CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2012)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.2815

SPECIAL ISSUE PAPER

Applying a metadata level for concurrency in wireless sensor networks

Pedro Javier del Cid*,†, Danny Hughes, Sam Michiels and Wouter Joosen

IBBT-DistriNet Katholieke Universiteit Leuven, Leuven, Belgium

SUMMARY

Achieving a clean separation of concerns is a well known approach to improving system adaptability and evolution. We propose to apply this principle to run-time reconfigurable component models for networked embedded systems. By separating configuration properties from runtime component instances, we achieve: (i) improved support for concurrent component use, (ii) optimized resource use, and (iii) reduced effort in runtime configuration management. We demonstrate how this approach can be seamlessly implemented on existing component models for wireless sensor networks without imposing additional constraints and without changes to their API or coordination model. Furthermore, significant memory savings are achieved in concurrent scenarios. Copyright © 2012 John Wiley & Sons, Ltd.

Received 22 September 2011; Revised 6 December 2011; Accepted 14 December 2011

KEY WORDS: wireless sensor network; concurrency; quality of data; resource management; optimization; middleware

1. INTRODUCTION

The complexity of embedded software development, coupled with the need to manage environmental dynamism has steered wireless sensor network (WSN) application developers to implement functionality in a modularized and reconfigurable fashion using component-based software engineering. This has led to a proliferation of component models in the WSN domain, such as: NesC [1], RUNES [2], TinyCOPS [3], and LooCI [4]. As defined by Szyperski in [5], a software component is 'a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by a third party'.

In component-based systems, application functionality is created through the composition of the generic components. Component functionality can be fine-tuned or parameterized through the use of configuration properties. As reported in [2, 3, 6, 7], it is common for parameters such as sampling frequency, processing parameters or actuation thresholds to vary significantly between applications and throughout the lifecycle of a single application.

Early WSN deployments were designed to be used by single-purpose highly specialized applications. At development time the appropriate components were parameterized and connected into a static composition as in NesC [1]. Later, the need for runtime reconfiguration required component models that allowed parameterization and reconfiguration of the application composition at runtime, for example, RUNES [2] and LooCI [4]. New WSN deployment paradigms require concurrent use of WSN platforms while offering fine-grained parameterization of component functionality as

*Correspondence to: P. del Cid, IBBT-Distrinet , Katholieke Universiteit Leuven Leuven, Belgium.
†E-mail: javier.delcid@cs.kuleuven.be

multipurpose WSNs [8]. None of the existing component models for WSN offer explicit support for concurrent use of components by multiple users. If additional users require different parameterization of the component functionality, additional instances of each component must be instantiated, leading to three major problems: (i) rapidly overloading the embedded node's resources, (ii) decreased resource control, and (iii) increased runtime configuration management complexity. Runtime configuration management refers to component deployment, assembly, and activation processes [9]. We propose a technique for separating configuration properties from runtime component instances that addresses these problems and demonstrate how our approach, which we refer to as 'the Metadata level', can be seamlessly implemented on runtime reconfigurable component models for WSNs without imposing additional constraints and without changes to the component model's API or coordination model.

2. BACKGROUND

Our previous experience with a river monitoring and warning scenario deployed in the city of São Carlos, Brazil [4] has allowed us to identify that the need of using WSN functionality with different parameterizations is common wherever multiple users require different data resolution or behavior from the same component — a scenario that is very common in shared deployments that need to minimize infrastructure and deployment costs. Consider the following examples that illustrate how modification of a component's sample interval, which is a configuration property, can significantly modify the semantic value of sensor output: (i) pressure data from a hydrostatic depth gauge is used to monitor wave height and water depth. Wave height monitoring requires continuous sampling of the pressure sensor multiple times per second, while depth monitoring is infrequent and requires averaged readings taken over periods of multiple seconds [4]; (ii) an accelerometer sampled at 125 Hz can be used to determine the orientation of an object, while a sampling frequency of 250 Hz is needed to determine motion [10]; and (iii) a magnetometer only needs to be sampled at 16 Hz to detect vehicle motion, and it can make a magnetic field signature when sampled at 256 Hz [6].

