Flexible Integration of Data Qualities in Wireless Sensor Networks

Nelson Matthys, Christophe Huygens, Danny Hughes, Sam Michiels, and Wouter Joosen
IBBT-DistriNet, Katholieke Universiteit Leuven,
B-3001, Leuven, Belgium
{firstname.lastname}@cs.kuleuven.be

ABSTRACT
Wireless sensor networks have been confirmed as a valuable asset to deploy in an increasing number of real-world business scenarios. The existence of several run-time reconfigurable component models for networked embedded systems facilitates this as they enable developers to build complete end-to-end systems composed of reusable building blocks. However, the integration of sensor networks with enterprise environments is not straightforward as it requires that several long-standing challenges related to data quality management and security be addressed. This paper advocates a high-level policy-based approach to more easily support the specification and enforcement of these non-functional requirements in existing component compositions for network embedded systems. We have realized and evaluated a prototype of this framework for the LooCI component model.

Categories and Subject Descriptors
C.2.4 [Computer-Communication Networks]: Distributed Systems, Distributed Applications

General Terms
Design, Management

Keywords
component models, policy-driven integration, data quality

1. INTRODUCTION
Over the last few years, Wireless Sensor Networks (WSNs) have transitioned from a single-purpose application paradigm such as habitat monitoring [11] or disaster management [3] towards potentially serving a multitude of business domains like logistics [8] or industrial process management [1, 14]. The platforms enabling this integration of WSNs into the enterprise environment follow the vision of not being merely data-centric but rather be constructed of set of reusable building blocks organized in a service-oriented fashion [14, 15]. Within the enterprise, the WSN is therefore becoming another tier of enterprise information technology that provides a general-purpose platform, albeit with limited execution capability.

In these business scenarios, complete applications are made up as a composition of different services running on heterogeneous sensor nodes, gateway infrastructure, and powerful enterprise back-end systems. The desired functionality can thus easily be enabled by deploying the necessary services on this infrastructure and by wiring them together to fulfill application requirements.

To this end, a number of run-time reconfigurable component models have already been developed for embedded systems, most notably OpenCOM [4], RUNES [3], and OSGi [12]. These component models are all suited to support the functional requirements of end-users by offering the ability to develop reusable components as key building blocks. Concrete advantages of these component models are:

- Concrete interfaces to uniquely specify component functionality and promote reuse of components between applications.
- On demand component deployment can be used to manage dynamism and evolution through the injection of new functionality.
- Component rewiring can be used to modify component compositions on the fly and thus offers a mechanism to manage dynamism and evolution. The ability to dynamically wire a third party component into a composition also promotes reuse.

However, integration of enterprise infrastructure with the specific nature of the WSN requires that some long-standing challenges, such as data quality management and security, be addressed in new ways. Novel ways of supporting these non-functional concerns must be provided, taking into account the resource-constrained and dynamic nature of the WSN. In addition, as these concerns are typically handled by domain experts unfamiliar with the different flavors of WSN technology, a comprehensible way of specifying and integrating them in the WSN tier should be provided.

This paper advocates a high-level policy-based approach for specifying various non-functional concerns, such as security or the desired quality of particular data. We provide a lightweight framework to integrate and enforce various non-functional concerns on an underlying run-time reconfigurable component model.
The remainder of this paper is structured as follows: Section 2 motivates our approach by indicating the importance of non-functional concerns, and showing how they can be easily integrated by using a policy-based approach. Section 3 briefly zooms in on the LooCI component model used to realize our approach. Section 4 describes our policy language and its corresponding framework, whereas Section 5 evaluates it by using a case study. Section 6 gives an overview of related work. Finally, Section 7 concludes this paper and sketches future work.

2. MOTIVATION

Firstly, the role of non-functional concerns regarding functional compositions in WSNs is explained. Secondly, we discuss how these concerns can be integrated in the WSN by using a policy-driven approach.

