# A component and policy-based approach for efficient sensor network reconfiguration

Nelson Matthys, Christophe Huygens, Danny Hughes, Sam Michiels, and Wouter Joosen
IBBT – DistriNet, Department of Computer Science, KU Leuven, 3001 Heverlee, Belgium
Email: nelson.matthys@cs.kuleuven.be

***Abstract*—Contemporary wireless sensor network deployments are long-lived, large scale and resource constrained. Longevity demands mechanisms that support reconfiguration to meet changing application requirements, large scale demands high-level abstractions to support network management and stringent resource constraints necessitate a high degree of efficiency in middleware to support application development and management. This paper presents the Component and Policy Infrastructure (CaPI), a middleware providing two complementary abstractions for contemporary sensor network development and management. A reconfigurable component model supports embedded developers during initial application development and enables management of evolving functional requirements, while an expressive policy language supports the specification and management of behavioural concerns by administrators or domain experts. CaPI provides a clear decoupling between application logic and behaviour, enabling efficient customization and dynamic reconfiguration of application functionality and behaviour.**

***Index Terms*—Policy, Middleware, Wireless Sensor Networks, Component models, Reconfiguration**

## I. Introduction

In recent years there has been a significant and growing interest in Wireless Sensor Networks (WSNs). A WSN is composed of tiny embedded devices, equipped with low-power radios and sensors, known as 'motes'. These motes or nodes self-organize to form large networks that are capable of sensing the physical world. The typical mote platform is resource-constrained with limited power, computation, memory and bandwidth. These resource constraints motivated the pioneers of WSN research to focus primarily on efficiency and this resulted in highly specialized Operating Systems [1] and programming languages [2], designed to realize static but optimized single-purpose WSN applications.

Today's generation of WSNs are now moving out of the lab and into the real world, where they are being deployed at increasing scale [3], [4], [5] and in diverse application domains ranging from safety-critical systems [6], [7] to habitat monitoring [8], [9]. The once dominant model of single-purpose, monolithic WSN applications is being challenged by the concept of dynamic and multi-purpose WSN infrastructure [10], the most extreme example of which is the 'Smart City' vision [5]. This pressure has driven the WSN community to look again at the rich development and management approaches that have been successfully applied in other large-scale distributed systems, which face similar challenges.

Influenced by the characteristics of real-world WSN deployments, contemporary research on WSN development and management takes a more holistic view on the software life cycle. Run-time reconfigurable component models such as OpenCOM [11], RUNES [7] or Lorien [12] have proven to provide a good fit to cost-effectively decompose and develop WSN applications, while offering support for dynamic adaptation of deployed functionality. The ability to customize applications by deploying and changing individual software components on the fly, removes the need for expensive replacements of the entire image running on the mote. However, in a purely component-based approach non-functional concerns such as distribution, security or persistence become tangled with functional application concerns, compromising the reusability of components. An orthogonal and yet equally promising stream of research aims to tackle this problem through policy-based management [13], [14]. This approach allows for a clean separation of functional and behavioural concerns, providing high-level rule based languages that are suitable for system administration [15]. Yet, the current generation of policy-based management approaches for WSNs [16], [17] remain limited to the static and monolithic approach of provisioning application functionality that was used in early WSNs. This limits the extent to which policies can be used to fully manage the evolution of functional concerns.

In this paper, we propose the Component and Policy Infrastructure (CaPI), a reconfigurable middleware platform for building and managing WSN applications. CaPI offers two first class programming abstractions: *components* and *policies*. Components implement coarse-grained and reusable units of software functionality that are used to realize functional application concerns. Resource constraints dictate that these components must allow full access to low-level features provided by heterogeneous OS and hardware platforms. In contrast, policies are a more lightweight abstraction, implemented in an efficient, expressive, platform-independent language that is designed to support customization and management of behavioural concerns at runtime. In contrast to other policy-based management approaches for WSNs [16], [17], CaPI provides a more flexible reconfiguration model that supports dynamic evolution of both functional and non-functional concerns. Specifically, at runtime new components and policies can be added, removed, or recomposed dynamically to meet ever changing application requirements. Our evaluation shows that CaPI provides an expressive policy language and efficient support for the evolution of functional and non-functional requirements with minimal overhead.

978-0-7695-4735-0/12 $26.00 © 2012 IEEE
DOI 10.1109/POLICY.2012.17

The remainder of this paper is structured as follows: Section II further motivates the need for multiple complementary programming abstractions to enable dynamic reconfiguration in WSNs, followed by the identification of concrete requirements for a supporting middleware environment. Section IV presents the design of CaPI, Section V gives concrete examples of CaPI-policies. Section VI presents a detailed evaluation of CaPI. Finally, in Section VII, we discuss related work, after which we conclude with a look at future research.