Proposed runtime reconfigurable component models for WSN, for example, TinyCOPS [3], RUNES [2], and LooCI [4] share the basic abstractions and follow similar principles to enable implementation, parameterization, and assembly of components in composition. Components offer their functionality through interfaces, which are defined as a set of types and operation signatures. Components express their dependencies on other components through receptacles. Bindings are used to explicitly model connections between interface-receptacle pairs. Component configuration properties or attributes are key-value pairs used to express metadata pertaining to a component. As such, they can parameterize component functionality or behavior. A component type is the collection of the configurable aspects of a component, that is, interfaces, receptacles and attributes. Therefore, component instances can be instantiated from component types.

The main requirements on WSN component models to enable the use of the metadata level are: (i) components need to be persistent artifacts at runtime, thus the technique is suitable for use with reconfigurable component models like RUNES [2] and LooCI [4], not static models like NesC [1]; and (ii) the availability of a configuration interface that allows component parameterization at runtime, as offered by LooCI and RUNES.

3. THE METADATA LEVEL

We propose the separation of configuration properties or state from runtime component instances. This technique, that is, state management deferral, is commonly used in service-oriented approaches to improve performance and scalability [11]. Stateless component instances can be more easily shared by concurrent users, while allowing the metadata level (MDL) to manage their state and avoid conflicting component configurations, optimize resource use, and ease runtime configuration management. These configuration properties, which parameterize how a component executes its functionality or behavior, are typically part of the internal state of a component instance. In our approach, however, these configuration properties are delegated to and managed by the MDL. They are only returned to the component instance when required during execution. Figure 1 depicts the

DOI: 10.1002/cpe

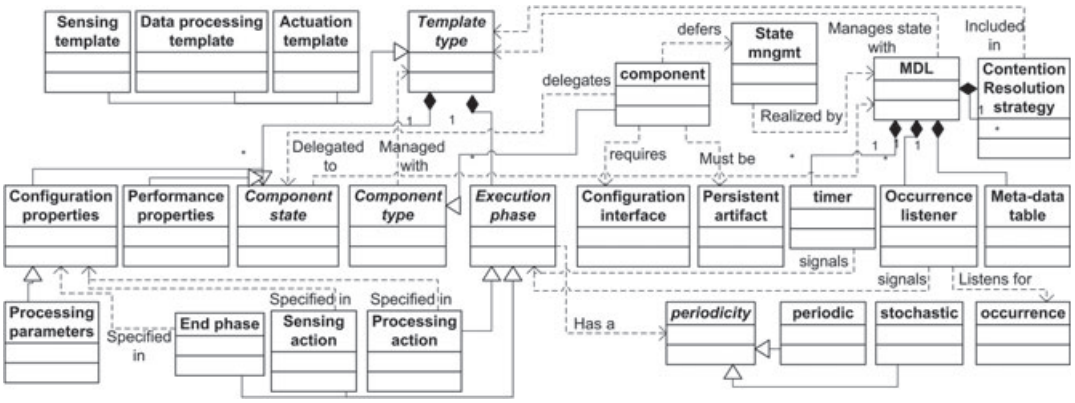

Figure 1. UML diagram that depicts the relations between relevant concepts in our approach.

relations among the relevant concepts in our approach using UML notation, which we further clarify in the rest of this section. Each sensor node requires an instance of the MDL to locally manage component instances.

3.1. Component functionality and state

Components implement functionality, which at application level can be classified as sensing, data processing, or actuation. Sensing functionality provides access to sensors, for example, pressure, accelerometer. Data processing functionality refers to any processing that can be made on sensed values, for instance filtering or aggregation. Actuation functionality provides access to actuators, for example triggering an alarm.

A component's execution can be represented using an execution timeline (see Figure 2(A)), which depicts when the implemented functionality is executed. We refer to these moments as action execution phases. Sensing, processing, and actuation components have an action phase and an end phase. Action phases, that is, sensing action, processing action and actuation action, can be repeated multiple times during a component's execution timeline and the periodicity of this repetition can be periodic or stochastic. Units of time determine periodic repetition and occurrences determine stochastic repetition. Occurrences refer to real world or system events, that is, context events, such as proximity of a target, a flood warning or low availability of free memory. We assume these occurrences are monitored by other elements in the WSN and are outside the scope of this discussion.

