Advanced Sensor Network Software Deployment using Application-level Quality Goals

Wouter Horré, Sam Michiels, Wouter Joosen
IBBT-Distrinet, Department of Computer Science, Katholieke Universiteit Leuven, Belgium
Email: {wouter.horre,sam.michiels,wouter.joosen}@cs.kuleuven.be

Danny Hughes
Computer Science and Software Engineering, Xi'an Jiaotong-Liverpool University, Suzhou, China
Email: daniel.hughes@xjtlu.edu.cn

***Abstract*— If we are to deploy sensor applications in a realistic business context, we must provide innovative middleware services to control and enforce required system behavior. Sensor application developers typically reason about required system behavior in terms of high-level quality goals. Due to the extreme dynamism, scale and unreliability of wireless sensor networks, managing these goals using contemporary software management techniques without support for high-level quality goals quickly becomes challenging. This paper presents QARI, a middleware service which addresses these management challenges by offering a simple yet flexible way to define, enforce, and maintain high-level quality goals for software deployment in wireless sensor networks. We have evaluated QARI using the LooCI component model on two sensor node platforms; results confirm that QARI enables quality aware software deployment for a single application as well as multiple applications, and even in the presence of node failure and mobility.**

***Index Terms*— fault tolerance, large scale networks, management, middleware, quality awareness**

I. Introduction

Wireless Sensor Networks (WSNs) are inherently dynamic, unreliable and large in scale. In addition, WSNs are evolving towards interconnected, multi-purpose sensing infrastructure that is expected to host multiple applications. These applications may be deployed by diverse actors, who make use of sensor network infrastructure in return for direct payment or reciprocal application hosting.

From an application perspective, an important requirement for software deployment in such interconnected, multi-purpose sensing infrastructure is the compliance with high-level quality goals. For example, to gather sensor data with sufficient accuracy, an application might impose coverage requirements on the deployment of its sampling component. Current software management techniques for WSNs do not provide application developers with tools to specify these quality requirements. As a consequence, managing software deployment on a WSN quickly becomes challenging.

In our previous work [1], we introduced QARI (Quality Aware Reconfiguration Infrastructure), a middleware service for quality aware software deployment. Middleware is traditionally used to address the software management challenges in complex network infrastructure; the complexity of WSNs is orders of magnitude greater than traditional distributed systems and thus requires sophisticated software management support in the middleware.

In this paper, we provide a detailed description of QARI's *quality-aware deployment specifications* and QARI's implementation. We also show QARI's platform independence by providing an implementation on a second hardware platform.

QARI addresses the software management challenge by offering a way for application developers to express their goals in a *quality-aware deployment specification*. QARI takes these specifications as its input and automatically enforces and maintains the quality requirements. In QARI's decentralized approach, the responsibility of assigning nodes to applications and monitoring these nodes to maintain the required quality level over time, is delegated to a local management entity close to the deployment target. This allows QARI to use locally gathered context data and specific characteristics of the deployment target to inform efficient deployment of software.

The remainder of this paper is structured as follows: first, we discuss related work in Section II. Then, we present details on QARI and evaluate it in Sections III and IV. Finally, we conclude and reflect on future work in Section V.

II. Related work

This section analyzes the state of the art in software deployment support for WSNs, quality awareness in WSNs and planning techniques in WSNs.

Three categories of runtime support for deployment and reconfiguration can be distinguished [2]:

- **Monolithic:** replace all functionality during the update by re-flashing and re-starting the nodes.
- **Script-based:** change the behavior of previously deployed functionality by injecting lightweight scripts.
- **Modular:** replace coarse-grained units of functionality (modules) at run-time.

Deluge [3] is a reliable epidemic code dissemination protocol that is used to support monolithic flashing of

© 2011 ACADEMY PUBLISHER
doi:10.4304/jsw.6.4.528-535

a network of TinyOS [4] motes. Deluge focuses upon achieving good network performance, providing high throughput and imposing minimal additional overhead due to control messages.

Replacing a complete code image using Deluge with only small changes to the behavior implies a large energy overhead for a small functionality change. Script-based approaches such as Maté [5] address this by supporting the injection of lightweight scripts that are interpreted on the sensor nodes to drive the execution of pre-deployed TinyOS functionality. DVM [6] combines this approach with the dynamic modules of SOS [7] to remove the pre-deployed limitation of Maté.

The Sun SPOT [8] and Sentilla Perk [9] WSN platforms are Java based and allow for modular reconfiguration via the dynamic deployment of MIDP 1.0 compliant Java applications. SOS [7] and Contiki [10], [11] provide similar support for the deployment of binary application modules that are dynamically loaded into the operating system. While this is an improvement over monolithic approaches, the relationships between modules are opaque and may not be reconfigured.