## II. A COMPONENT AND POLICY-BASED APPROACH TO DYNAMIC RECONFIGURATION

### A. *Reconfiguration and evolution*

The dynamism and longevity of WSN deployments demands support for reconfiguration to optimize system behavior to suit changing environmental conditions and to meet evolving application requirements. Reconfiguration may be caused by a planned adaptation, triggered by a change in the requirements of a particular stakeholder, or the result of self-adaptive behavior. Reconfiguration may also include both the adaptation of functional and behavioural concerns. For example, the GridKit middleware for WSN-based flood monitoring [18] was deployed for three years on a river in England. This system exploited behavioural adaptation to reconfigure application-level and physical-level network protocols based upon changing environmental conditions [6]. Moreover, the functional software base of the system was also subject to evolution as user requirements were refined in response to deployment experiences. For long-lived deployments in dynamic environments, it is infeasible to anticipate all future usage scenarios and environmental conditions at build-time and therefore runtime reconfiguration of both the functional software base and system behaviour becomes essential.

### B. *Multi-level heterogeneity*

Constructing distributed software for WSNs is extremely complex given the heterogeneity of platforms, stakeholders and concerns. WSN platforms are heterogeneous in terms of hardware, operating systems and programming languages. For example, in the Great Duck Island experiment [8], motes running TinyOS [1] and programmed in nesC [2] were interfaced with PDAs running Linux. Platform independent middleware is required to minimize the complexities of platform heterogeneity. Stakeholder heterogeneity manifests itself in various stakeholders with distinct requirements and different skill sets, such as embedded developers, end-users, and network administrators. Hence, different types of abstractions are required to serve the requirements and skill set of these stakeholders. For example, the embedded systems developers who provide functional software components are low-level experts and require abstractions that allow full access to the hardware-specific features of the underlying WSN platform. On the other hand, WSN administrators require abstractions based on high-level goals allowing for fine-grained platform management and optimization at runtime.

We believe WSN middleware should provide both coarse and fine-grained programming paradigms. A fine-grained mechanism based on lightweight policies enables management and optimization of the runtime environment by administrators based upon an expressive rule-based policy language that is a good fit with the typical skill set of a system administrator [15]. On the other hand, coarse-grained approaches based on components facilitate the realization of plain application objectives for various languages and platforms, allowing for optimal resource exploitation. Functional requirements are thus provisioned by node-local services, running on individual nodes and encapsulated via components, while behavioural concerns are a natural fit with policy-driven approaches. The lifespan of functional and behavioural mechanisms is also quite different: the control rules that dictate application behaviour tend to be of a more volatile nature than the artifacts that encapsulate regular application functionality and therefore a fine-grained approach, which incurs minimal disruption during reconfiguration, is likely to be significantly more efficient. In addition, functional adaptation deals with the replacement or addition of new functionality at runtime and in this, coarse-grained adaptations minimize complexity, promote reification and facilitate activities in the management plane.

### C. *Cost and flexibility of various types of runtime reconfiguration*

As argued in Section II-A, enacting changes at runtime in a WSN is essential, however, all reconfiguration comes at a cost. The costs of reconfiguration are comprised of the energy required for wireless transfer of commands and software elements as well as on-node storage and processing overhead. The impact of a reconfiguration approach indicates the potential scope of reconfiguration that is possible. For example, approaches relying on complete image replacement [1] allow for arbitrarily complex reconfigurations, yet incur very high costs due to transmission and replacement of a complete system image. At the other end of the scale, re-parameterization of existing functionality incurs little overhead, yet the scope of change that is possible is much smaller. CaPI aims to achieve maximal flexibility by offering a spectrum of approaches, each of which has a different trade-off in terms of impact and cost.

## III. REQUIREMENTS

Three streams of requirements for middleware to support flexible and efficient WSN reconfiguration can be identified; dedicated abstractions, a flexible execution environment and resource efficiency.

### A. *Dedicated Abstractions*

As argued in Section II, WSNs require support for the evolution of the functional software base as well as behavioural reconfiguration. To support runtime management activities, the functional plane and behavioural plane should remain independent and reifiable throughout the lifetime of the system. This separation of functional and behavioural concerns may be enabled through a combination of components and policies. While component-based abstractions are a good fit for implementing generic and reusable node-local functionality, policies are a good fit for implementing behavioural concerns. Thus, both components and policies must be supported by a coherent middleware framework and preserved as independent entities throughout the life cycle of the system.

### *B. Dynamic reconfigurable execution environment*

The middleware must offer a flexible runtime environment that allows for dynamic reconfiguration at multiple levels:

- Evolution of the functional software base. It must be possible to evolve the functional software base and underlying execution environment through the insertion and removal of supporting software components at runtime.
- Reconfiguration of system behaviour. It must be possible to manage behaviour at runtime by using policies to tailor and optimize the behaviour of functional components.
- Functional reconfiguration. It must be possible to manage functional reconfiguration at runtime. Small-scale functional reconfiguration may involve embedding additional logic within a self-contained policy while large-scale functional reconfiguration may necessitate the use of policies to orchestrate re-composition of functional components and/or behavioural policies.

### *C. Resource efficiency*

WSN nodes are constrained in terms of energy, computational capability, and memory. The primary cause of energy consumption in WSNs is radio communication, followed by the use of flash memory and to a lesser extent the CPU. Thus, WSN middleware and its associated software artifacts must be sufficiently compact to minimize radio communication and memory overhead during deployment. These artifacts must also be computationally efficient, incurring minimal overhead at runtime.

## IV. CAPI: A COMPONENT AND POLICY INFRASTRUCTURE

CaPI is a runtime reconfigurable middleware promoting components and policies as first-class programming abstractions. Figure 1 illustrates the overall architecture of CaPI.

