A Graph Based Approach to Supporting Software Reconfiguration in Distributed Sensor Network Applications

Danny Hughes [1, 2], Kevin Lee [3], Wouter Horré [2], Sam Michiels [2],
Ka Lok Man [2] and Wouter Joosen [2]

[1] Department of Computer Science and Software Engineering, Xi'an Jiaotong-Liverpool University, 111 Ren Ai Road, Suzhou Industrial Park, Suzhou, Jiangsu, China 215123.
{ daniel.hughes, ka.man }@xjtlu.edu.cn

[2] IBBT-Distrinet, Katholieke Universiteit Leuven, 3001 Leuven, Belgium.
{ danny.hughes, wouter.horre, sam.michiels, wouter.joosen }@cs.kuleuven.be

[3] School of Information Technology, Murdoch University, Murdoch 6150, Western Australia
kevin.lee@ieee.org

Abstract

Considerable research has been performed in applying run-time reconfigurable component models to wireless sensor networks. The capability to dynamically deploy or update software components allows the changing requirements of sensor network applications to be effectively managed, while concrete interface definitions promote re-use. Realizing distributed reconfiguration in wireless sensor networks is complicated by the inherently asynchronous and unreliable nature of sensor network environments. In such an environment, traditional, centralized approaches to achieving distributed reconfiguration are impractical. This paper introduces a graph-based approach to specifying the reconfiguration of software resources that may be distributed across multiple sensor networks. This approach requires application developers to specify only high-level reconfiguration graphs, which are then optimized and enacted in a hierarchical and autonomic manner. We demonstrate and evaluate our approach using a case-study scenario.

1. Introduction

The deployment of large-scale sensor networks remains a time-consuming and expensive task that is infeasible for many single-application scenarios. The difficulties of Wireless Sensor Network (WSN) deployment have in turn driven the development of WSN technologies that promote a shared infrastructure, multi-application paradigm [16]. Furthermore, WSN applications are increasingly making use of resources that may be distributed across multiple networks that may be owned and administered by 3rd parties [17]. This environment presents a number of complex challenges for software deployment.

Reconfigurable component-based approaches hold significant promise for managing the complexity of WSN application development, as they allow for the dynamic deployment of new functionality along with the distributed reconfiguration of existing functionality to meet changing application requirements and environmental conditions. Examples of reconfigurable component models that have been applied in WSN scenarios include OpenCOM [18], RUNES [2], OSGi [19] and LooCI [16]. However, centralized approaches to achieving distributed reconfiguration are unsuitable for modern WSN environments.

To illustrate the benefits of reconfigurable component models in sensor network environments, consider the following motivating example: a WSN is deployed by the company 'STORAGE_CO'. Initially, this WSN infrastructure supports a single application that monitors the location of medical supplies held in a warehouse. After deployment, new regulations are introduced that require that the temperature of medical supplies be monitored at all times. In a component-based system, this requirement can be met by the *dynamic deployment* of a temperature monitoring component, rather than the wholesale replacement of a monolithic application image [3], thus conserving valuable network, storage and power resources. Later, in the same application scenario, STORAGE_CO introduces new equipment into their warehouse which generates periodic interference and thus causes predictable errors in location data. In a component-based system, this problem may be addressed by re-wiring location components via a filter component, which removes this junk data and conserves resources. These examples are illustrated in Figure 1 below.

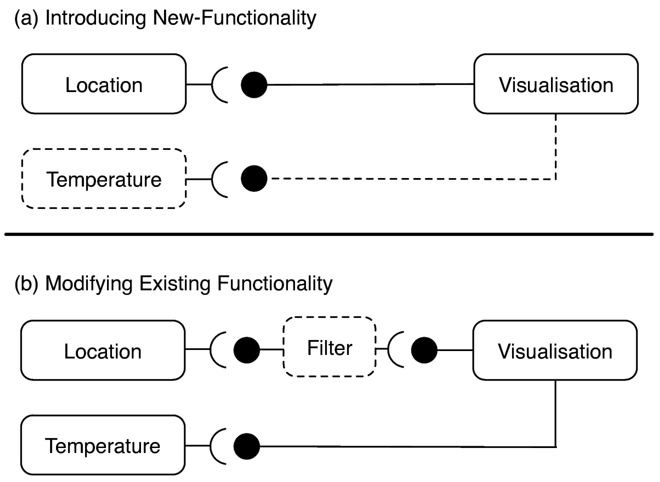

Figure 1: Example Reconfigurations

The simple example discussed above illustrates the benefits of a reconfigurable component-based approach in terms of managing changing application requirements and environmental conditions. However, successfully achieving dynamic reconfiguration is further complicated by the characteristics of WSN:

- ***Asynchronous and Unreliable Communication:*** the low-power network protocols used in WSN tend to be event-based and unreliable. Thus it is impossible to be certain whether a reconfiguration action has been successfully enacted [1] without the imposition of significant networking overhead.
- ***Large Scale*****:** WSN may be comprised of thousands of nodes and thus any reconfiguration approach must scale effectively. Partitioning reconfiguration graphs for delegation to agents close to the nodes being reconfigured thus has the potential to significantly improve scalability.
- ***Multiple Owners:*** applications may require data from third party WSN. In these cases reconfigurations cannot be executed directly, but should be partitioned and distributed to the appropriate organization to be enacted based upon their specific reconfiguration policies, which may control for factors like reconfiguration frequency and provide network-specific optimizations.
- ***Dynamicity*****:** due to the dynamic nature of WSN environments and the limited resources of the sensor nodes, it is practically infeasible for a central entity to have perfect and up-to-date knowledge of the current state of the WSN and their available resources.