On top of these runtime systems, reconfigurable component systems (such as OpenCOM [12] and LooCI [13], [14]) provide additional support to change the composition of reconfigurable functional blocks (components). A combination of these runtime types might be used to achieve this: for example, DAViM [15] uses a modular runtime to update coarse grained units of functionality (components) and a script based approach to change the interaction between these components.

The code distribution mechanisms used by these approaches typically provide whole network updates (e.g. Deluge [3], Sentilla Perk [9]) or single node updates (e.g. Sun SPOT [8]). We believe that the set of nodes where a component needs to be deployed, depends upon the quality requirements of the application. QARI therefore aims to provide quality aware management of deployed functionality regardless of the deployment approach and code distribution technique used.

The quality-of-service (surveillance) that can be provided by a particular WSN is commonly expressed in terms of k–coverage or related coverage metrics. Most approaches [16], [17] calculate coverage after deployment, propose node deployment heuristics or otherwise use coverage information in single application WSNs. We propose to exploit the knowledge of coverage requirements to optimize the software deployment process in shared WSN scenarios.

Macro-programming approaches, such as Magnet OS [18], RuleCaster [19] and COSMOS [20], compile a network level program into smaller programs that can run on a single node. These node level programs are then assigned to nodes by the system. If assignment based on node capabilities yields multiple possibilities, these approaches perform optimization of a certain system property (e.g. path length of data packets between communicating nodes, overall energy consumption, etc.). QARI introduces support for user specified quality requirements as an additional assignment requirement.

Semantic Streams [21] is a framework that allows to pose declarative queries over semantic compositions of sensor streams. The framework allows user specified cost functions to be attached to queries. The planner uses this cost function to choose between possible execution scenarios for the query. Semantic Streams thus allows users to influence the trade offs made by the system (e.g. energy consumption versus accuracy). However, this affects only the querying of pre-deployed functionality.

MiLAN [22] also addresses the topic of determining the optimal subset of sensors that is required to reach a desired quality-of-service level. MiLAN confirms that there are usually multiple subsets of sensors that can provide the required data, but with different quality-of-service properties. We believe that this knowledge can be exploited to improve and optimize the software deployment process in shared sensor network scenarios.

III. Quality aware software deployment for WSN

This section describes our approach to quality aware software deployment for WSN. Section III-A provides a high-level overview of the approach and discusses the deployment specifications used in our approach. Section III-B describes how network monitoring and software deployment functionality is realized. The planning of the deployment process is discussed in Section III-C.

A. High-level overview

Figure 1 shows an overview of QARI, with Application Managers and Network Managers as key abstractions. Application Managers are responsible for the management of one or more sensor network applications. Application Managers split their centralized deployment plans into smaller deployment specifications for decentralized processing by Network Managers [23]. These specifications include a description of the component to deploy as well as details of the desired quality level (expressed as an interval from the minimally required level to the preferred level). We therefore call them *quality aware deployment specifications*.

The per-WSN Network Manager, which is close to the WSN, is responsible for merging specifications into a single target specification for the software deployment of the WSN. It deploys and updates the software on the WSN to reach and maintain this target state. The merger of specifications with complex quality requirements might require a conflict resolution strategy to deal with potential conflicting requirements, however, we have not yet addressed this issue in the current version of QARI.

This approach provides a clean separation of concerns. The definition of quality requirements, which requires application specific knowledge, is handled by the Application Managers. The responsibility for achieving and maintaining these quality requirements, a task which requires local, network specific knowledge, is handled by

