Safe Reparametrization of Component-Based WSNs

Wilfried Daniels(✉), Pedro Javier del Cid Garcia, Wouter Joosen, and Danny Hughes

iMinds-DistriNet, KU Leuven, 3001 Heverlee, Belgium
{wilfried.daniels,pedrojavier.delcid,wouter.joosen, danny.hughes}@cs.kuleuven.be

Abstract. Modern Wireless Sensor Networks are moving from singe-purpose custom built solutions towards multi-purpose application hosting platforms. These platforms support multiple concurrent applications managed by multiple actors. Reconfigurable component-models are a viable solution for supporting these scenarios by reducing management and development overhead while promoting software reuse. However, implicit parameter dependencies spanning component compositions make reconfiguration complex and error-prone. This paper proposes composition-safe reparametrization of components. This is accomplished by offering language annotations that allow component developers to make dependencies explicit and network protocols to resolve and enforce parameter constraints. Our approach greatly simplifies reparametrization while imposing minimal runtime overhead.

1 Introduction

Early WSNs supported single purpose applications with little or no runtime reconfiguration support. Typically, a single party built, owned and maintained the WSN. With the advent of shared sensing infrastructures, e.g. Smart Cities [7] and Smart Offices [10], WSNs are evolving to become more open and multi-purpose [9,13]. A single WSN infrastructure can host multiple applications at the same time that are managed by multiple actors.

Reconfigurable component models have proven to be a promising solution for managing the complexity of developing WSN applications [9]. Examples of runtime reconfigurable component systems include OpenCOM [5], RUNES [4], OSGi [12], REMORA [14] and LooCI [8]. These systems provide the capabilities required to manage component life-cycle, configuration, introspection, and assembly at runtime. An essential feature of component-based WSN infrastructures is component reuse, where one component can offer functionality to multiple application compositions. This way, both platform resources (i.e. Flash, RAM) and code are shared between applications.

Contemporary component-based systems have some drawbacks when sharing component instances. Configuration conflicts may arise due to resource competition and contention. While component binding dependencies are explicit in

I. Stojmenovic et al. (Eds.): MOBIQUITOUS 2013, LNICST 131, pp. 524–536, 2014.
DOI: 10.1007/978-3-319-11569-6_41

the form of interfaces and receptacles, implicit dependencies may arise in a software composition due to application level constraints which are enforced through the setting of configuration parameters. Consider a software composition that detects vehicle motion, using a composition of a magnetometer component and a motion detection component. The magnetometer component must sample at 2 Hz in order for the motion detection component to function properly. This required configuration generates an implicit *parameter* dependency between the magnetometer and motion detection components that is not expressible with state of the art component models.

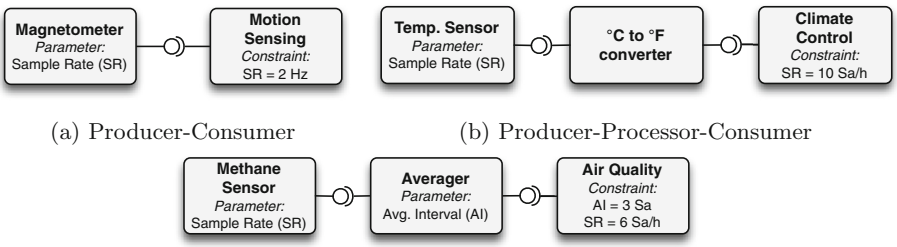
Manually resolving these implicit dependencies is difficult and error-prone. In a multi-purpose WSN infrastructure where multiple actors reuse and reconfigure components, no single actor has an accurate understanding of all existing compositions and parameter dependencies. Existing compositions have to be introspected remotely, which incurs developer overhead and message passing overhead. Failure to resolve an implicit parameter dependency will cause disruption due to erroneous configurations of existing applications.