3.1.1. Configuration properties. These parameterize how a component executes its functionality. Using configuration properties, a component user specifies when each execution phase should occur and what parameter value(s), if any, should be returned for the action phases for proper execution of component functionality. For example in the case of an average calculating component, the configuration properties may be: (i) processing action start = low memory, (ii) parameter value = 60

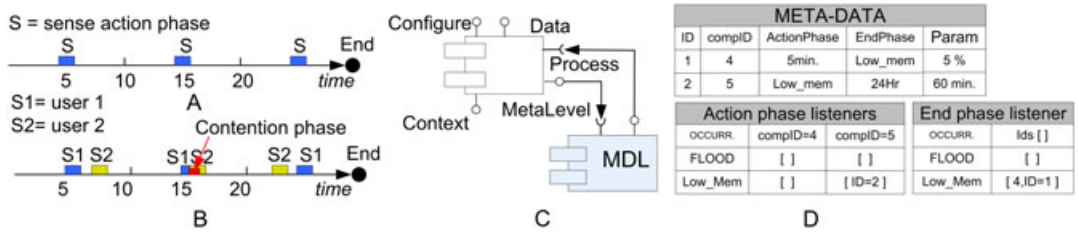

Figure 2. (A) Sensing component execution timeline, (B) execution timeline with two users, (C) simplified component view of the MDL for both prototypes, and (D) depicts the table structures for metadata, action phases, and end phase occurrence-listeners in both prototypes.

DOI: 10.1002/cpe

min, and (iii) end = arrival at destination. These properties specify that the processing action phase should occur every time the occurrence low memory is received and that this configuration is valid until the occurrence arrival at destination is received, thus the end phase is reached. The value 60 would be returned to the component at the processing action to parameterize the averaging period to 60 min. Configuration properties in sensing components are: (i) sensing action start, (ii) parameter values, and (iii) end. For instance in the pressure component we use a parameter to specify the percentage of allowable variation from the expected sensing time. In the actuation components these are: (i) actuation action start, (ii) parameter values, and (iii) end. In a lighting actuator we use a parameter to specify the level of brightness a light should be set to. The amount of parameters and their use are arbitrary, a component developer has the freedom to select different ones.

3.1.2. Performance properties. **C**omponent-specific performance properties can be used, for instance the time required to complete one action phase, which in sensing components is the time it takes to sample once. The component configuration and performance properties comprise the component state that is delegated to the MDL. We refer to this delegated component state as metadata. An MDL template type is generated based on the component properties, which execution phases it has, and a contention resolution strategy (see Section 3.3). We have one template type, from which three templates are generated, and these are: (i) sensing, (ii) data processing, and (iii) actuation. For example, a temperature sensing component is managed by the MDL using the sensing template.

The component user submits configuration properties to a component using the configuration interface provided by the component model's API. Performance properties are arbitrary and hard-coded into the component by the component developers and will be dependent on the component implementation, sensor hardware, and platform in use. Upon the reception of the configuration properties, the component instance delegates the corresponding state to the MDL component using the corresponding interface on the MDL.

3.2. State management in the metadata level

To manage the delegated state or metadata from components a unique identifier (Id) is associated to every metadata set received, which the MDL maintains in a metadata table. Each set of configuration properties submitted by a user to a component will generate a metadata set, thus a single component can delegate multiple metadata sets to the MDL. The MDL extracts the following information from each metadata set: (i) action phase start, for example, sampling interval; (ii) parameter(s) — value(s) that need(s) to be returned to the component at action phase start, for example, time range for the averaging component; and (iii) end — when the end phase is reached, that is, how long the specified meta-data set is valid for. As one may recall from the previous section, sensing action, processing action, actuation action and end, which we refer to as execution phases, can be specified in units of time or upon a specific occurrence. Therefore, the MDL requires a timer to signal time triggered meta-data sets and occurrence-listeners to signal when occurrence triggered metadata sets are active. An occurrence-listener is simply a record in the listener table, where the MDL can associate the metadata set Ids to the corresponding occurrence. A time table is used to associate time and the corresponding metadata set Ids. At the moment when an action phase is signaled, the corresponding parameter value(s) are returned to the corresponding component for sensing, data processing, or actuation. Components discard these values after the completion of each action phase, thus remain stateless. However, because the metadata set is kept in the MDL until the end phase is reached, the MDL will again return the appropriate values to the component at the next processing or sensing action phase.