© 2011 ACADEMY PUBLISHER

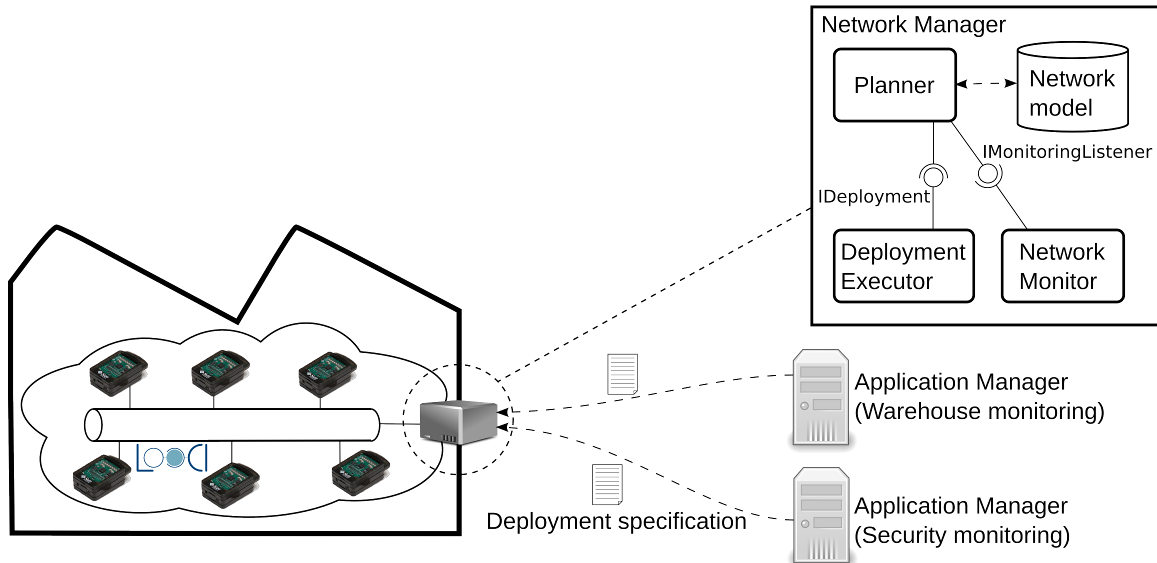

Figure 1. An overview of QARI, our approach for advanced software deployment using application-level quality goals, including the architecture of the Network Manager (top right).

the Network Manager. The ability to specify the required quality for a target also raises the abstraction level for the Application Managers compared to state-of-the-art approaches that allow only whole network or single node deployment (cfr. Section II).

Consider, for example, a warehouse monitoring scenario in which a storage company, STORAGE-CO, uses a WSN infrastructure—with LooCI [13], [14], our reconfigurable component model—to monitor the conditions in a temperature controlled warehouse. Nodes participating in the STORAGE-CO WSN are evenly distributed throughout the warehouse. Each node has an effective radius for which it can provide sensor readings with sufficient accuracy (the node is said to *cover* this area). STORAGE-CO's warehouse monitoring application uses the WSN infrastructure to monitor both temperature and relative humidity in the warehouse. To have a usable view of the warehouse conditions, the monitoring application requires temperature measurements of a subset of the nodes that at least cover all areas of the warehouse. For better accuracy, double coverage (i.e. at least two nodes that cover each area) is preferred. Humidity measurements of at least 40% of the nodes in the network are required, while 60% is preferred.

The quality aware deployment specification for this application on the sensor network is shown in Figure 2. The format is an extension of Service Component Architecture (SCA) [24], a vendor and technology neutral format for specifying component assemblies. As of now, QARI only supports components that are wired by the implementation. Support for composition in the quality aware deployment specification itself is one of our top priorities for future work on QARI.

The specification in Figure 2 contains the temperature sampling component (TEMP-SAMPLING) and the humidity sampling component (HUMIDITY-SAMPLING) needed by STORAGE-CO's warehouse monitoring application. Both components have a target named *WSN1* (cfr. lines 4 and 10), i.e. they need to be deployed on the sensor network with name *WSN1*. The target specification includes a quality requirement: in this case the TEMP-SAMPLING component requires single coverage and prefers double coverage (cfr. line 5). The HUMIDITY-SAMPLING component requires at least 40% of the nodes in the network, but prefers 60% (cfr. line 11). In addition, the specification contains a hint about the type and location of the component implementation (cfr. lines 7 and 13).

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <composite name="TEMP-HUMIDITY">
3   <component name="TEMP-SAMPLING">
4     <target name="WSN1">
5       <goal.coverage min="1" pref="2"/>
6     </target>
7     <implementation.looci.sunspot file="
          tempsample.jar"/>
8   </component>
9   <component name="HUMIDITY-SAMPLING">
10    <target name="WSN1">
11      <goal.fraction min="0.40" pref
            ="0.60"/>
12    </target>
13    <implementation.looci.sunspot file="
          humidity.jar"/>
14  </component>
15 </composite>
```

Figure 2. Example of a quality aware deployment specification. The format is an extension of Service Component Architecture (SCA) [24]. Non-essential parts have been left out for brevity.

B. Enacting and monitoring deployment

The *Network Monitor* and *Deployment Executor* are the building blocks of QARI responsible for monitoring the WSN and enacting deployment actions on the WSN

© 2011 ACADEMY PUBLISHER

respectively. The *Planner* is responsible for deployment planning and maintenance (see Section III-C). It uses the feedback from the *Network Monitor* and *Deployment Executor* to keep its view of the WSN (stored in the repository called *Network Model*) up to date (see Figure 1).

The *Network Monitor* and *Deployment Executor* are the interface to the WSN and apart from their deployment and monitoring functionality, they perform two additional tasks: (1) they provide a point of virtual synchrony (i.e. they expose monitoring of and deployment to the WSN as operations on a synchronous system, although the WSN is asynchronous in nature) [25] and (2) they abstract over the platform specific details of WSN. To perform their tasks they use platform specific knowledge, e.g. to select appropriate timeouts for node failure detection, to select the appropriate deployment mechanism for a given deployment request, etc.

The *Network Monitor* uses platform specific knowledge and mechanisms for gathering information about the WSN. It implements a publish-subscribe interface over which this network information is disseminated to interested software entities (such as the *Planner*). Remote entities that wish to be notified of changes in the WSN environment, should implement the *IMonitoringListener* interface (pseudocode):