In this paper we present an approach that externalizes implicit distributed dependencies generated from component parametrization in distributed applications. This allows for automatic composition-safe reparametrization in multi-actor WSNs. Our solution has 3 elements: Firstly, we provide language constructs which can be used by component developers to make parameter constraints and dependencies explicit across component compositions. Secondly, a network protocol is presented which automatically resolves these dependencies when composing components at runtime and flags constrained parameters with the corresponding constraints. Lastly, we introduce a reparametrization algorithm which enforces constraints and enacts the necessary distributed reparametrization when setting constrained parameters, guaranteeing composition-safe reconfiguration.

Our evaluation shows that our approach reduces runtime reconfiguration effort and latency in scenarios where component sharing is necessary. It incurs very little development and runtime overhead that is quickly compensated for. Critically, we notice a significant decrease in network traffic when we compare our reparametrization algorithm to an automated back end graph walk to resolve parameter dependencies.

2 Motivation

One year ago our research group created a smart office environment that supports four applications. These applications are: facility management, workforce management, security and workplace safety, each of which is managed by a different stakeholder. For these applications, we have logged and analyzed reconfiguration effort and latency. Our analysis revealed the previously unknown problem of implicit dependencies between parametrized components in a composition. To further investigate, we conducted a series of experiments designed to better understand the impact that implicit dependencies have on reconfiguration effort and latency.



(a) Producer-Consumer

(b) Producer-Processor-Consumer

(c) Producer-parametrizedProcessor-Consumer

Fig. 1. Classes of compositions with implicit dependencies

Experiment Description. During the experiments, we tasked seven experienced component developers with a series of reconfiguration exercises to be conducted on the smart office. In each exercise the component assembler had to plan and enact an extension to one of the running applications. Throughout the experiment we tracked reconfiguration effort, latency and disruption.

We assume there is no inter-stakeholder coordination of reconfiguration plans and there is no up to date global view of the system. The assemblers all started out having limited information in regards with the configuration and state of all the running component instances and the applications.

The reconfiguration typically happens in 4 steps: (i) reading and understanding the required changes, (ii) identifying and remotely introspecting the components of interest to check if and how they are reusable, (iii) checking if the desired reconfiguration adversely affects any of the running applications, (iv) creating and executing a reconfiguration plan.

Results. Analysis revealed that implicit parameter dependencies arise every time the consumer component has application requirements that constrain the possible values used to configure the functionality implemented by the producer component.

Classes of Component Compositions. We have identified three classes of component compositions in our smart office environment where implicit dependencies are generated, see Fig. 1.

Figure 1a depicts a **Producer-Consumer** composition. This is the implementation of the scenario previously explained in the introduction, where a *Motion Sensing* component requires a *Magnetometer* to sample at 2 Hz. In this case, the *Motion Sensing* component has an implicit dependency with the sample interval of the *Magnetometer*. Therefore the application requirements reified in the *Motion Sensing* component constrain the valid values for configuration property in the *Magnetometer*.

Figure 1b exemplifies the **Producer-Processor-Consumer** composition, where the implicit dependency is propagated from producer to consumer through

a data processing component without any configurable parameters, thus only relaying implicit dependencies. A *Temp. Sensor* component samples at 10 Sa/h, the reading's units are converted and consumed by a *Climate Control* actuator component.

In Fig. 1c, a **Producer-parametrizedProcessor-Consumer** composition is shown. The *Methane Sensor* component samples at a rate of 6 Sa/h, then an *Averager* component aggregates the data of 3 samples, which is consumed by an *Air Quality* component. *Averager* is a data processing component which has a configurable parameter that is constrained by the consumer, i.e. *Air Quality*. Both the *Methane Sensor* and *Averager* have implicit parameter dependencies with the consumer component because *Air Quality* has requirements on the temporal resolution of the readings. This constrains the valid parameter values for both *Averager* and *Methane Sensor*.