3.3. Contention resolution and optimization

Component reusability is achieved by sharing a component instance in time, as illustrated in Figure 2(B), where we have plotted the action phases of a sensing component for two separate users. As one can see, there can be a contention phase, whenever functionality is invoked at the same moment by both users. In these situations, a resolution strategy is applied by the MDL. The MDL keeps track of the last time an action phase in a component was called and the duration of

DOI: 10.1002/cpe

each action phase, therefore being able to proactively identify and resolve resource contention. In the case of comparing periodically triggered metadata sets, this contention phase is more easily identified and resolved by applying priority-based scheduling. In stochastically triggered ones, first come first serve scheduling is used. These contention resolution strategies can be modified or extended and are part of each template. In sensing components, these contention phases lead to optimization, because a single sensor reading can be used for both users. Reusing sensor readings in WSNs can significantly reduce energy expenditure [12] and contention over sensors.

3.4. Complexity in the metadata level

Increasing the amount of users that concurrently use a component does not increase complexity in MDL design or execution, because every new user contributes an additional metadata set, which is simply a new record in the metadata table and the corresponding record in the time table or an additional metadata set Id in the record corresponding to the specified occurrence in the listener table. The use of different parameter values at the action phase does not increase complexity; the current templates support this variability. The use of enhanced contention resolution strategies, can impact performance, given that these are usually computationally intensive but they do not increase complexity in use or implementation of the MDL. The current templates support the use and extension of different strategies.

If a developer implements a combination of functionalities in a component, for example sensing + processing, this would mean that there are two different types of action phases and an end phase. To address this, the MDL would need to be extended with an additional template type, and the corresponding data structures for the metadata and occurrence-listeners. However, such an implementation would limit component reusability and code cohesion, as recommended by best practices [5].

3.5. Reusability of elements in the metadata level

Reusability of the three component templates, that is, sensing, data processing, and actuation, which are included in the MDL, is expected to be high. This is because WSNs are commonly limited to sense–process–react usage patterns, where the variability observed lies in the combination of parameter values needed for proper execution of functionality, which is supported by these templates. Furthermore, these templates support the periodic or stochastic execution of functionality and arbitrary addition of parameters and occurrence-listeners without any modifications. Interfaces can also be arbitrarily added, as long as the configuration interface is available. Of course, there also needs to be a process interface and metalevel receptacle (see Figure 2(C)).

4. PROTOTYPE IMPLEMENTATION AND EVALUATION

To prove that the MDL can be seamlessly implemented on WSN runtime reconfigurable component models, we implemented the MDL on the RUNES [2] and LooCI [4] component models, and named them MDL1 and MDL2, respectively. We discuss these prototypes and demonstrate that no restrictions were imposed on the underlying component models or their coordination and that no modifications are required on the component model's APIs. To demonstrate the applicability of the MDL we evaluate its performance, reusability, and support for runtime configuration management. Furthermore, we evaluate our system through a performance comparison against the only available comparable alternative, which is replication based. Replication based concurrency strategies require multiple instantiation of components. To perform the comparison, we compare MDL performance against plain RUNES and LooCI while implementing the same interactions and increasing the number of concurrent users.

We selected these models based on their maturity and relevant differences. They have both been implemented on a wide range of platforms from nodes with 10 KB of RAM and 8-bit microcontrollers to powerful 32-bit devices using C for Contiki [13] and Java, respectively. RUNES uses tightly coupled request-reply based coordination, while LooCI uses a distributed and loosely coupled event-based publish-subscribe approach. We used the Runes Java Kernel–0.2 for Java SE

DOI: 10.1002/cpe

on an Alix board (**PC Engines GmbH**flughofstr.588152 Glattbrugg Switzerland) with a 500 MHz AMD CPU (AMD Sunnyvale, California) and 2MB of RAM from which the RUNES runtime consumes about 880 KB of RAM, and the LooCI runtime version 1.0.2 for Java CLDC 1.1. on the Sunspot (Oracle LABS Redwood Shores, CA) platform (180 MHz ARM CPU (AMD Sunnyvale, California), 512 KB RAM) [10], from which the LooCI runtime consumes about 32 KB and the underlying Squawk VM (Oracle LABS Redwood Shores, CA) [10] about 78 KB of RAM memory.