2.1 The role of non-functional concerns

Today’s business requirements drive the composition of applications in terms of reusable and flexible building blocks, exemplified by the use of several component models as argued in Section 1. Similarly, non-functional concerns also dictate a certain amount of modularization in software architecture, whereas this modular approach is needed throughout the entire application life cycle. On one hand, at development time, concerns exist that require a division of the problem. For instance, code maintenance mandates a non-monolithic approach to design, whereas concerns related to auditability, result in highly concentrated and security-specific code fragments. On the other hand, at runtime, data quality concerns like precision or reliability may exist that translate into specific sampling frequencies, resource selection strategies, or alternative replication tactics.

The existence of these non-functional concerns, each having explicit goals, logic and code, that might often be contradictory or at least remote from the application, is a key driver for system structure. This is even more important for WSNs where stringent boundary conditions such as resource scarcity or weak connectivity exist that limit the set of possible tactics for system organization. The explicit distinction between functional and non-functional goals is therefore important. Both universes, for example, adhere to different timelines: in a node localization scenario where nodes need to temporarily limit their transmission to conserve energy during the night, there is no reason to change the overall functional composition to deal with this problem. At the non-functional level, this problem could easily be solved by changing the default transmission policy into a temporary hourly store-and-forward strategy during the night, which is orthogonal to the composition.

Not only timelines but also the goals differ. For instance, the administrative owner of a WSN wants to optimize data quality for all users, whilst an individual functional composition is selfish and will only optimize locally for its particular data. This is similar to the runtime requirement of the non-functional quality of security: “segregation of duties” that requires the application and security logic to be independently operated. On the other hand, whilst the objectives of the functional and non-functional requirements differ, the mechanisms used to address these non-functional requirements can be quite similar over all the applications subject to the specific non-functional concern. The quality requirements of a health-care patient tracking system are not all that different of an environmental spill monitor: one wants the data to arrive with sufficient level of accuracy and integrity, and thus one can employ the same approaches to achieve this regardless of the application.

The rules governing the non-functional universe are therefore (i) distinct from the application, (ii) generic for each single concern and generally considered to be (iii) more dynamic and changeable when compared to the functional universe. Many solutions have been suggested to support non-functional concerns focussing either on (i) modularization, (ii) reuse, or (iii) the dynamic nature in the context of distributed systems development and, as discussed in Section 6, some have been transposed to the WSN domain.

2.2 Policy-driven integration of non-functional concerns

Since the non-functional rules are generally considered to be short-lived and subject to frequent changes, it is hard to fully capture all the logic and scenarios thereof at development time. A more flexible system consisting of a programmable execution engine with run-time interpretation and enforcement of the dynamic, high-level non-functional objectives provides a solution to this problem. These systems refer to the dynamic non-functional ruleset as “policy”. Over the last decade, research on policy-based management [2] has been applied to facilitate management tasks, such as component configuration, security, or Quality of Service in large-scale distributed systems. Allowing the specification of requirements about the intended behavior of a managed system using a high-level policy language, which is automatically enforced in the system, enables domain experts to effectively achieve results. Furthermore, policies can be changed dynamically without having to modify the underlying programmable logic (i.e. the implementation of the engine) or requiring the consent or cooperation of the components being governed.

The approach proposed in this paper is to combine the key benefits of a run-time reconfigurable component model (i.e. the ability to dynamically inject new functionality and reason about distributed relationships between components), with the efficiency of policy-based integration of non-functional concerns. So, the presented modularization approach is different for both classes of concerns, since we believe non-functional policies need a more dynamic solution in WSNs. As we will show in Section 5, this reduces the burden on developers, has a very low memory footprint, and comes with a very limited performance overhead. Furthermore, the policy language we propose is high-level and easy to understand, allowing end-users, as well as domain experts, to customize the non-functional behaviour of component compositions.