### *A. CaPI components*

Components in CaPI are implemented using the LooCI component model [19]. Components encapsulate coarse-grained, independently deployable units of functionality. Every component possesses a component *type* and a *component identifier*. The component type is a free-form string used to describe the type of functionality the component implements, whereas a component identifier identifies a unique instance of that component within the local context of a single node.

The interaction model adopted in LooCI is event-based. Hence, all components interact with each other using *semantically typed events*. A component's provided interfaces are defined as the set of events the component can produce, whereas the required interfaces of a component are defined as the events the component is capable to process or consume. A

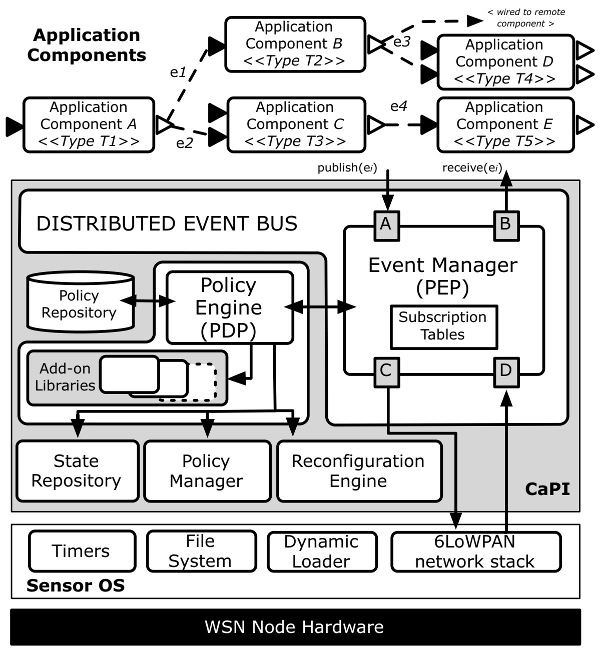


Fig. 1. CaPI's architecture (node-local perspective)

*wiring* or binding is a logical connection between a component's provided interface and required interface. Components are agnostic to these wires since they always publish events to a *distributed event bus* or consume events from this bus. As such, all code implementing distribution concerns is cleanly separated from component implementation, allowing a component to be reused in a variety of distributed configurations. Not only does this event-based style of interaction simplifies application distribution, it eases reconfiguration by introducing a loosely coupled style of system composition.

The *distributed event bus* implements a decentralized publish/subscribe model. A configurable event manager on every node acts as a broker and handles the wires of components by means of subscription tables. Components subscribe to local or remote events via dedicated subscriptions. Local subscriptions represent wires between two components on the same node and contain entries in the following form: *(EventType, ComponentID)* $\rightarrow$ *ComponentID*. Remote subscriptions represent wires between two components residing on different nodes. A remote subscription is split into entries in the subscription tables of both nodes: *(EventType,ComponentID)* $\rightarrow$ *NodeID* at the producer side and *(EventType, ComponentID, NodeID)* $\rightarrow$ *ComponentID* at the receiver. Both the *ComponentID* as well as *NodeID* may be replaced by wild cards, thus allowing the creation of $n$-ary distributed relationships.

All events are semantically typed using a compact encoding scheme based upon the work presented in [20]. All *EventTypes* $e_i$ are classified in a hierarchical taxonomy and are associated with a prime number $p_i$. When a particular EventType $e_j$ is added as a child of $e_i$, we store an unique identifier $uid_j$, computed as the product of $p_j$ and $uid_i$. As such, $uid_j$ uniquely encodes the position of $e_j$ in the hierarchy in relation to its various ancestors. The complete hierarchy is stored at the back-end and only small subset of this taxonomy needs to be stored on a node; corresponding with the events that local components produce or consume.

At runtime, based on this $uid_j$, we can efficiently compute whether an event of type $e_j$ and corresponding $uid_j$ is a subtype of $e_i$ by dividing $uid_j$ by $p_i$. If the result of this operation has no remainder, then due to the unique properties of prime numbers, $e_j$ must be a subtype of $e_i$. We apply some optimizations [20] that only require the $uid$ or $p_i$ to be transmitted during component deployment and binding time.

Finally, to address application evolution over time, components in CaPI can be dynamically deployed, started, stopped, removed, or rewired. A per-node *reconfiguration engine* maintains references to all installed components and enacts incoming deployment control, introspection and rewiring commands.

## B. CaPI policies

Policies in CaPI are specified using a policy language featuring an (Event)-Condition-Action (ECA) model. While the ECA-paradigm provides a good fit with the event-based nature of CaPI applications, we also allow for more general-purpose Condition-Action policies to be specified. Using this language, rich policies can be specified for a number of behavioural concerns, allowing the definition and use of both global and local variables, as well the use of a rich set of actions and language constructs. This section describes the general structure of policies and we refer to Section V for concrete policy examples.