The software compositions classified above may be arbitrarily long, for instance having multiple Processors or parametrizedProcessors interconnected in a composition.

3 Background

In this section we discuss the requirements of our approach on component based middleware. We then enumerate the roles that a component can play during composition-safe reparametrization. Finally we provide a classification of constraint types.

3.1 Component Model Requirements

Our approach requires: (i) explicit interface and receptacle declarations, (ii) a **u**nique **id**entifier (*uid*) for each interface and receptacle type and (iii) reparametrization of running components. These requirements are met by all runtime reconfigurable components models, including: RUNES [4], REMORA [14], OpenCOM [5] and LooCI [8].

3.2 Component Roles

Our analysis revealed three component roles, each of which must be considered when resolving distributed parametrization dependencies. We describe each of these roles, with reference to the example smart office compositions shown in Fig. 1.

1. **Constrained components:** These are components which produce events differently based upon their parametrization. A concrete example of such a component is the *Temp. Sensor* component shown in Fig. 1b. The Sampling Rate parameter (SR) influences how often temperature data is sensed and transmitted. which is constrained by the *Climate Control* component.

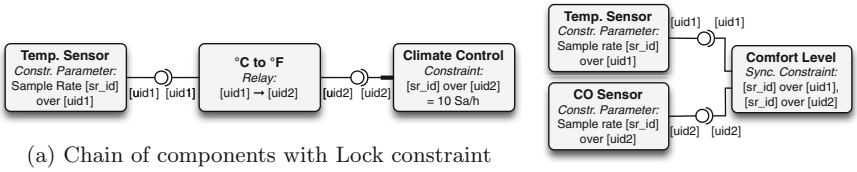
2. **Relaying components:** Relay components do not have constrained parameters, or constrain the parametrization of other components. They do however serve as a relay of parameter constraints along the chain of components in a composition. In Fig. 1b, the *°C to °F converter* is an example of a relaying component. It relays data between a constrained component *Temp. Sensor* and the constraining component *Climate Control*.
3. **Constraining components:** Constraining components consume data that is produced and processed by other components in the composition. Constraining components require a specific parametrization of components producing and processing the data. The *Climate Control* component shown in Fig. 1b is an example of a constraining component. It requires that the sampling rate of the *Temp. Sensor* component has a fixed value.

A component may play multiple roles at the same time. For example, the *Averager* component in Fig. 1c relays a parameter dependency from the *Methane Sensor* (Sampling Rate) and also has a constrained parameter (Averaging Interval). Both of these parameters are constrained by the *Air Quality* component.

3.3 Constraints

Constraining components impose two categories of parameter constraints: *Locking* and *Synchronizing*.

1. **Locking constraints** lock constrained parameters to a specific value or range. An example of this is the constraint imposed by the *Climate Control* component in Fig. 2a. This constraint locks the Sampling Rate parameter of *Temp. Sensor* to 10 Sa/h. Another possibility is constraining the range of acceptable parametrization, e.g. 8 to 14 Sa/h.
2. **Synchronizing constraints** require the synchronization of multiple parameters in the composition. This is illustrated in Fig. 2b, where the sampling rate of the 2 *Sensor* components has to be time synchronized in order for the *Comfort Level* component to aggregate the data and function properly.



(a) Chain of components with Lock constraint

(b) Synchronize constraint

Fig. 2. Component roles and constraint types

4 Design

4.1 Language Annotations

In order to offer composition-safe reparametrization, component developers must specify parameter dependencies, relaying behaviour and constraints. To achieve this, we introduce a set of language annotations.

For *constrained components*, developers use the syntax shown in Listing 1 to identify constrained parameters. Constrained parameters are assigned an id, which is used by the constraint resolution protocol to reference the parameter. The component developer must specify both the id of the constrained parameter and the *uid* of associated the outgoing interface. Figure 2a shows an annotated version of the composition visualized in Fig. 1b. In this scenario, the *Temp. Sensor* component specifies `ConstrainedParameter(sr_id,uid1)` to define the constrained parameter.