3. LOOCCI: LOOSELY-COUPLED COMPONENT INFRASTRUCTURE

The Loosely-coupled Component Infrastructure (LooCI) [6] is designed to support Java ME CLDC 1.1 platforms such as the Sun SPOT [19]. LooCI is comprised of a component model, a simple yet extensible networking framework and a common event bus abstraction. The component model supports run-time reconfiguration, interface definitions, introspection, and wiring of bindings. LooCI provides two component types, macrocomponents and microcomponents.
Macrocomponents are coarse-grained and service-like, and build upon the notion of Isolates inherent in embedded Java Virtual Machines such as Sentilla [1] and SQUAWK [20]. Isolates are process-like units of encapsulation and provide varying levels of control over their execution (exactly what is provided is dependant on the specific JVM). LooCI standardizes and extends the functionality offered by Isolates. Each macrocomponent runs in a separate Isolate and communicates with the run-time middleware via Inter Isolate RPC (IIRPC), which is offered by the underlying system. Unlike microcomponents, macrocomponents may use multiple threads and utility libraries.

Microcomponents are fine-grained and execute in the master Isolate alongside the LooCI runtime. Unlike macrocomponents, microcomponents must be single threaded and self contained, using no utility libraries. Aside from these restrictions, microcomponents offer identical functionality to macrocomponents in a smaller memory footprint. LooCI components are indirectly bound over a lightweight event bus and define their provided interfaces as part of the set of LooCI events that they publish. Similarly, the receptacles of a LooCI component are defined as the events to which they subscribe. As bindings are indirect, they may be modified in a manner that is transparent to the composition. Furthermore, as all events are part of a globally specified event hierarchy, it becomes easier to understand and interpret data flows.

4. POLICY FRAMEWORK

4.1 Policy language

The specification of policies is accomplished by using policy rules following Event-Condition-Action (ECA) semantics, which correspond well to the event-driven nature of the target WSN platforms. An ECA policy consists of a description of the triggering events, an optional condition which is a logical expression typically referring to external system aspects, and a list of actions to be enforced in response. In addition, our prototype policy language allows various functions to be called inside the condition and action parts of a policy. Listing 1 gives an overview of our policy language specification in Backus-Naur Form (BNF), whereas we refer to Section 5 for concrete examples about the policy language usage. By using these policies, we offer a simple, yet powerful method to integrate various non-functional concerns in functional component compositions.

**Listing 1: Policy Language Specification in BNF**

Furthermore, we provide tool support that allows end-users to firstly select the individual components and inter-
the set of policies at runtime, the framework can be adapted according to evolving application demands.

5. CASE STUDY EVALUATION

This section illustrates and evaluates the proposed policy-based approach to integrate non-functional concerns in component compositions by means of a case study in a logistics scenario [8]. A logistics company 'LOGISTICS_CO' ships different categories of products in crates equipped with sensor nodes: (a) expensive pharmaceutical products produced by 'PHARMA_CO', and (b) low-cost furniture supplies sold by 'STORE_CO'. Consider the following two cases that have the need for integrating several non-functional concerns:

1. Government legislation obliges PHARMA_CO to be able to reconstruct the full trace of their products during transport. To fulfill this requirement, LOGISTICS_CO must store location information of each individual pharmaceutical crate to reconstruct individual movements of each crate. Service agreements between both companies also require this during transport when no direct connection with the enterprise back-end can be established. Furthermore, to protect this info against tampering and to assure that all information arrives in the trace log, all location information should be checked for integrity and delivered confidentially to the back-end of LOGISTICS_CO.

2. The service contract between STORE_CO and LOGISTICS_CO is best-effort and specifies a pay-per-use formula, i.e. only LOCATION events that are effectively received by STORE_CO are accounted. In addition, the best-effort scheme does not require location information to be secured at all.