```
void nodeOffline(address)
void nodeOnline(address)
void componentChanged(component, address,
    state)
```

The interested parties subscribe themselves at the *Network Monitor* which then pushes notifications of node failures, notifications of node arrivals and notifications of status changes of a deployed component through the *IMonitoringListener* interface.

The *Deployment Executor* provides an abstraction on top of one or more platform specific deployment mechanisms. The *Planner* interacts with the *Deployment Executor* through the *IDeployment* interface (pseudocode):

```
address[] deployComponent(component,
    address[])
address[] startComponent(component, address
    [])
address[] stopComponent(component, address
    [])
address[] undeployComponent(component,
    address[])
```

The *IDeployment* interface allows the *Planner* to deploy, start, stop and undeploy a component—in the remainder of the paper, we use the term component for a piece of software that is independently deployable— on a given list of addresses. Each of the methods indicates in its return value for which addresses the action succeeded. The interface is called in synchronous fashion by the *Planner*, thus it is up to the *Deployment Executor* to decide when an action must be considered unsuccessful (and thus provide virtual synchrony over the asynchronous WSN).

Although state-of-the-art code distribution techniques do not provide code deployment to groups of nodes, the *IDeployment* interface is defined in terms of groups nonetheless. This will allow the *Deployment Executor* to take advantage of group deployment techniques as soon as they become available. In the meantime, the *Deployment Executor* can easily realize the interface through sequential or parallel single node deployments.

C. Deployment planning and maintenance

The *Planner* of the Network Manager assigns components to sensor nodes based on the specifications it receives from the Application Managers. The calculation of these assignments uses local knowledge about the WSN. The *Planner* also updates the assignments in reaction to significant changes in the WSN, which it is notified of through the *Network Monitor*.

As discussed in Section III-A, the specifications contain a description of the component that is requested—called component type in the remainder of this paper—and a deployment target with two quality levels: a minimal and a preferred level.

Initial deployment: When a component type is to be deployed for the first time, the *Planner* first calculates an assignment that achieves the minimal quality level needed for the component type. After the component type has been deployed to this initial assignment, the assignment is updated to achieve the preferred quality level for the component type.

Combining specifications: The *Planner* calculates only one assignment per component type. If there are multiple specifications for the same component type coming from different sources, the *Planner* calculates an assignment for the component type that satisfies at least the minimal quality level of all specifications. If possible, the *Planner* will try to satisfy all preferred quality levels.

If for example the fire detection application of STORAGE-CO also wants to use the WSN infrastructure in the warehouse to monitor the temperature, it will submit a deployment specification for a TEMP-SAMPLING component. Suppose the fire detection needs more accurate data and requests a minimal coverage of two nodes and a preferred coverage of three nodes. The *Planner* will merge this specification with the one shown in Figure 2, resulting in a minimal coverage of two and a preferred coverage of three for the TEMP-SAMPLING component.

The possibility of sharing component deployments across multiple applications is a clear benefit of delegating deployment to the WSN edge. The individual Application Managers have no means to optimize deployments based on the requirements of other applications. However, since the Network Manager is responsible for enacting all deployments, it has the knowledge required to perform such optimization.

Failures: The *Planner* reacts to node and component failures by calculating a new assignment. The new assignment is updated locally—i.e. only in the vicinity of the failed node—to repair the quality level while making as little changes as possible to the original assignment. This

© 2011 ACADEMY PUBLISHER

minimizes the disturbance to the network and the time needed to reach the targeted quality level again.

Node mobility: Node mobility is handled in a similar manner to node failure: the component assignment is updated locally around the area where the node left. If the node is not needed to ensure the quality level in the area it moved to, the corresponding component will be removed from the node.

Network sharing: The *Planner* uses local knowledge about the state of the WSN (gathered through the *Network Monitor*) to minimize the interference between applications. As discussed above, the *Planner* combines multiple specifications for the same component type.