We implemented pressure sensing and averaging components, as the ones we used for environmental monitoring in a real world deployment [4]. Sensing action and end phases for the former can be set in units of time or by a flood-warning occurrence, for the latter processing action and end phases can be set in units of time or a low-memory occurrence. The averaging component uses a processing parameter that determines the time range for processing, that is, when set to 60 it averages in periods of 60 min. Figure 2(C) is a simplified component view of both MDL prototypes. The averaging component is omitted for readability purposes because it has the same interfaces as the pressure component shown in the diagram. Arrows depict the direction of the flow of the component state. The configure interface is used by the component user to submit configuration properties. The context interface is used to notify the component of an occurrence, which it then delegates to the MDL. The data interface is used to retrieve output data. MetaLevel and Process are the only interfaces added to the components when using the MDL. A segment of the used table structures can be seen in Figure 2(D); both implementation use the same structures. A single metadata table is used in the MDL to manage state from both components. Action phases share an occurrence-listener table; end phase occurrence-listeners have a separate table, where a tuple contains meta-data set Id and source component.

4.1. Seamless implementation

In both prototypes, during component assembly, no changes are made to the way components are instantiated, connected, or interact. The only additional steps required when using the MDL on either prototype are: (i) the instantiation or deployment of the MDL component, (ii) connecting MetaLevel and Process interfaces accordingly, and (iii) passing relevant references to the MDL, that is, abstractions defined by the component model required to connect the corresponding components. Coordination and communication between a component and its users or between components in composition is not affected or intercepted by the MDL. The MDL is also implemented as a component according to the model in use and all interactions between it and any component follow the coordination defined by the model. As one may see no implicit dependencies or additional restrictions are introduced or any changes to the component model's API are needed.

4.2. Performance

We evaluate performance across three dimensions: latency, CPU usage, and memory overhead on both prototypes.

4.2.1. Latency. Using plain RUNES for every new user supported: two components need to be loaded and instantiated (10 ms), configured (0.029 ms), and connected (4 ms). Using the MDL1 most of these actions are only made once. Only the configuration action is needed, per additional user two components configured (0.029 ms). In plain LooCI for every new user supported: two components need to be deployed over a single hop (30 s), activated (70 ms), configured (30 ms), and connected (48 ms). Using the MDL2, one only has to configure (30 ms) and connect (48 ms), for each additional user. In Figure 3(A) one may see the total latency for plain RUNES, LooCI, and both MDL prototypes (plain LooCI above 1 user was omitted for readability). It is evident that using plain RUNES or LooCI the latency overhead quickly becomes considerable.

4.2.2. CPU usage. We have plotted the CPU usage while increasing the number of supported concurrent users for MDL1 and compared against the CPU usage for plain RUNES. The JConsole (Oracle LABS Redwood Shores, CA) tool was used to perform these measurements. Component

DOI: 10.1002/cpe

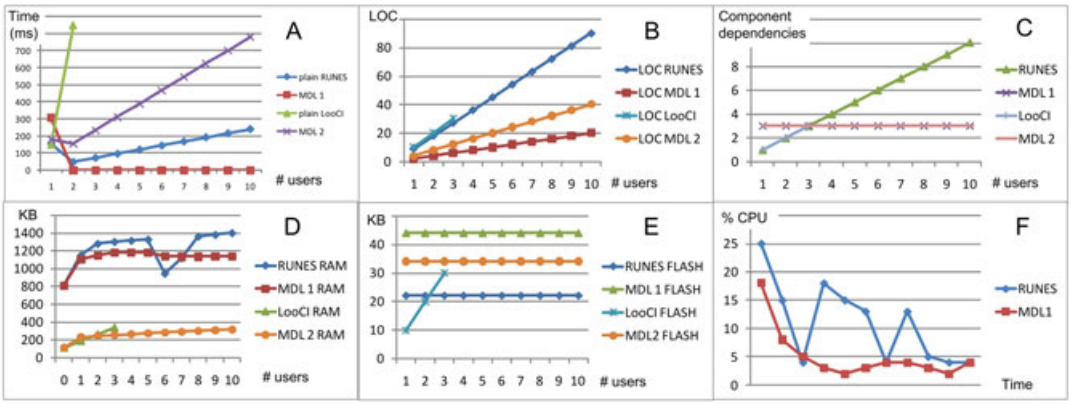

Figure 3. (A) Latency time in ms, (B)LOC for deployment and configuration, (C) component dependencies, (D) RAM memory required, (E) FLASH memory required, and (F) CPU usage.

instantiation accounts for the majority of CPU usage and as one may see in Figure 3(F), the MDL consumes less CPU cycles because each component is only instantiated once.