To build such a system, LOGISTICS_CO uses a set of generic LooCI components as illustrated in Figure 2. To periodically determine the location of each crate, a generic LOCATION_COMP (e.g. a GPS unit or signal strength-based localization component) is installed on the sensor attached to the crate. This component produces events of type LOCATION, containing a timestamp and spatial coordinates of the location of the crate inside a warehouse or during road transport. The LOCATION_COMP is then wired to a LOCATION_AGGREGATION component deployed on the gateway in each warehouse or truck during road transport. When a connection exists between the gateway and back-end, LOCATION events are delivered directly to individual TRACKER_COMPs for PHARMA_CO and STORE_CO. To fulfill the service contract between PHARMA_CO and LOGISTICS_CO, and to cope with possible disconnections during transport, the following redundancy strategy needs to be applied: location data from pharmaceutical crates should be replicated using a LOCATION_STORAGE_COMP at the gateway. The service contract between LOGISTICS_CO and STORE_CO focuses on offering best-effort crate tracking and therefore specifies a pay-per-use type, mandating a BILLING_COMP in the back-end of LOGISTICS_CO used for accounting purposes.

Section 5.1 first shows how confidentiality and integrity qualities on LOCATION data destined for PHARMA_CO can be flexibly integrated using our policy-based approach. Section 5.2 then illustrates how redundancy of LOCATION information destined for PHARMA_CO can be supported. Finally, Section 5.3 shows how accounting on a per-use-base can be integrated. For each of the three scenarios, Figure 2 shows the exact location where the policy is integrated.

5.1 Scenario 1: Integrating Security

Listing 2 presents an example of a security policy that becomes integrated in our system. Whenever a LOCATION event is received on the event bus of a PHARMA_CO node, confidentiality and integrity is assured by encrypting the event’s payload using the shared key between the node and back-end. Furthermore, a Message Authentication Code (MAC) is added to the event ensuring its integrity. Finally, the event is allowed for further dissemination on the bus.

```java
Listing 2: Example security policy deployed on PHARMA_CO nodes

At the LOGISTICS_CO back-end, a similar policy which is described in Listing 3 becomes enabled. This policy is used to check the integrity of the LOCATION event payload and decrypt it. Once the MAC of the payload has been verified and decrypted, the event is allowed for further dissemination to the PHARMA_TRACKER_COMP.

```java
Listing 3: Example security policy deployed on LOGISTICS_CO back-end

```java
Listing 4: Example redundancy policy on gateway

In all cases, even if the gateway node on a truck becomes disconnected from the back-end system, LOCATION information will be stored. Whenever the connection is reestab-
Figure 2: Component composition for the LOGISTICS_CO case study. Policy integration points are numbered, marked in gray, and correspond with the three individual scenarios.

lished, the PHARMACOMP can retrieve the missing data and empty the gateway’s cache.

The policy engine intercepts all LOCATION events coming from pharmaceutical crates and locally stores their content (i.e. ID of origin node, spacial coordinates and timestamp both inside event payload). Note that this policy does not apply to LOCATION events coming from other nodes besides those attached to pharmaceutical crates. To check this, the policy condition uses a built-in `memberof(x,y)` function to determine whether the source address x of the event is contained in the list of pharmaceutical nodes.

5.3 Scenario 3: Integrating Accounting

To enable LOGISTICS_CO to apply billing of LOCATION data on a per-use-base the following policy (see Listing 5) is integrated at the back-end tier of the system. Each time a LOCATION event is intercepted, containing STORETRACKER_COMP as destination, an ACCOUNT_USAGE event, destined for the BILLING_COMP, is published to the event bus. This ACCOUNT_USAGE event contains information about the data inside the LOCATION event and its recipient STORE_CO.