Every CaPI policy is identified via a name, informal description, and a semantic identifier. The semantic identifier is based on a similar mechanism as the one used with events. This not only allows for compact classification and typing of the functionality the policy specifies, it also allows for reification and discovery at runtime. All policies may include the definition and use of local and global variables, allowing for more expressive and clear specifications. Local variables are contained within the local execution context of the policy, whereas global variables correspond to system-wide shared state, maintained by the *state repository* component. Currently, we support the use and manipulation of different data types such as boolean and string data types, 8-bit, 16-bit, 32-bit integers, 32-bit floating point numbers and collections thereof.

CaPI policies may define a number of optional operations that can be executed upon policy installation, activation, deactivation, removal, evaluation, and enforcement. This gives the policy developer additional freedom to execute a number of actions at well-defined moments. For instance, this permits one-time initialization of certain components upon policy installation, such as the preparation of security (e.g. construction of initialization vectors) or storage components (e.g. file system initialization)

Policy *conditions* allow for the definition of arbitrary logical expressions containing a valid combination of values, variables, built-in functions, logical and mathematical operators. Variables can either be local or global, variables corresponding with values contained in the payload of the event currently triggering the policy, or contextual properties associated with the intercepted event. These properties include event origin and destination *(ComponentID, NodeID)* and the location where the event has been intercepted inside the CaPI middleware.

The *action* part of a policy may consist of an arbitrary number of actions. CaPI provides a number of standard actions, such as admittance or blocking of the intercepted event, publishing of new events, storage of data, and the execution of custom functions. This set of standard build-in functions includes operations for dynamic *component management*, such as activating, deactivating, reparametrisation, or rewiring of components, and operations for dynamic *policy management*, such as enabling, disabling, or recomposing of policies. Moreover, in order to maximally support system evolution over time, the set of actions is dynamically extensible: new libraries containing new actions can be deployed at runtime.

Finally, since ECA-policies in CaPI act on semantically typed events, they can easily be applied to generic events. For instance, we can write a single policy to persist all events of type $e_i$, instead of having to write separate policies for every subtype $e_j$.

## C. Run-time adaptable policy infrastructure

*1) Configurable enforcement points:* CaPI policies are enforced at a number of well-defined locations within the middleware. Since all interactions between distributed applications are reified through events, the distributed event bus offers a single point of interception, allowing CaPI policies to govern all behaviour occurring at run-time. The event manager defines four entry and exit points ($A$, $B$, $C$, and $D$), corresponding with the publishing and reception of local and remote events. Each time an event traverses one of these points, it is redirected to the policy engine together with contextual information regarding that redirection. This context includes information about the exact point in the event manager where the event is intercepted, its origin component and destination, i.e. either a local component or destination inside the network. The policy engine uses this information to lookup which policies apply to that enforcement point and to check whether they are triggered by the said event. When a policy is added to an enforcement point by the WSN administrator, a priority needs to be assigned. Every enforcement point defines a list corresponding with the active policies and their individual priorities, sorted from high to low priorities, as such defining their order of execution. Obviously, conflicts may arise at runtime and CaPI currently only executes the policy with the highest priority. However, we are working on a formal analysis of the CaPI policy language to identify potential conflicts before deployment and to help the administrator in assigning these priorities [21].

*2) Adaptable policy engine:* The CaPI policy engine implements a small Virtual Machine (VM) featuring a stack-based execution model. Upon event interception in one of

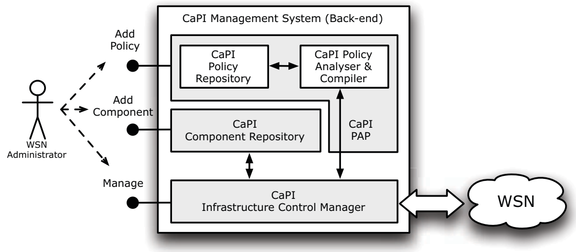

CaPI Management System (Back-end)
Add Policy
Add Component
Manage
WSN Administrator
CaPI Policy Repository
CaPI Policy Analyser & Compiler
CaPI Component Repository
CaPI PAP
CaPI Infrastructure Control Manager
WSN

Fig. 2. CaPI's PAP and management model

```
 1 deployPolicy <pFile> <nodeID>
 2 deployComponent <cFile> <nodeID>
 3 removePolicy <pID> <nodeID>
 4 enablePolicy <pID> <ePoint> <prio> <nodeID>
 5 disablePolicy <pID> <ePoint> <nodeID>
 6 changePriority <pID> <ePoint> <new prio> <nodeID>
 7 getPolicies <nodeID>
 8 getPolicyType <pID> <nodeID>
 9 getPoliciesByType <pType> <nodeID>
10 getEnforcementPointsForPolicy <pID> <nodeID>
11 getPoliciesOnEnforcementPoint <ePoint> <nodeID>
12 getPriority <pID> <ePoint> <nodeID>
13 schedule <pID> <frequency> <nodeID>
14 getSchedule <pID> <nodeID>
```

Listing 1. CaPI's management API (subset)

the enforcement points, the event is presented to the VM which then evaluates and executes the policies triggered. To allow for their execution, CaPI policies are represented as highly compact byte code instructions, the format of which is inspired by the Java VM specification. The adopted stack-based execution model makes it easy to exactly determine in advance how much memory a policy will actually require at run-time, a crucial concern for resource-constrained devices.