For different component types, two measures are taken to minimize the interference. First, the *Planner* takes the state of a sensor node into account during the calculation of an assignment. One of the parameters which is accounted for is the number of components already deployed on a sensor node. This way, different components are spread as evenly as possible across the WSN.

Second, the *Planner* schedules deployment actions so that at least the minimal quality level of other component types is maintained during the deployment. To realize this, the *Planner* splits a deployment into multiple smaller deployments that have less effect on the network. That way, the minimal quality level of other component types is only broken if it is absolutely necessary.

Consider the example where a security application of STORAGE-CO wants to deploy a light sampling component to detect unauthorized overnight accesses to the warehouse. This application will require a decent coverage, thus resulting in a deployment to a significant subset of the nodes. If QARI would simultaneously deploy to this complete set of nodes, these nodes would be busy deploying the new component all at the same time. This would result in too many nodes being unavailable simultaneously and the quality requirements for the TEMP-SAMPLING and HUMIDITY-SAMPLING components would be broken. Therefore QARI will deploy the new component to smaller groups sequentially.

IV. Evaluation

This section evaluates a prototype implementation of QARI. We evaluated the *Planner* in two ways. First, we implemented a prototype of the *Network Monitor* and *Deployment Executor* for a WSN using the LooCI component model [13] on a test-bed of 9 standard Sun SPOT motes (180MHz ARM9 CPU, 512KB RAM, SQUAWK VM 'RED' version) [8]. Second, we implemented a stub version of the *Network Monitor* and *Deployment Executor* that simulate a grid of 10x10 sensor nodes. This allowed us to move beyond simple feasibility tests and conduct the analysis provided in the Sections IV-C to IV-E.

A. Implementation

We implemented the *Monitoring* for both a simulated testbed and for the LooCI component model on Sun SPOT hardware. The implementation for LooCI depends upon the LooCI introspection capabilities that allow to query the state of nodes and components. The *Monitoring* performs periodic discovery of the network state using the LooCI introspection calls. When changes are detected, the subscribers will be notified through the *IMonitoringListener* interface.

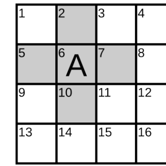

Figure 3. Coverage for Node A

We have implemented this basic version of the *Monitoring* to support our proof-of-concept implementation and the evaluation in this paper. There is clear potential for further improving the *Monitoring*, for example in case of highly dynamic networks; yet, we consider these optimizations out of scope for this paper.

The *Deployment Executor* has also been implemented for a simulated testbed and for LooCI on Sun SPOT. The loose coupling between LooCI components is of great benefit to QARI. It allows QARI to easily reassign components to other nodes if the network situation changes. The flexible loose bindings of LooCI can be easily updated without the need to put all components involved in the binding in a reconfiguration safe state.

In addition, to confirm the platform independence of the *IDeployment* interface, we implemented the *Deployment Executor* for the Contiki 2.4 [10] port of the LooCI component model on AVR Raven hardware [26]. To date, we only used the Contiki/AVR Raven platform to perform measurements of the deployment time. Since these first results are promising, we intend to use the platform in an industrial case study to further evaluate QARI outside the lab.

B. Case Study and Coverage Heuristic

We evaluated our approach in the context of the warehouse monitoring scenario introduced in Section III. The nodes participating in the STORAGE-CO WSN are evenly distributed throughout the warehouse which is divided into 100 unique areas (9 areas for the physical testbed). For the purposes of our case study, we simplify the effective radius of a node to the unique areas adjacent to the location of the node. Figure 3 illustrates our notion of coverage for Node A, which is deployed in unique area 6 and may be used to monitor the surrounding areas (2,5,7,10).

The optimal assignment of components to nodes in a distributed system is known to be a hard problem, for which heuristics are often used. Our implementation of the Network Manager *Planner* also uses a heuristic algorithm to calculate an assignment that satisfies a coverage requirement. The algorithm (see Figure 4) orders the areas based on the number of nodes that are able to cover the area. It then considers each of the areas in this order.

© 2011 ACADEMY PUBLISHER

$assignment :=$ empty set
$areas :=$ all areas in warehouse
sort $areas$ by number of covering nodes
for $area \in areas$ do
    $candidates :=$ nodes covering $area$