4.2.3. Memory overhead. We have plotted required RAM memory (see Figure 3(D)) and FLASH (see Figure 3(E)) consumption per user supported in plain RUNES, LooCI and both MDL prototypes. In plain RUNES each component requires 89 KB RAM and 12 KB of FLASH memory and two components per user are required. Using the MDL1 only two components are needed and each consumes 89 KB RAM, 12 KB FLASH and additionally the MDL1 component is needed, which consumes 179 KB RAM and 21 KB FLASH; however, as load increases, there is no need for additional instances. In plain LooCI two new components are required per user, each one consumes 38 KB RAM and 5 KB FLASH. The Sunspot platform can only support seven components because of limitations on the management of isolates by the virtual machine, therefore only 3 users can be supported. In the MDL2 prototype, two components are required and each consumes 38 KB RAM, 7 KB FLASH and additionally the MDL2 component is needed, which consumes 41 KB RAM and 20 KB FLASH; however, as load increases there is no need for additional instances. The benefits of using the MDL are clear when supporting > 2 users in terms of RAM memory. There is little overhead for FLASH memory in RUNES. LooCI would ordinarily require more FLASH to store executable code, but because the SunSpot [10] can only support 7 components, only three users are supported.

4.2.4. Transmission overheads. In plain LooCI, per new user two new components need to be deployed. This requires the transmission of 10 KB of data, which is not the case when using either MDL prototype. In plain RUNES this is not the case because this model has a component factory, which can instantiate additional instances from the existing code base. The MDL does only node local processing, therefore generates no transmission overhead.

4.3. Reusability

The only difference between our prototypes was found in the boiler plate code required by each component model, for example, implementation of mandatory interfaces, registration of component with the runtime, etc. and the code needed to communicate with other components. These differences are due to the fact that the MDL in each prototype is implemented as a component of the corresponding component model. All of the MDL's functionality, data structures, and utility classes were identical, thus demonstrating a very high degree of reusability across implementations on different component models.

DOI: 10.1002/cpe

4.4. Runtime configuration management

To illustrate expected configuration management complexity, in Figure 3(C) we have plotted component dependencies, and in Figure 3(B) the corresponding Lines Of Code (LOC) that the component user must implement to support additional users. In plain RUNES, for every new user, two additional component instances are required and the corresponding LOC are: loading (2 LOC), instantiation (2 LOC), configuration (2 LOC), and connection (3 LOC). Using the MDL1, no new component instances are needed and only configuration (2 LOC) is needed per additional user. In plain LooCI two additional components are required per user and the corresponding LOC are: deployment (2 LOC), activation (2 LOC), configuration (2 LOC), outbound Connection (2 LOC), and inbound Connection (2 LOC). In MDL2 only configuration (2 LOC) and inbound Connection (2 LOC) are needed per additional user. As one can see in Figure 3(B), the additional LOC and Figure 3(C) resulting component dependencies needed, quickly add up. Additionally, in plain RUNES and LooCI manual consistency checks need to be performed per additional user to ensure that any new configurations do not adversely affect existing users. A task that becomes considerably harder in large scale deployments. Using the MDL no manual checking is required to avoid inconsistencies because of conflicting configuration properties. It is important to note that in order for a user to perform these consistency checks introspection support is needed from the component model runtime, which in the case of LooCI is available but not for RUNES.

4.5. Resource control

To improve resource control on plain RUNES and LooCI, one could instantiate a single component, as a singleton instance, to access a shared resource and then instantiate filtering components to obtain the required sampling frequencies. However, this would impose additional constraints for the component user and disruption of data flow every time a new filter needs to be added. Furthermore, this requires the user to manually ensure that the resulting configurations are consistent and that conflicts do not arise. Moreover, this solution would not work appropriately for users that invoke sensors stochastically, for example, when motion is detected. Using the MDL a singleton instance is used to access shared resources and no filtering components are needed to obtain different temporal resolutions.