The CaPI architecture is highly adaptable as it not only offers dynamic installation and removal of policies, it also allows for the policy engine to be extended with new functionality at run-time. New software components, implementing new operations or actions can be added dynamically, enabling CaPI to support continuous system evolution. The CaPI policy engine keeps track of all actions currently offered by installed components. Every action is mapped to a corresponding *execute*-instruction with as operands the identifiers of the component and the specific operation. The engine also ensures that policies using actions not present any more are automatically disabled. Finally, the policy engine allows for periodic evaluation of Condition-Action policies by keeping track of them and maintaining timers to schedule their evaluation.

*3) Policy repository:* Upon policy installation on a node, the policy repository component calculates and allocates the necessary memory and stores the policy along with some metadata. This metadata includes policy status (enabled, disabled), the set of enforcement points where it has been applied to.

### *D. Component and policy management*

Figure 2 gives an overview of CaPI's Policy Administration Point (PAP) and component repository. Both the PAP and component repository reside on a back-end machine (or gateway device), acting as central management system towards the WSN. When a new policy is submitted to the PAP, the CaPI-toolchain, consisting of a policy analyser [21] and compiler, transforms its high-level (E)CA-specification into the corresponding byte code representation. The task of the analyser is twofold: (i) it checks for syntactical and semantical correctness of the policy while performing some simple type-checking, and (ii) it searches for potential conflicts prior to policy deployment and provides feedback regarding the assignment of priorities. Finally, the compiler generates the corresponding compact byte code representation. The *CaPI infrastructure control manager* interacts with the per-node *policy manager* and maintains a view of all installed components and policies inside the entire network, including how they are composed together. The API for policy and component management includes a number of operations for deployment and reconfiguration of components and policies. Listing 1 illustrates a subset of these operations, namely to deploy new policies and update the policy engine with add-on components (lines 1-2), remove, enable and disable policies on enforcement points or change their priorities (lines 3-6), inspect the set of policies on a node (line 7-12), and schedule periodic evaluation of condition-action policies (line 13-14).

## V. Examples

This section presents some examples of how CaPI policies can be used to exercise policy-driven control over various objectives: (i) non-functional concerns by domain experts, (ii) customization of functional behaviour by application experts, and (iii) automatic management and (re)configuration of the WSN platform by administrators.

### *A. Policy-based security of sensor data*

Listing 2 illustrates a policy enforcing confidentiality on sensor data via the XTEA-cypher. This policy is enforced on the enforcement point interfacing with the network stack (i.e. $C$ in Figure 1) . Upon policy enabling, we first initialize the security components allowing for faster encryption, while maintaining a unique identifier for later use (line 5). When an event of type SENSOR_READING is sent over the network, we enforce encryption on its contents (line 11). Note that this single policy also applies to all events of type temperature, humidity, or light readings, assuming they are subtypes of SENSOR_READING.

```
 1 policy "EC-SensorData" "Encrypt readings" "SEC_ENCR_XTEA"{
 2  include security as sec; //security library
 3  uint8_t id = 0; //local variable
 4
 5  on ENABLE { id <- sec::initialise("xtea",wsnkey.key); }
 6  on DISABLE { sec::clear(id); }
 7
 8  on event SENSOR_READING as s;
 9  condition ( true )
10  action (
11     sec::encrypt(id,s);
12  )
13 }
```

Listing 2. Confidentiality of sensor readings sent over the network

## B. *Policy-based customization of functionality*

To address the unique requirements of every application, we need support for advanced tailoring and customization of individual components. Instead of embedding all variability and customization logic directly inside an application component, hence compromising on reuse, policies can be specified that externalize this customization logic. Listing 3 illustrates a policy to calculate a simple moving average of temperature data (lines 10-17), followed by applying a custom filter to this value (line 19). When this value exceeds predefined thresholds, an alarm is raised and recorded (lines 21-24). In any case, this policy can be easily changed over time. For instance, when the simple moving average is not sufficient anymore we can easily switch to a weighted moving average policy.

```
 1 policy "TempAlert-SMA" "Alerting based on SMA"
 2                                         "AVERAGING" {
 3   float[] readings = new float[10];
 4   uint8_t oldestObservation = 0;
 5   uint8_t total = 0;
 6   float smavg = 0;
 7
 8   on event TEMP_CELSIUS as t;
 9
10   readings[oldestObservation] <- t.value;
11   if (total < 10) { total <- total + 1;}
12   oldestObservation <- (oldestObservation+1) % 10;
13   smavg <- 0;
14   for i in readings[0:total] do {
15      smavg <- smavg + readings[i];
16   }
17   smavg <- smavg/total;
18
19   condition (smavg < 4 || smavg > 7 )
20   action (
21      persist("temp.log",append,[t.src_addr,
22                               t.timestamp, smavg]);
23      publish(TEMP_ALERT,[t.src_addr, t.timestamp,
24                               smavg]);
25   )
26 }
```

Listing 3. Calculating simple moving average as a CaPI policy

## C. *Policy-based resource management*

Finally, policies can be used to perform automatic on-node reconfiguration, monitoring and management of system resources. Based on our previous experiences with solar-powered WSN motes in a scenario of flood monitoring in England [18], the WSN administrator could write a Condition-Action policy, illustrated in Listing 4, and schedule it for periodic evaluation. The policy defines that when battery levels drop at night, the sampling frequency of the (power-hungry) water pressure monitoring sensor, controlled by a low-level 'Pressure'-component, will be decreased automatically given there is no imminent danger of flooding (lines 9-15). Of course other policies may define compensating actions to take during the day when batteries become recharged.