    sort $candidates$ by score
    while $area.coverage <$ required coverage
        $\wedge$ $candidates$ not empty do
        $candidate := candidates$.remove(0)
        if $candidate \notin assignment$
            then $assignment$.append($candidate$)
        fi
    od
od

Figure 4. The heuristic algorithm used to calculate an assignment (pseudocode).

For each area, the algorithm adds the *best* nodes to the assignment until the area is sufficiently covered. The *best* nodes are defined as the nodes that are not already in the assignment and have the best score associated with them. The score is calculated based on the number of currently uncovered areas the node covers, the number of components already running on the node and the reputation of the node. The reputation of a node is used to penalize nodes that have failed in the recent past.

If there is an assignment that satisfies a coverage requirement, the algorithm is always able to find it, since it keeps adding nodes until all areas are sufficiently covered. The algorithm doesn't always find the best assignment (i.e. minimal number of nodes), but our experiments show that it always finds a good assignment (i.e. close to the minimal number of nodes) with respect to the current state of the WSN.

C. Single application coverage

To demonstrate the feasibility and correctness of our coverage heuristic, we applied different coverage requirements for a single component type to our 10x10 simulated testbed. The experiments defined quality requirements by setting the following minimal/preferred coverage parameters: 0/1, 1/2 and 2/3. We executed each experiment 10 times, each of which yielded the same deployment pattern.

The realized coverage patterns for each experiment are shown in Figure 5. In the case of 3/4-coverage, there is no assignment that provides 4-coverage since the corner areas have only three neighboring nodes. In such cases, our system generates an "as good as possible" assignment: all other areas still maintain sufficient coverage to meet the requirements.

Once the areas in the grid are sorted, the prototype calculates the assignment in $O(n)$ time, where n is the number of areas. The calculation of 1000 assignments for a 10x10 grid requires on average 6233 ms (10 samples; standard deviation 25 ms) on a standard desktop (Sun Java 1.6.0_16, Pentium D 3.2GHz, 1GB RAM). The time to calculate the assignment is thus orders of magnitude

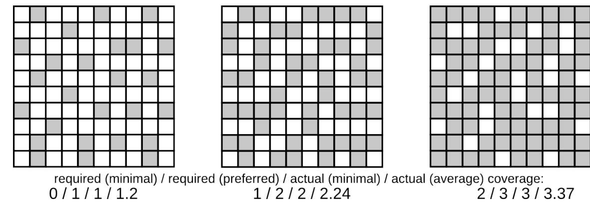

Figure 5. Calculated deployment patterns for a single component type.

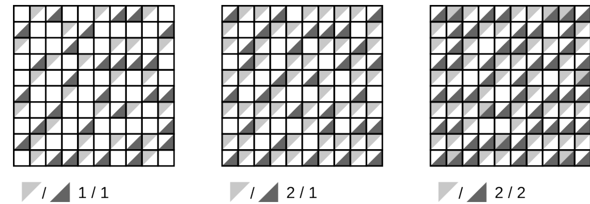

Figure 6. Calculated deployment patterns for a shared network: TEMP-SAMPLING (light gray) and HUMIDITY-SAMPLING (dark gray). The preferred coverage for each deployment is also indicated in the figure.

smaller than the time it takes to actually enact it in the network (see below).

Transferring the TEMP-SAMPLING component requires 1664 bytes to be sent to a selected Sun SPOT node, installing and starting it takes on average 8.63 seconds (10 samples; standard deviation 0.24s). Deploying the TEMP-SAMPLING component to an AVR Raven node with LooCI and Contiki 2.4, requires 1576 bytes to be sent and takes on average 11.62s (10 samples; standard deviation 0.01s) to deploy and start. Even though the implementations used in these experiments are not optimized, the results show considerable improvement in throughput compared to manually updating each sensor node in the field (even in the case of this relatively small test case).

D. Multiple application coverage

QARI not only enables the deployment of components for a single application (i.e. the same component type on every selected node), it also allows for sharing the WSN infrastructure by multiple applications. We have verified this by deploying both a TEMP-SAMPLING and a HUMIDITY-SAMPLING component.

Figure 6 shows the deployment patterns of this test case; again, three experiments were defined with different minimal/preferred coverage requirements: (1) single coverage for both the TEMP-SAMPLING and the HUMIDITY-SAMPLING component, (2) double coverage for the TEMP-SAMPLING component, and single for the HUMIDITY-SAMPLING component, and (3) double coverage for both applications.

As illustrated, QARI is able to maximally spread different applications over the WSN, while not breaking the coverage requirements of other applications. Only when no alternative deployment patterns exists (Figure 6 right side), QARI will deploy multiple applications on a single node.

© 2011 ACADEMY PUBLISHER

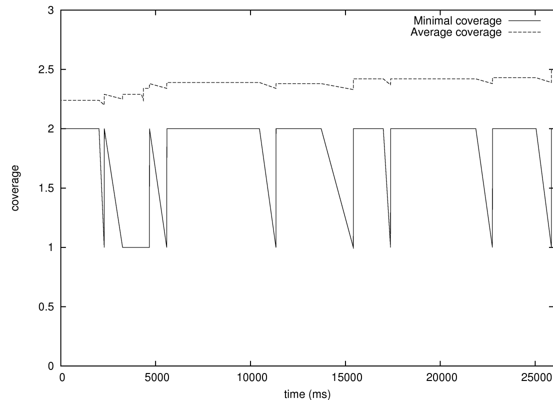

Figure 7. Minimal and average coverage over time in the presence of node failures.

E. Resilience to node failure and mobility

Having sketched the feasibility of QARI to achieve coverage requirements in an automated and correct way, we finally show that QARI is able to maintain quality requirements in the context of unreliable network conditions, i.e. deployment failures, node failures and node mobility.