For *relaying components*, developers must specify the *uid* of the incoming receptacle and the outgoing interface over which the dependencies have to be relayed. Listing 1 shows the syntax that is used by relaying components. For example, the *°C to °F* component in Fig. 2a specifies `DependencyRelay(uid1,uid2)`.

For *constraining components*, developers use the syntax shown in Listing 1. Constrained parameters are designated by their parameter id together with the *uid* of the incoming receptacle. The tuple formed by this pair of values uniquely specifies a constrained parameter. Lock constraints are specified by appending the parameter identifying tuple with the constraint itself, which is either a range or a single value. Synchronizing constraints are specified by providing a number of parameter-identifying tuples, the values of which must remain the same. The example lock constraint of the *Climate Control* component shown in Fig. 2a is expressed as follows `ParameterLock(sr_id,uid2,10Sa/h)`. The example synchronization constraint of the *Comfort Level* component shown in Fig. 2b is expressed as `ParameterSync({sr_id,uid1},{sr_id,uid2})`.

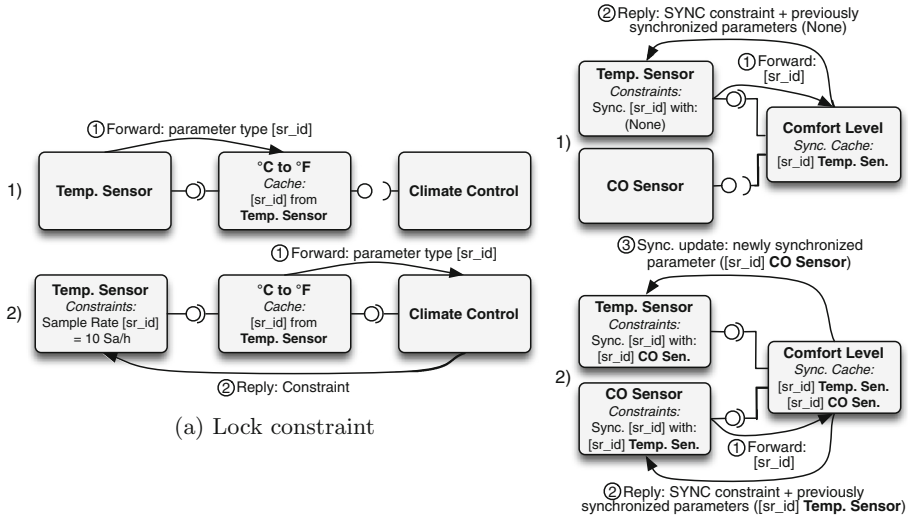
```
ConstrainedParameter(
        parameter-id,   //Reference to constr. parameter
        interface-uid); //Outgoing interface uid

DependencyRelay(
        receptacle-uid, //Incoming receptacle uid
        interface-uid); //Outgoing interface uid

ParameterLock(
        parameter-id,   //Reference to constr. parameter
        receptacle-uid, //Incoming receptacle uid
        constraint);    //Open/closed interval or value

ParameterSync(
        {parameter-id,  //Reference to constr. parameter
        receptacle-uid},//Incoming receptacle uid
        {paramter-id,receptacle-uid}, ... );
```

Listing 1. Language annotations defining implicit parameter dependencies



(a) Lock constraint

(b) Synchronize constraint

Fig. 3. Step-by-step constraint resolution of compositions from Fig. 2

4.2 Constraint Propagation Protocol

In this section, we introduce a network protocol that efficiently relays appropriate constraints from constraining components to constrained components. Bindings may be local, or may cross node boundaries, requiring the transmission of a radio message.