## VI. Evaluation

CaPI has been implemented on top of the Contiki 2.5 [22] WSN operating system using AVR-Raven sensor nodes. These nodes consist of an ATmega1284p 20MHz 8-bit microcontroller, 128 kB program memory (ROM) and 16 kB RAM.

```
 1 policy "AdaptTempSampling" "Adapt sampling based on
 2                        available energy" "ENERGY_MGT" {
 3
 4   global uint8_t battery; //shared variables
 5   global boolean high_flood;
 6
 7   condition (battery < 30)
 8   action (
 9     uint8_t[] pIDs <- getComponentIDsType("Pressure");
10     for pID in pIDs[0:pIDs.len] do {
11       uint8_t current_srate <- getProperty(pid,"s_rate");
12       if(current_srate < 10 && !high_flood) {
13         setProperty(pid,"s_rate", 25); //increase interval
14       }
15     }
16   )
17 }
```

Listing 4. Resource management policy, subject to periodic evaluation

Based on positive experiences with IPv6 regarding interoperability and reliability in real-world WSN scenarios [23], we used the Contiki IPv6 (6LoWPAN) networking stack for communication and reliable over-the-air reconfiguration. For dynamic component deployment, we use the standard Contiki ELF loader. Thus, both components and policies are reliably disseminated using IPv6. Although the use of IPv6 introduces an overhead in terms of code size (ROM and RAM), it adds significant flexibility and eases the overall manageability of the network.

### A. *Memory footprint*

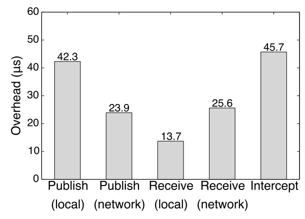
Table I illustrates the memory footprint of CaPI. We measured RAM and ROM requirements and compared the overhead with the memory requirements of Contiki. Although CaPI introduces some additional memory requirements on top of Contiki, the added value is significant in terms of flexibility. Moreover, the combined version leaves approximately 50% of ROM and 30% of RAM on the AVR-Raven sensor node available for application components and policies.

### B. *Run-time performance*

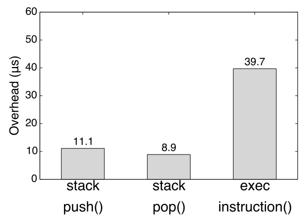
We measured the overhead the event manager incurs for (i) event handling and (ii) event interception plus redirection to the policy engine. As illustrated in Figure 3a, the runtime performance of CaPI is reasonably fast. Next, we measured memory requirements and runtime performance of the policy engine. The policy engine (i.e. CaPI VM) only requires 5546 bytes of ROM and 76 bytes of RAM. Since the overhead of

|  | Contiki | CaPI |  |  | Contiki + CaPI |
|---|---|---|---|---|---|
|  |  | Component runtime | Policy runtime | Total |  |
| ROM (bytes) | 40262 | 14600 | 10540 | 25140 | 65402 |
| RAM (bytes) | 8484 | 2291 | 624 | 2915 | 11339 |
| ROM overhead | - | 36.3% | 26.2% | 62.4% | - |
| RAM overhead | - | 27.0% | 7.4% | 34.4% | - |
| % of total ROM | 30.7% | 11.1% | 8.0% | 19.2% | 49.9% |
| % of total RAM | 51.8% | 14.0% | 3.8% | 17.8% | 69.6% |

TABLE I
Memory footprint of CaPI



(a) Event manager performance



(b) CaPI VM performance

Fig. 3. Run-time performance overhead

| Objective | Policy DC | Component DC | Component memory footprint (ROM/RAM) |
|---|---|---|---|
| EC-SensorData | 32 bytes | 1692 bytes | 283 bytes / 71 bytes |
| SMA-TempAlert | 126 bytes | 1864 bytes | 274 bytes / 101 bytes |
| AdaptSampling | 72 bytes | 1642 bytes | 260 bytes / 61 bytes |

TABLE II
DEPLOYMENT COST: AMOUNT OF BYTES TO BE SENT OVER THE AIR

policy interpretation (i.e. evaluating the condition and execution of actions) is highly policy specific, we first measured the performance of its basic operations: fetching a new instruction, decoding it, and performing the necessary stack operations. As shown in Figure 3b, the overhead is very minimal. Next, we measured the time it takes to evaluate a number of policies. The overhead is acceptable, as on average it takes between 350 $\mu$s and 650 $\mu$s to evaluate and execute a policy.

### C. Dynamic deployment costs

Dynamic reconfiguration of application components and policies comes at a certain cost. We measured the amount of bytes needing to be transmitted over the air. Table II illustrates the Deployment Costs (DC) of dynamic policy installation. To put these numbers in perspective, we compared them with their functionally equivalent component-based counterparts. Accordingly, we also evaluated the deployment costs and memory requirements of these components.