We have reevaluated the single application test case (cfr. Section IV-C), and randomly injected failures during component deployment. QARI succeeded in establishing a usable deployment pattern; obviously, in case of failures, the generated deployment patterns deviate from the patterns in Figure 5 (more nodes are needed to achieve sufficient coverage).

In addition, we evaluated the capability of QARI to maintain coverage requirements in case of node failures. We evaluated this by randomly removing nodes from and reinserting nodes into the testbed and verifying the system's reaction. As expected, QARI was able to (1) detect that coverage was no longer optimal and (2) reapply the heuristic to start a new deployment cycle. As long as enough nodes were alive, QARI was able to re-establish a coverage pattern that satisfies the requirements.

QARI minimizes the impact of a node failure by locally—in the vicinity of the failed node—repairing an assignment instead of globally re-assigning component types to nodes. For each of the 10 experiment runs, recovering from a failure that affected an existing assignment took at most 3 repair actions; on average only 1.69 repair actions were needed.

Figure 7 shows the minimal and average coverage over time for one of these runs. The coverage requirements for this experiment were 1 and 2 (minimal and preferred). QARI is able to keep the minimal coverage that is actually achieved between these bounds. It can be seen from the figure that at around 3s, the minimal coverage achieved is lower than the preferred coverage for about 1.5s. This is because not enough nodes were available to realize the preferred coverage for some areas. QARI recovers from this situation when enough nodes reappear. Even if it is not possible to meet the requirements—preferred coverage or in the worst case minimal coverage—, QARI tries to do “as good as possible”, resulting in a good level of average coverage despite the fact that the coverage of some areas is lower than preferred.

Node mobility is handled in a similar way to node failure: a mobile node disappears in one area (cfr. node failure) and appears in another (cfr. node reappearing).

F. Discussion

In the previous sections we illustrated the feasibility of our approach and presented a quantitative evaluation of our prototype implementation. While we did not perform quantitative scalability tests, we expect that these tests would show that the scalability of QARI is bounded by the scalability of the underlying deployment mechanism. This belief is based on the fact that the *Network Manager* runs on a resource rich device at the edge of the WSN and our measurements in Section IV-C indicate that the *Network Manager* offers good performance on such devices.

The number of node failures that QARI can tolerate without breaking the quality requirements will of course depend on the level of node redundancy in the network. Also the failure pattern will influence the failure rate that can be tolerated in the case of a coverage requirement. If the failed nodes are concentrated in a certain area of the network, less failures can be tolerated than when the failed nodes are spread over the whole network. If the number of component types on the network is larger, the number of possible assignments will also be limited by the capabilities of the nodes to run multiple components simultaneously. This will also have an effect on the number of failed nodes that can be tolerated.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we presented QARI, a decentralized, quality aware deployment service for inclusion in management middleware for interconnected, multi-purpose sensing infrastructure. QARI provides separation of concerns between defining quality goals (requiring application knowledge) and achieving these goals in a wireless sensor network (requiring knowledge of the physical deployment).

QARI offers application developers a simple, yet flexible way to specify deployment goals, including specification of the desired quality. The realization of these specifications is delegated to an entity close to the sensor network. This entity uses locally gathered contextual data to achieve the required quality and to maintain it throughout the lifetime of the application.

We have realized a prototype implementation of QARI for the LooCI component model on both the Sun SPOT platform and the Contiki/AVR Raven platform. We evaluated QARI using this prototype as well as a simple simulation environment.

We are currently evaluating QARI in the IBBT-AdMid project in a real-world case study in collaboration with industrial partners. To support this work, we will develop additions that further increase the applicability of QARI in such real-world use cases. First, we intend to extend the

© 2011 ACADEMY PUBLISHER

deployment specifications with support for compositions and more rich quality specifications. Second, we will extend our approach to support the deployment of policies using the policy-based management framework for LooCI (PMA) [27]. Third, we will investigate the possibility to allow the network administrator to express constraints for its network (e.g. the maximum number of components per node or minimum battery level required for deploying a new component to a node).