The constraint propagation protocol has two phases. In *phase one*, descriptions of constrained parameters are propagated along the chain of components as bindings are made. A caching mechanism is used at every component along the chain from the constrained component to the constraining component. Every time a new binding is made, the local cache of the component is checked for constrained parameters that should be forwarded. These cache entries are then forwarded along the chain as far as existing bindings allow. Passing constraint information along the component graph is required because components have no a priori knowledge of the application composition in which they will participate. In *phase two*, constraints are sent back directly to the constrained components.

The process of locking constraint propagation for the composition shown in Fig. 2a is shown in Fig. 3a. When the first binding is made, *Temp. Sensor* forwards its constrained parameter to the $°C$ *to* $°F$ component, which caches it. Next, when the last binding is made, $°C$ *to* $°F$ forwards this constrained parameter to *Climate Control*, which matches it with a lock constraint and sends this constraint back directly. A synchronizing constraint must be propagated to all synchronized components and thus the constraining components must store references to each synchronized parameter and its associated component. Figure 3b illustrates how synchronizing constraints are propagated when building

the composition of Fig. 2b. When the first binding is made, *Temp. Sensor* forwards its constrained parameter to *Comfort Level*, where it is matched with a synchronization constraint. As there are currently no other synchronized parameters, none are sent back with the reply containing the synchronization constraint. When *CO Sensor* is bound, the same happens except this time a reference to sr_id on *Temp. Sensor* is sent back with the synchronization constraint to *CO Sensor*. Lastly, an update with a reference to sr_id on *CO Sensor* is sent to *Temp. Sensor*.

4.3 Constraint Enforcement Protocol

The constraint enforcement protocol ensures parameter constraints are maintained during reparametrization. This is accomplished by checking the constraints specified in the constraining component every time a parameter on a constrained component is set.

This requires 3 steps: (i) check all lock constraints on the parameter, and return a *Constraint not met* error message if one is broken, (ii) if no lock constraints are broken, tentatively store the new value for the parameter *(NPV)*, (iii) enforce all synchronization constraints by setting all synchronized parameters on remote components to *NPV*. This recursively triggers the same 3 step constraint check on the remote component. If any remote component returns a *Constraint not met* error message, the local change is rolled back and the same error is returned. If all parametrizations succeed, set the local parameter to *NPV* and return a *Success* message to the parametrizing entity.

5 Implementation and Evaluation

5.1 Implementation

A prototype was implemented on the LooCI [8] middleware. LooCI is a middleware for building distributed component based WSN applications. It complies with all of the requirements listed in Sect. 3. LooCI supports a number of platforms. We implemented our approach on the Contiki [6] based AVR Raven [1] port of LooCI. The Raven mote offers a 16 MHz Atmel MCU, 16 KB of RAM and 128 KB of flash memory. All of the functionality necessary was implemented using LooCI components, requiring no modifications to the middleware.

5.2 Evaluation

We evaluated the performance of our approach against the original version of LooCI in terms of both middleware overhead and commands issued. The performance was assessed in 3 specific scenarios inspired by our Smart Office deployment. The scenarios build on top of each other, and Fig. 4 shows the final complete distributed application composition. In every scenario, functionality is added by deploying new components and binding them to existing ones.