4.6. Replication based concurrency

Plain RUNES and LooCI support concurrency by using component replication. Only a single user is able to configure the components and connect them in composition, if additional users require to use different values for the configuration properties, additional component instances need to be instantiated. As one may see from Figures 3(D), (F), and (A), there is a significant overhead for both plain component models in memory usage, CPU, and latency. Furthermore, Figures 3(B) and (C) illustrate clear advantages for the MDL in runtime configuration management.

As one can see from the evaluation, configuration properties can be cleanly separated from runtime component instances and this technique can be seamlessly implemented on existing runtime reconfigurable component models for WSNs. Lower RAM memory requirements can be seen above two users. Effort in configuration management is lower in terms of amount of dependencies, LOC and manual checks required. Evident benefit is obtained in performance time for configuration operations of existing functionality. Furthermore, in the case of LooCI, significant reduction in the deployment transmission overhead can be achieved with the MDL. The MDL is capable of managing state for components used in a distributed composition. However, the component model runtime infrastructure needs to provide support for distribution, which is provided in LooCI but not in RUNES.

DOI: 10.1002/cpe

5. RELATED WORK

Research approaches that allow multiple concurrently running applications to share a sensor network can be broadly classified into network level sharing and node level sharing. Network level sharing is based on the idea of partitioning the network, where each node can be dynamically or statically assigned to an application [14]. However, this commonly requires node redundancy. On the other hand, node level sharing approaches provide concurrency support and allow multiple applications to run on the same node, which makes these more cost effective and resource efficient [8]. Node level concurrency support is commonly included in WSN operating systems [13] and virtual machines [10]. Developers are then free to select from various software modularization techniques like modules [13], agents [15], scripts [8], services [6], or components [2]. All these approaches, even those offering loosely coupled interaction for WSNs, for example, publish subscribe based coordination [3, 4], mainly rely on the strategy of instance replication to enable sharing and offer no support to ease runtime configuration management. We opted to focus on component based modularization because of its proven benefits in software reusability [5], evolution, and adaptability [16]. However, our approach is conceptually applicable to other modularization techniques as long as they provide persistent artifacts and explicit configuration interfaces at runtime. Distributed database and multi-query optimization techniques [12], implicitly support concurrent applications. However, they offer limited support for localized actuation and stochastic occurrence based interaction, for example, motion tracking [8].

Approaches that support runtime configuration management have been proposed for resource rich environments in [9, 16] or with the use of component frameworks in [17], which ease configuration effort but do not focus on concurrency in resource constraint environments. In the domain of web services the search to enhance interface matching with semantic awareness focuses more on extending expressivity rather than resource efficiency and optimization [18]. Approaches that focus on mobile environments but still require resource intensive models, as in Puppeteer [19] do not focus on concurrency. In WSNs lightweight reconfiguration management has been proposed [20]; however, they rely on component replication also. Important to note that the MDL could be used with the Figaro approach [20] and offer support for concurrency as it did for RUNES.

Reflective middleware has been used in concert with separation of configuration metadata in [7] to improve application performance but does not explicitly consider concurrency. Previous research efforts have identified the interest in the WSN community to use modular architectures that allow a clean separation of concerns and demonstrated clear benefits from separating system configuration from system logic [21]; however, limited to improve throughput and energy efficiency in the context of data transmission.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have demonstrated how the proposed MDL can be seamlessly implemented on existing component models for WSNs. We have shown that no restrictions need to be imposed on the component model and no modifications to its API or coordination model are required. Ease of runtime configuration management is achieved because the MDL manages component state and ensures that no conflicting component configurations are used, which would otherwise render compositions useless for some users. This relieves the component user from manually insuring consistent compositions. Memory overheads incurred from the use of the MDL are low and quickly compensated under concurrent use.

In our work we assume that concurrency support is provided by the underlying operating system or virtual machine. We have not yet evaluated our approach for the development of system software or lower layer middleware services, where we expect communication latency to be an issue and that the addition of different types of action execution phases can be needed, which can considerably increase complexity of using our approach. Our current implementation does support state management for components used in distributed compositions. However, distributed optimization is not supported, which could provide interesting benefits in WSNs.

