G.P., 2007. Programming Wireless Sensor Networks with the TeenyLIME Middleware. In: Proceedings of the 8th ACM/IFIP/USENIX International Middleware Conference (Middleware 2007), Newport Beach, CA, USA, November 26–30.
Deshpande, A., Guestrin, C., Madden, S., 2005. Resource-aware wireless sensor-actuator networks. IEEE Data Engineering 28 (1).
Dijkstra, E.W., 1982. Selected Writings on Computing: A Personal Perspective. Springer-Verlag, pp. 60–66.
Hauer, J., Handziski, V., Kopke, A., Willig, A., Wolisz, A., 2008. A Component Framework for Content-Based Publish/Subscribe in Sensor Networks. In: Wireless Sensor Networks, pp. 369–385.
Heidemann, J., Silva, F., Estrin, D., 2003. Matching data dissemination algorithms to application requirements. In: SenSys 2003. Proc. of the 1st international conference on Embedded networked sensor systems, New York, USA.
Heinzelman, W., Chandrakasan, A., Balakrishnan, H., 2000. Energy-efficient Communication Protocols for Wireless Microsensor Networks. In: Proceedings of the 33rd Annual Hawaii International Conference on System Sciences (HICSS), Hawaii, USA.
Heinzelman, W.B., Murphy, A.L., Carvalho, H.S., Perillo, M.A., 2004. Middleware to support sensor network applications. IEEE Network 18 (1).
Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K., 2000. System architecture directions for networked sensors. In: ASPL 2000. Proc. of the ninth international conference on Architectural support for programming languages and operating systems.
Intanagonwiwat, C., Govindan, R., Estrin, D., 2000. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In: Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking, ACM, Boston, MA, USA, August, pp. 56–67.
Intanagonwiwat, C., Govindan, R., Estrin, D., Heidemann, J., Silva, F., 2003. Directed diffusion for wireless sensor networking. IEEE/ACM Transactions on Networking (TON) 11 (1).
Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., Culler, D., 2005. TinyOS: An Operating System for Sensor Networks. In: Ambient Intelligence, pp. 115–148.
Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W., 2005. Tinydb: an acquisitional query processing system for sensor networks. ACM Transactions on Database Systems 30 (1).
Parnas, D.L., 1972. On the criteria to be used in decomposing systems into modules. Communications of the ACM 15 (December 12), 1053–1058.
Rafaeli, S., Hutchison, D., 2003. A Survey of Key Management for Secure Group Communication. ACM Computing Surveys 35 (3), 309–329.
Ratnasamy, S., Karp, B., Yin, L., Yu, F., Estrin, D., Govindan, R., Shenker, S., 2002. GHT: a geographic hash table for data-centric storage. In: Proceedings of the ACM Workshop on Sensor Networks and Applications, ACM, Atlanta, Georgia, USA, September, pp. 78–87.
Russello, G., Mostarda, L., Dulay, N., 2008. ESCAPE: A Component-based Policy Framework for Sense and React Applications. In: Proceedings of 11th International Symposium on Component Based Software Engineering (CBSE-2008), Karlsruhe, Germany, October.
Sloman, M., Magee, J., Twidle, K., Kramer, J., 1993. An architecture for managing distributed systems. In: Proc. 4th IEEE Workshop on Future Trends of Distributed Computing Systems, pp. 40–46.
Souto, E., Guimares, S., Vasconcelos, G., Vieira, M., Rosa, N., Ferraz, C., Kelner, J., 2005. Mires: a publish/subscribe middleware for sensor networks. Personal Ubiquitous Computing 10 (1).
Yavatkar, R., Pendarakis, D., Guerin, R., 2000. A framework for policy based admission control. In: IETF RFC 2753.
Y. Yu, R. Govindan, D. Estrin. Geographical and energy aware routing: A recursive data dissemination protocol for wireless sensor networks. Technical Report TR-01-0023, University of California, Los Angeles, Computer Science Department, 2001.

Dr Giovanni Russello is Senior Researcher at Create-Netand and leads the Security Technical Group. His research interests focus on techniques for privacy and confidentiality enforcement in untrusted environments. Currently, he is leading research projects in the area of usage control and enforcement of privacy legislations in the

ICT. During his period at Philips Research Labs in Eindhoven he applied some of the results of his PhD research to healthcare application scenarios. After moving to the Department of Computing at Imperial College London he developed a decentralised trust-management system for the e-health application domain.

Dr Leonardo Mostarda is a lecturer in the School of Engineering and Information Science at Middlesex University. His research interests include distributes systems, wireless sensor networks, monitoring systems and security. From 2007 to 2010 he was a post-doc student at Imperial College London. There he was working on the Ubival project in cooperation with UCL, Oxford, Cambridge and Birmingham universities. From 2006 to 2007 he was cooperating with the European Space Agency (ESA) on the Cuspis European project. This project included several partners such as Vodafone, IBM and The Italian ministries of Cultural heritage. He received his Ph.D and computer science degree from the University of L'Aquila in 2006 and 2002, respectively.

Dr Naranker Dulay is a Reader in the Department of Computing at Imperial College. He works on techniques and applications of security, trust and privacy applied to pervasive, mobile and internet systems. He has over 100 publications and has served on the programme committees for numerous conferences. He currently leads research projects on pervasive workflows, home networking and medical healthcare, with a focus on usability issues.