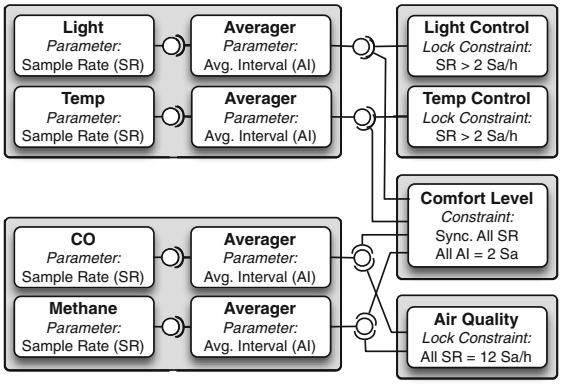


Fig. 4. Distributed component composition used in evaluation

In **scenario 1**, an automatic light/climate control system is deployed over 2 nodes. A first node collects the light and temperature readings and averages them. The controller components are deployed on a separate node and constrain the sensor sampling rates. **Scenario 2** expands upon the first scenario by introducing a node which samples and averages CO and Methane readings. All 4 sensor readings are aggregated in a *Comfort Level* component deployed on a separate node, which evaluates the level of comfort in an office. *Comfort Level* imposes a synchronization constraint on all sensors, together with a lock constraint on the averaging window of all averager components. In **scenario 3**, *Air Quality* is added, which uses the existing *CO* and *Methane* sensors. This component imposes additional constraints on the sampling rate of the sensors.

Component Size Overhead. Component sizes in reconfigurable component models are significant because they largely determine energy expenditure during deployment due to radio usage, and thus impact node lifetime. Our approach requires components to be annotated with metadata. The storage of this metadata incurs overhead on the size of the deployable components. On average, **scenario 1** incurs a component size overhead of 18.3 bytes, **scenario 2** incurs an extra 18.4 bytes and **scenario 3** incurs an extra 19 bytes. These overheads are insignificant when compared to an average unannotated component size of 790 bytes (worst case overhead of 2.4 %).

Static Middleware Overhead. Without any components deployed, LooCI still needs space in both flash and RAM memory in order to provide functionality. The additional components required to implement our approach increase this static overhead. We evaluate both Flash and RAM usage. Considering the 128 KB of flash and 16 KB of RAM available, our modified version uses respectively 5.3 % (65130 bytes vs. 58162 bytes) and 2 % (9351 bytes vs. 9030 bytes) more of the total flash and RAM than the original LooCI. In conclusion, the overhead imposed by our modifications is minimal.

Dynamic Middleware Overhead. The execution of components results in the dynamic allocation of RAM on top of the base consumption discussed in the previous subsection. Figure 5 shows a detailed breakdown of the average allocated memory per node. The allocations can be split in three categories: memory used by component instance bookkeeping, memory used for storing component bindings and dependency and constraint caches. Note that **scenario 2** and **scenario 3** have a higher average overhead then the first scenario in our prototype. This is due to the introduction of the synchronization constraint in the second scenario, which requires more cache space on each node. Taking into account the 16 KB RAM available, in the worst case (**scenario 2**) this means on average 0.83 % more RAM used on each node (213 bytes vs. 76.3 bytes).

Network Overhead. Another point of comparison is the overhead of messages sent over the network when reparametrizing and binding. When binding components dependencies are propagated, generating some message overhead. In case of **scenario 2** and **scenario 3**, we only count the messages sent when expanding the previous scenario. For **scenario 1**, we measured an increase from 6 to 12 messages, for **scenario 2** we increase from 10 to 62 and in **scenario 3** message overhead increases from 4 to 10.

It is clear that the modified version has more overhead due to the constraint propagation protocol, which runs at bind time. This is again most apparent in **scenario 2**, where a nontrivial amount of network traffic is generated keeping the synchronization caches consistent over all nodes. It is important to note that binding is less frequent than reparameterization, where the constraint enforcement protocol gives significant savings.

Table 1 gives an overview of the amount of messages sent when reparametrizing either one of the sampling rate parameters of the sensors, or the interval of one of the averagers. In the original version, the running component composition has to be remotely instrospected to ensure that all constraints are met. This generates significant network usage. A worst case example is the reparametrization