```java
policy "accounting on per-use-base" "1" {
    on LOCATION as loc;
    if(loc.dest == STORE_TRACKER_COMP)
        then( // do accounting on a per-use-base
            allow loc; // and allow loc to continue
            publish ACCOUNT_USAGE("STORE_CO",
            loc.payload[1]);
        )
}
```

Listing 5: Example accounting policy deployed in back-end

5.4 Performance evaluation

We implemented the policy framework on Java ME CLDC 1.1 compliant SunSPOT nodes [19] (180 MHz ARM9 CPU, 512 kB RAM, SQUAWK VM ‘BLUE’ version). The resulting size of the framework is 26 kB. Representing a single non-functional concern from Section 5 as a policy is very efficient as it only consumes 376 bytes of memory on average, when compared to implementing the concern as a LooCI component, which requires 26 kB on average.

In terms of development overhead, implementing a concern using our policy-based approach only requires 9 Source Lines of Code (SLoC) on average, whereas implementing the same concern using standard LooCI components resulted in 40 SLoC on average (i.e. 78% less overhead per policy). Not only memory footprint, but also time to deploy (200 ms) and initialize (6 ms) a policy is significantly lower than deploying (11335 ms) and initializing (7420 ms) a similar LooCI component. Furthermore, the overhead of evaluating a policy and executing its actions is equal to the overhead of very efficient LooCI microcomponents (i.e. 4 ms).

6. RELATED WORK

Relevant related work focuses on software modularization through component models, reuse of generic concepts, and dynamic integration of non-functional concerns in component compositions through policy-based mechanisms.

NesC [5] is perhaps the best known component model for WSNs and is used to implement the TinyOS [10] operating system. NesC provides an event-driven programming approach together with a static component model. NesC components cannot be dynamically reconfigured, however, the static approach of NesC allows for whole-program analysis and optimization. OpenCOM [4] is a general purpose, run-time reconfigurable component model that has been deployed in a number of WSN scenarios. OpenCOM supports dynamic reconfiguration via a compact run-time kernel. The RUNES [3] run-time reconfigurable component model brings OpenCOM functionality to more resource-constrained embedded devices. Along with a smaller footprint, RUNES adds a number of introspection API calls to the OpenCOM kernel.

Aspect-oriented programming [9] focuses on modularity by allowing the developer to specify non-functional concerns that crosscut several components. An aspect can alter the behavior of the base component by applying additional behavior at various selected locations in a program.

Reuse is mainly supported by the use of patterns [17, 18] that have been identified to capture solutions in various non-functional fields such as security or reliability. They detail the time-tested combination of solution elements, such as infrastructure and software elements, with their function and operational model. For example, the clustering pattern in WSN may be used to provide reliability by assigning control to a selected cluster head and provide data quality through increased precision by cluster averaging. Applicable to many problems, patterns strongly increases reuse.
To date, several policy systems exist that address enforcement of non-functional concerns in component-models for embedded systems. ESCAPE [16] is a component-based policy framework for programming sensor network applications using TinyOS [10]. ESCAPE starts from the Separations of Concern principle, in which developers use policies to exclusively specify interactions between components, removing interaction code from these individual components. Our approach applies policy techniques to integrate various non-functional concerns on entire component compositions. In addition, ESCAPE is implemented on top of the static NesC component model [5], whereas our framework builds on top of a more flexible run-time reconfigurable component model.

Driven by industry initiatives, the Service Component Architecture (SCA) defined a Policy framework specification [13], which aims to use policies for describing capabilities and constraints that can be applied to service components or to the interactions between different service components. While not being bound to a specific implementation technology, the SCA policy framework focusses on service-oriented environments such as OSGi [12] which may only be applied to more powerful embedded devices.

7. CONCLUSION

This paper presented how various non-functional concerns can be flexibly integrated in functional component compositions using using a lightweight policy-based approach suitable for use in sensor networks. We detailed the characteristics of the rules governing the non-functional universe. The presented policy framework that addresses these non-functional needs builds on top of a run-time reconfigurable component model, offering an event bus abstraction as a common policy interception point.

In the short term, future work will focus on improving the expressiveness of our policy language, in order to be capable to express more advanced non-functional crosscutting concerns and resolution strategies. In the longer term, we hope to implement and evaluate the framework for a wider range of WSN application scenarios and platforms, using the OpenCOM [4] and OSGi [12] component models.

Acknowledgments

Research for this paper was partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, Research Fund K.U.Leuven, and is conducted in the context of the IBBT-DEUS project [7] and IWT-SBOSTADIUM project No. 80037 [8].

8. REFERENCES