As can be seen from Table II, if we were to implement the same functionality provided by policies through a component-based abstraction, we would incur a much higher deployment cost. However, we attribute a lot of this overhead caused by the non-optimal ELF representation used to encapsulate our components. Note that approximately only 20% of the ELF-file contains relevant component data. Nonetheless, policies are always significantly smaller than their component-based counterparts. This is an advantage since the behaviour of a system (governed by policies) in general changes more frequently than its structure (components) during its lifespan.

## VII. RELATED WORK

Policy-based management has already been a well-studied area of research by the community in the context of ad-hoc networks and regular distributed systems [13], [14]. Nonetheless, WSNs exhibit significantly different characteristics in terms of resource-constraints, scale, dynamism and use cases. Therefore, we focus on related work in the area of policy systems for WSNs. Since policy-based WSN middleware is a relatively new area of research, only a few policy-systems for WSNs exist to date. The systems that are most related to CaPI are Finger [16] and ESCAPE [17].

Finger [16] is a WSN policy system developed at Imperial College and completely implemented on top of the TinyOS [1] operating system. Finger operates on a reduced subset of the Ponder2 policy specification language. Similar to CaPI, policies in Finger can be dynamically installed, removed, activated and deactivated. However, as Finger is TinyOS-based, any reconfiguration options other than complete image replacement are not supported. Finger's static approach to reconfiguration has its disadvantages; since the functional software base always remains static, policies in Finger always remain constrained to the set of base functionality present on a node. In contrast, CaPI allows both components and policies to evolve dynamically and independently from each other, providing full flexibility for system evolution.

ESCAPE [17] is a policy-based WSN middleware implemented on TinyOS [1]. Applications in ESCAPE are implemented using nesC-components [2] and adopt a similar pub/sub interaction model as in CaPI. However, ESCAPE differs from CaPI in a significant number of elements. First, the set of components and policies on a node in ESCAPE is not dynamically adaptable. Policies and application components cannot be dynamically installed, removed, nor recomposed. At compile time, ESCAPE policies are statically woven together with application code. As a result, policies can only be activated and deactivated at runtime. Second, ESCAPE policies are used to specify all application behaviour, including the definition of distributed interactions. For instance, policies are used to schedule timers to sample data, handle time-outs, filter resulting measurements, and publish events over the network.

Application-specific virtual machines (ASVMs) [24] aim at implementing complete WSN applications as lightweight scripts that are interpreted at runtime using a tiny application-specific virtual machine. Applications are written using a byte code language, and interact with installed components. Similar to CaPI, one of the main arguments is that the cost of interpretation is acceptable given the lower cost of software updates. However compared to CaPI, the current generation of ASVMs only focus on implementing complete applications as scripts, hence disregarding stakeholder and concern heterogeneity.

Aspect-oriented programming (AOP) [25] is the approach put forward by the software engineering community to address the principle of separation of concerns. AOP focusses on modularization of external concerns via dedicated software constructs, called aspects. These aspects allow for the specification of concrete behaviour to be invoked at well-defined locations inside application programs. Aspects are combined with application code through 'weaving'. The purely development-centric approach adopted in AOP is, however, too narrow to be used in the context of WSN applications. First, it requires the component developer to be aware of all concerns at development time, including all (current and future) behavioural concerns (including concerns from other stakeholders). Secondly, the weaving process directly embeds aspects inside application logic. This might possibly compromise component reusability when components are to be reused in future compositions, or it might even introduce conflicts at runtime which are hard to resolve [26]. Third, runtime weaving is infeasible to execute on resource-constrained devices as it is a very resource-intensive operation requiring dedicated platform support.

## VIII. Conclusion

We presented the design and implementation of CaPI, a dynamically reconfigurable component and policy-based middleware for WSNs. The CaPI middleware offers components and policies as first-class programming abstractions to realize heterogeneous concerns in a very efficient fashion. The multi-paradigm approach adopted in CaPI addresses the need for adequate abstractions to realize and manage functional and behavioural concerns inside WSNs. Components implement functional application concerns in a coarse-grained fashion, whereas policies provide a more natural fit to provision behavioural concerns. In contrast to related state-of-the-art approaches, CaPI offers full support for reconfiguration and long-term system evolution, an essential requirement for long-lived successful WSN deployments. Furthermore, due to the existence of both a fine-grained and coarse-grained paradigm, runtime adaptations can be enacted very efficiently. Future work will focus on three areas of research: formal analysis of the CaPI policy language and middleware to further enhance the development and management of WSN applications, secure management of CaPI-based WSNs, and evaluation of the CaPI-middleware in a number of real-world deployments.

## Acknowledgments

This research is partially funded by the Interuniversity Attraction Poles Programme of the Belgian State of the Belgian Science Policy, the Flemish Agency for Innovation by Science and Technology (IWT Vlaanderen), and the KU Leuven Research Fund. Part of this work has been funded by the IWT-SBO-SymbioNets project No. 090062.

## References

[1] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "TinyOS: An operating system for sensor networks ambient intelligence," in *Ambient Intelligence*, W. Weber, J. M. Rabaey, and E. Aarts, Eds. Berlin/Heidelberg: Springer Berlin Heidelberg, 2005, ch. 7, pp. 115–148.

[2] D. Gay, P. Levis, R. V. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *Proc. Programming Language Design and Implementation (PLDI) 2003*. New York, USA: ACM Press, May 2003, pp. 1–11.