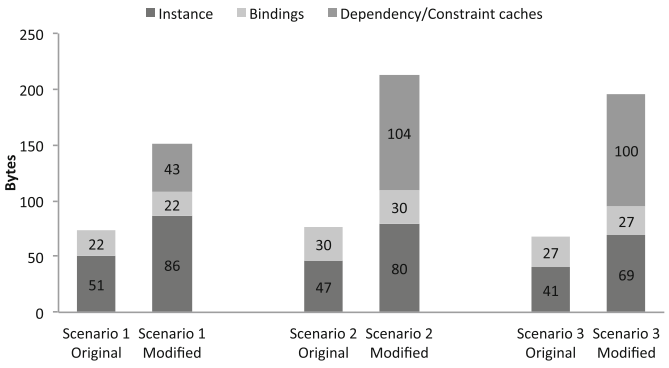


Fig. 5. Breakdown of the average dynamic RAM overhead for each scenario

Table 1. Number of messages and commands sent when reparametrizing

	Scenario 1		Scenario 2		Scenario 3	
	Messages	Commands	Messages	Commands	Messages	Commands
Sample Rate						
Original	8	8	31	31	32	32
Modified	1	1	4	1	4	1
Avg. Interval						
Original	4	4	6	6	6	6
Modified	1	1	1	1	1	1

of the sampling rate in scenario 3, where 28 extra messages have to be sent for each reparametrization. Our approach is thus a significant improvement.

Reparametrization Commands Issued. The advantages of our approach is also most apparent when comparing the amount of commands an actor has to issue and the associated network traffic generated during reparametrization. Table 1 shows this data for each scenario when reparametrizing the sampling rates and the averager intervals. Because the original version has no automatic constraint enforcement, the actor has to introspect the composition manually to ensure that all constraints are met and that the reparametrization is composition-safe. In all cases this generates significantly more commands and messages (on average 24 times more).

6 Related Work

In this section we provide an overview of the state of the art in two areas: reconfiguration approaches used in component-based systems and enhancing reconfiguration by leveraging metadata annotations.

6.1 Reconfiguration in Component-Based Systems

Reconfiguration approaches in component-based systems for WSNs can be broadly categorized into structural and behavioral reconfiguration [2]. Structural reconfiguration is defined by the ability to modify the structure of the component graph. This is achieved by the addition or removal of components and bindings. Behavioral reconfiguration on the other hand allows for the modification of component behavior by offering fine-grained adjustments at runtime.

In the context of WSNs, several well known component-based systems support structural reconfiguration. Examples include RUNES [4], REMORA [14] and LooCI [8]. In these systems, components can be deployed, configured and interconnected at runtime. Our approach builds upon, and is complementary to, these approaches by extending their capabilities through the externalization of implicit parameter dependencies.

Fine grained behavioral reconfiguration is commonly achieved by modifying component configuration parameters. Modifying component configuration parameters is commonly achieved by exposing a configuration interface, as in LooCI [8], RUNES [4] and REMORA [14]. To the best of our knowledge, no existing system checks parameter dependencies.

6.2 Metadata Annotation in Components

Using metadata to inform reconfiguration has been proposed in a variety of approaches. As in Meshkova et al. [11] who proposes a standardized metadata language for component-based WSNs aimed to enhance interoperability. An XML based language is used to describe component dependencies, interfaces, attributes etc. This metadata can be used in the backend for deployment decisions and runtime reconfiguration. Cervantes et al. [3] specify metadata in XML files called *Instance descriptors*, which are deployed along components. These are used to describe a context aware component model which dynamically adapts component compositions based on component availability.

Neither [11] nor [3] would actually run on resource constrained WSN nodes due to their processing and communication overheads. In order to overcome processing overheads they rely either on back end processing or a resource rich platform. Furthermore they are not concerned with energy expenditure, thus transmission overheads incurred in generating updated global view are suboptimal. Our metadata language refrains from using standardized markup languages and does not require updated global view to inform reparametrization. Thus it does not incur in high processing or transmission overheads.

7 Conclusion

In this paper we identified a new problem for distributed component-based systems: implicit parameter dependencies. These implicit distributed dependencies occur when parameters are constrained across component compositions and introduce complexity during reparametrization. We propose a solution which leverages annotated components to externalize and enforce these implicit constraints at runtime. We do this in a way which minimizes messaging over the network and with acceptable memory costs for a resource constrained platform. By resolving parameter constraints in the middleware, our approach greatly simplifies reparametrization by avoiding the need for extensive introspection to identify these implicit dependencies.