DOI: 10.1002/cpe

In our future work we plan to extend the MDL to achieve distributed optimization and to integrate it into a resource optimization framework.

REFERENCES

1. Gay D, Levis P, Von Behren R, Welsh M, Brewer E, Culler D. The NesC language: a holistic approach to networked embedded systems. *Proceedings of ACM SIGPLAN*, New York, NY, USA, 2003; 1–11.
2. Costa P, Coulson G, Gold R, Lad M, Mascolo C, Mottola L, Picco GP, Sivaharan T, Weerasinghe N, Zachariadis S. The RUNES middleware for networked embedded systems and its application in a disaster management scenario. *PerCom'07*, White Plains NY, NY, USA, 2007; 69–78.
3. Hauer J, Handziski V, Kopke A, Willig A. A component framework for content-based publish/subscribe in sensor networks. *Proceedings of 5th EWSN, LNCS*, Bologna, Italy, 2008; 369–385.
4. Hughes D, Joosen W, Michiels S, Huygens C, Matthys N, Thoelen K, del Cid PJ. Building wireless sensor network applications with LooCI. *International Journal of Mobile Computing and Multimedia Communications* October 2010; IGI Global, **2**(4):38–64.
5. Szyperski C. *Component Software Beyond ObjectOriented Programming*. Addison-Wesley: Addison Wesley, Boston, MA, USA, 1998.
6. Liu J, Zhao F. Composing semantic services in open sensor-rich environments. *IEEE Network* July/August 2008; **22**(4):44–49.
7. Capra L, Emmerich W, Mascolo C. Reflective middleware solutions for context-aware applications. In *REFLECTION '01*. Springer-Verlag: London, UK, 2001.
8. Yu Y, Rittle L, Bhandari V, LeBrun J. Supporting concurrent applications in wireless sensor networks. *ACM Proceedings of Sensys06*, Boulder, Colorado, USA, 2006; 139–152.
9. Magnus L, Crnkovic I. Configuration management for component based systems. *Proceedings of Workshops at ICSE*, Toronto, Canada, 2001.
10. SunSPOT. www.sunspotworld.com (visited 08/2011).
11. Erl T. *SOA: Principles of Service Design*. Prentice Hall, Upper Saddle River, New Jersey, USA, 2008.
12. Madden S, Hong W. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems* March 2005; **30**(1):122–173.
13. Dunkels A, Grönvall B, Voigt T. Contiki - a lightweight and flexible operating system for tiny networked sensors. *Proceedings of LCN'04*, FL, USA, 2004; 455–462.
14. Marron P, Lachenman A, Minder D, Hahner J, Sauter R, Rothermel K. TinyCubus a flexible and adaptive framework for sensor networks. *EWSN*, Istambul, Turkey, 2005; 278–289.
15. Fok C, Roman G-C, Lu C. Rapid development and flexible deployment of adaptive wireless sensor network applications. *ICDCS*, Columbus Ohio, 2005; 653–662.
16. Kon F, Yamae T, Hess C, Campbell R. Dynamic resource management and automatic configuration of distributed component systems. *USENIX COOTS*, San Antonio Texas, USA, 2001.
17. 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 Transactions on Computer Systems* February 2008; **26**(1):1–42.
18. Wang X, Vitvar T, Kerrigan M, Toma I. A QoS-Aware selection model for semantic web services. *Lecture Notes in Computer Science* 2006; **4294/2006**:390–401.
19. Lara E, Wallach D, Zwaenepoel W. Puppeteer. Component based adaptation for mobile computing. *USENIX USITS*, San Francisco, Ca, USA, 2001.
20. Mottola L, Picco GP, Sheikh S. FiGaRo: fine-grained software reconfiguration for wireless sensor networks. *Lecture Notes in Computer Science* 2008; **4913/2008**:286–304.
21. Finne N, Eriksson J, Tsiftes N, Dunkels A, Voigt T. Improving sensornet performance by separating system configuration from system logic. *Lecture Notes in Computer Science* 2010; **5970/2010**:194–209.

DOI: 10.1002/cpe