[3] P. Dutta, J. Hui, J. Jeong, S. Kim, C. Sharp, J. Taneja, G. Tolle, K. Whitehouse, and D. Culler, "Trio: enabling sustainable and scalable outdoor wireless sensor network deployments," in *Proceedings of the 5th international conference on Information processing in sensor networks*. New York, NY, USA: ACM, 2006, pp. 407–415.

[4] Very Large-Scale Open Wireless Sensor Network Testbed, Lyon, France, 2010, http://www.senslab.info/.

[5] H. Schaffers, N. Komninos, M. Pallot, B. Trousse, M. Nilsson, and A. Oliveira, "The future internet," J. Domingue, A. Galis, A. Gavras, T. Zahariadis, and D. Lambert, Eds. Berlin, Heidelberg: Springer-Verlag, 2011, ch. Smart cities and the future internet: towards cooperation frameworks for open innovation, pp. 431–446.

[6] P. Grace, D. Hughes, B. Porter, G. S. Blair, G. Coulson, and F. Taiani, "Experiences with open overlays: a middleware approach to network heterogeneity," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. New York, NY, USA: ACM, 2008, pp. 123–136.

[7] P. Costa, G. Coulson, C. Mascolo, L. Mottola, G. P. Picco, and S. Zachariadis, "Reconfigurable component-based middleware for networked embedded systems," *International Journal of Wireless Information Networks*, vol. 14, no. 2, pp. 149–162, 2007.

[8] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler, "An analysis of a large scale habitat monitoring application," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*. New York, NY, USA: ACM, 2004, pp. 214–226.

[9] V. Dyo, S. A. Ellwood, D. W. Macdonald, A. Markham, C. Mascolo, B. Pásztor, S. Scellato, N. Trigoni, R. Wohlers, and K. Yousef, "Evolution and sustainability of a wildlife monitoring sensor network," in *SenSys*, 2010, pp. 127–140.

[10] C.-L. Fok, G.-C. Roman, and C. Lu, "Enhanced coordination in sensor networks through flexible service provisioning," in *Proceedings of the 11th International Conference on Coordination Models and Languages*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 66–85.

[11] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, "A generic component model for building systems software," *ACM Trans. Comput. Syst.*, vol. 26, pp. 1:1–1:42, March 2008.

[12] B. Porter, U. Roedig, and G. Coulson, "Type-safe updating for modular wsn software," in *International conference on Distributed Computing in Sensor Systems*, june 2011, pp. 1 –8.

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

[14] W. Han and C. Lei, "Survey paper: A survey on policy languages in network and security management," *Comput. Netw.*, vol. 56, no. 1, pp. 477–489, Jan. 2012.

[15] P. Anderson, "Short topics in system administration 14: System configuration," Published by the Usenix Association, Berkeley, CA, 2006.

[16] Y. Zhu, S. L. Keoh, M. Sloman, and E. C. Lupu, "A lightweight policy system for body sensor networks," *IEEE Transactions on Network and Service Management*, vol. 6, no. 3, pp. 137–148, 2009.

[17] G. Russello, L. Mostarda, and N. Dulay, "A policy-based publish/-subscribe middleware for sense-and-react applications," *J. Syst. Softw.*, vol. 84, pp. 638–654, April 2011.

[18] D. Hughes, P. Greenwood, G. Blair, G. Coulson, P. Grace, F. Pappenberger, P. Smith, and K. Beven, "An experiment with reflective middleware to support grid-based flood monitoring," *Concurr. Comput. : Pract. Exper.*, vol. 20, pp. 1303–1316, August 2008.

[19] D. Hughes, K. Thoelen, W. Horré, N. Matthys, J. D. Cid, S. Michiels, C. Huygens, and W. Joosen, "Looci: a loosely-coupled component infrastructure for networked embedded systems," in *Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia*. New York, NY, USA: ACM, 2009, pp. 195–203.

[20] K. Thoelen, N. Matthys, W. Horré, C. Huygens, W. Joosen, D. Hughes, L. Fang, and S.-U. Guan, "Supporting reconfiguration and re-use through self-describing component interfaces," in *Proc. of the 5th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*. New York, NY, USA: ACM, 2010, pp. 29–34.

[21] M. Patrignani, N. Matthys, J. Proença, D. Hughes, and D. Clarke, "Formal analysis of policies in wireless sensor network applications," in *Proceedings on the 3rd International Workshop on Software Engineering for Sensor Network Applications (SESENA)*, 2012, to appear.

[22] 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*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 455–462.

[23] J. W. Hui and D. E. Culler, "Ip is dead, long live ip for wireless sensor networks," in *Proceedings of the 6th ACM conference on Embedded network sensor systems*. New York, NY, USA: ACM, 2008, pp. 15–28.

[24] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 85–95, October 2002.

[25] G. Kiczales and E. Hilsdale, "Aspect-oriented programming," *SIGSOFT Softw. Eng. Notes*, vol. 26, pp. 313–, September 2001.

[26] R. Douence, P. Fradet, and M. Südholt, "Composition, reuse and interaction analysis of stateful aspects," in *Proceedings of the 3rd international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2004, pp. 141–150.