Acknowledgements. This research is partially supported by the Research Fund, KU Leuven and iMinds, and is conducted in the context of the COMACOD and ADDIS projects.

References

1. AVR RZ Raven Data Sheet. http://www.atmel.com/Images/doc8117.pdf (2013). Accessed 1 Aug 2013
2. Aksit, M., Choukair, Z.: Dynamic, adaptive and reconfigurable systems overview and prospective vision. In: International Conference on Distributed Computing Systems - Workshops (ICDCS '03), pp. 84–89. IEEE (2003)
3. Cervantes, H., Hall, R.S.: Automating service dependency management in a service-oriented component model. In: International Conference on Software Engineering (ICSE '04), pp. 614–623 (2004)
4. Costa, P., Coulson, G., Gold, R., Lad, M., Mascolo, C., Mottola, L., Picco, G.P., Sivaharan, T., Weerasinghe, N., Zachariadis, S.: The RUNES middleware for networked embedded systems and its application in a disaster management scenario. In: IEEE International Conference on Pervasive Computing and Communications (PerCom '07), pp. 69–78. IEEE (2007)
5. Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., Sivaharan, T.: A generic component model for building systems software. ACM Trans. Comput. Syst. **26**(1), 1–42 (2008)
6. Dunkels, A., Grönvall, B., Voigt, T.: Contiki - a lightweight and flexible operating system for tiny networked sensors. In: IEEE International Conference on Local Computer Networks (LCN '04), pp. 455–462. IEEE (Comput. Soc.) (2004)
7. Filipponi, L., Vitaletti, A., Landi, G., Memeo, V., Laura, G., Pucci, P.: Smart city: an event driven architecture for monitoring public spaces with heterogeneous sensors. In: Conference on Sensor Technologies and Applications (SENSORCOMM '10), July 2010, pp. 281–286. IEEE (2010)
8. Hughes, D., Thoelen, K., Maerien, J., Matthys, N., del Cid Garcia, P.J., Horré, W., Huygens, C., Michiels, S., Joosen, W.: LooCI: the Loosely-coupled Component Infrastructure. In: IEEE International Symposium on Network Computing and Applications (NCA '12) (2012)
9. Huygens, C., Hughes, D., Lagaisse, B., Joosen, W.: Streamlining development for networked embedded systems using multiple paradigms. IEEE Softw. **27**(5), 45–52 (2010)
10. Lau, S.-Y., Chang, T.-H., Hu, S.-Y., Huang, H.-J., Shyu, L.-D., Chiu, C.-M., Huang, P.: Sensor networks for everyday use: the BL-Live experience. In: IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC '06), pp. 336–343. IEEE (2006)
11. Meshkova, E., Riihijarvi, J., Ansari, J., Rerkrai, K., Mahonen, P.: An extendible metadata specification for component-oriented networks with applications to WSN configuration and optimization. In: International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC '08), September 2008, pp. 1–6. IEEE (2008)
12. Rellermeyer, J.S., Alonso, G.: Concierge: a service platform for resource-constrained devices. ACM SIGOPS Oper. Syst. Rev. **41**(3), 245 (2007)
13. Rezgui, A., Eltoweissy, M.: Service-oriented sensor-actuator networks: promises, challenges, and the road ahead. Comput. Commun. **30**(13), 2627–2648 (2007)
14. Taherkordi, A., Loiret, F., Abdolrazaghi, A., Rouvoy, R., Le-Trung, Q., Eliassen, F.: Programming sensor networks using Remora component model. In: Rajaraman, R., Moscibroda, T., Dunkels, A., Scaglione, A. (eds.) DCOSS 2010. LNCS, vol. 6131, pp. 45–62. Springer, Heidelberg (2010)