ACKNOWLEDGMENT

Wouter Horré is a PhD fellow of the Research Foundation - Flanders (FWO). This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, by the Research Fund K.U.Leuven, by the Flemish agency for Innovation by Science and Technology (IWT) in the context of the IWT-SBO-STADiUM project No. 80037, and by the IBBT in the context of the IBBT-AdMid project.

REFERENCES

[1] W. Horré, D. Hughes, S. Michiels, and W. Joosen, "Qari: Quality aware software deployment for wireless sensor networks," in *Proceedings of the seventh international conference on Information Technology: New Generations (ITNG 2010)*, April 2010.

[2] C.-C. Han, R. Kumar, R. Shea, and M. Srivastava, "Sensor network software update management: a survey," *International Journal of Network Management*, vol. 15, no. 4, pp. 283–294, 2005.

[3] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. New York, NY, USA: ACM Press, 2004, pp. 81–94.

[4] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 93–104, 2000.

[5] P. Levis, D. Gay, and D. Culler, "Active sensor networks," in *Proceedings of the 2nd USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, May 2005.

[6] R. Balani, C.-C. Han, R. K. Rengaswamy, I. Tsigkogiannis, and M. Srivastava, "Multi-level software reconfiguration for sensor networks," in *ACM Conference on Embedded Systems Software (EMSOFT)*, Oct. 2006.

[7] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*. New York, NY, USA: ACM Press, 2005, pp. 163–176.

[8] Sun Microsystems, "Sun SPOT world," http://www.sunspotworld.com/, Aug. 2008.

[9] Sentilla Corporation, "Sentilla website," http://www.sentilla.com/, Aug. 2008.

[10] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 455–462.

[11] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-time dynamic linking for reprogramming wireless sensor networks," in *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, Nov. 2006.

[12] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, "A generic component model for building systems software," *ACM Transactions on Computer Systems*, vol. 26, no. 1, pp. 1–42, 2008.

[13] D. Hughes, K. Thoelen, W. Horré, N. Matthys, J. Del Cid, S. Michiels, C. Huygens, and W. Joosen, "LooCI: a loosely-coupled component infrastructure for networked embedded systems," in *Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia (MoMM2009)*, Kuala Lumpur, Malaysia, Dec. 2009.

[14] "LooCI google code project," http://code.google.com/p/looci, May 2010.

[15] W. Horré, S. Michiels, W. Joosen, and P. Verbaeten, "Davim: Adaptable middleware for sensor networks," *IEEE Distributed Systems Online*, vol. 9, no. 1, Jan. 2008.

[16] S. Kumar, T. H. Lai, and J. Balogh, "On k-coverage in a mostly sleeping sensor network," *Wireless Networks*, vol. 14, no. 3, pp. 277–294, 2008.

[17] S. Meguerdichian, F. Koushanfar, M. Potkonjak, and M. Srivastava, "Coverage problems in wireless ad-hoc sensor networks," in *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3, 2001, pp. 1380–1387 vol.3.

[18] H. Liu, T. Roeder, K. Walsh, R. Barr, and E. G. Sirer, "Design and implementation of a single system image operating system for ad hoc networks," in *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*. New York, NY, USA: ACM Press, 2005, pp. 149–162.

[19] U. Bischoff and G. Kortuem, "Rulecaster: A macroprogramming system for sensor networks," in *Proceedings of the OOPSLA '06 Workshop on Building Software for Sensor Networks*, Oct. 2006.

[20] A. Awan, S. Jagannathan, and A. Grama, "Macroprogramming heterogeneous sensor networks using cosmos," in *Proceedings of EuroSys 2007*, Mar. 2007.

[21] K. Whitehouse, F. Zhao, and J. Liu, "Semantic streams: A framework for composable semantic interpretation of sensor data." in *EWSN*, 2006, pp. 5–20.

[22] W. B. Heinzelman, A. L. Murphy, H. S. Carvalho, and M. A. Perillo, "Middleware to support sensor network applications." *IEEE Network*, vol. 18, no. 1, pp. 6–14, 2004.

[23] W. Horré, K. Lee, D. Hughes, S. Michiels, and W. Joosen, "A graph based approach to supporting reconfiguration in wireless sensor networks," in *Proceedings of the 1st Workshop on Applications of Graph Theory in Wireless Ad hoc Networks and Sensor Networks*, Dec. 2009.

[24] OASIS, "Open SCA," http://www.oasis-opencsa.org/, Mar. 2010.

[25] A. Schiper, K. Birman, and P. Stephenson, "Lightweight causal and atomic group multicast," *ACM Trans. Comput. Syst.*, vol. 9, no. 3, pp. 272–314, 1991.

[26] Atmel Corporation, "The RZRAVEN 2.4 ghz evaluation and starter kit," http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4291, Apr. 2010.

[27] N. Matthys, C. Huygens, D. Hughes, S. Michiels, and W. Joosen, "Flexible integration of data qualities in wireless sensor networks," in *Proceedings of the 4th international workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*. ACM, Dec. 2009, pp. 31–36.

© 2011 ACADEMY PUBLISHER