In this paper we introduce and evaluate an architecture designed to provide support for the efficient execution of deployment and reconfiguration graphs in WSN environments. The proposed architecture models reconfiguration graphs using approaches inspired by *scientific workflows* [7]. Scalability is ensured through *decentralized and hierarchical* execution of the reconfiguration graph and dynamicity is managed through *adaptive* execution of the workflow close to the point of deployment.

The remainder of this paper is structured as follows: Section 2 provides background on distributed reconfiguration approaches for WSN and adaptive workflow processing. Section 3 discusses WSN reconfiguration as an adaptive workflow problem. Section 4 presents the proposed architecture. Section 5 evaluates the proposed architecture in the context of a case-study scenario. Section 6 discusses directions for future work. Finally, section 7 concludes.

2. Related Work

As argued in the introduction, reconfigurable component models are a good platform to support the dynamic deployment, modification and evolution of distributed applications [4]. This section discusses the state-of-the-art in *component models for WSN*, reviews the state-of-the-art in *distributed reconfiguration* for sensor networks and provides background on how *adaptive workflow techniques* may be applied to the problem of distributed reconfiguration.

2.1 Component Models for WSN

NesC [20] is perhaps the most widely-used component model for WSN and is used to implement the TinyOS operating system [3]. NesC provides an event-driven programming approach together with a static component model. The NesC binding model is based upon statically declared bidirectional component interfaces. Unlike OpenCOM [18], RUNES [2], OSGi [19] and LooCI [16], NesC components cannot be dynamically rewired to support reconfiguration. However, the static programming approach used in NesC allows for whole program analysis and optimization [20], which is advantageous in resource constrained WSN environments.

OpenCOM [18] is a general purpose, run-time reconfigurable component model. While OpenCOM does not target WSN applications specifically, it is used to implement the GridStix [22] sensor network platform. OpenCOM features a compact run-time kernel that supports dynamic, reconfigurable compositions. OpenCOM also offers a higher level of abstraction, known as Component Frameworks (CF) [21] which are used to model interactions between cooperating components. CF may be local or distributed and can be used as a tool to support dynamic reconfiguration.

The RUNES [4] middleware brings OpenCOM functionality to more embedded devices. RUNES has been realized in C and Java, adds a number of introspection API calls to the OpenCOM kernel and has achieved a significantly smaller footprint than OpenCOM [18] consuming less than 20KB of memory [2].

The OSGi component model [19] targets powerful embedded devices such as smart phones and network gateways along with desktop and enterprise computers. OSGi provides a secure execution environment, support for run-time reconfiguration, lifecycle management and various system services. Unfortunately, while OSGi is suitable for powerful embedded devices, the smallest implementation, Concierge [19] consumes more than 80KB, making it unsuitable for very resource constrained devices.

LooCI [16] is a run-time reconfigurable component model for WSN that provides event-based component bindings. These loose event-based bindings better suit the dynamic and unreliable nature of sensor networks. The event-based LooCI binding model also makes it easier to create one-to-many, many-to-one and many-to-many bindings. In the context of reconfiguration, the loosely-coupled binding model of LooCI simplifies consistency management since the dependencies between components are less stringent. The core LooCI runtime consumes less than 21KB of memory [16], making it suitable for even embedded WSN motes.

2.2 Reconfiguration Approaches for WSN

Reconfiguration approaches for WSN may be categorized as monolithic, application-based, script-based or dynamic-component-based. Each of these has distinct software deployment requirements.

- ***Monolithic***: nodes are re-flashed and re-started, replacing all functionality during the update, as in the TinyOS operating system [3]. Monolithic re-flashing is inefficient for small changes as it necessitates the transmission of largely redundant code images.
- ***Application-based***: units of functionality may be deployed at run-time but support is *not* provided for modifying relationships between functional units, as in the Java-based Sun SPOT sensor network platform [6].
- ***Script-based:*** these approaches allow developers to inject lightweight scripts to change the behavior of previously deployed functionality, however, while script-based approaches such as Maté [14] provide an efficient mechanism for tailoring existing functionality, they do not support the injection of new functionality.
- ***Dynamic Component based:*** application compositions, or individual components may be dynamically deployed and the relationships between components may be modified at run-time as in OpenCOM [18].

Deluge [5] is a reliable epidemic code dissemination protocol that is used to support *monolithic* flashing of TinyOS [3] motes. Deluge can only be used to distribute a single code-image to an entire WSN, limiting the variable of mote configurations. Furthermore, using monolithic re-flashing to achieve only small behavioral changes implies a high energy overhead due to unnecessary transmission of redundant functionality.

The *application-based* Sun SPOT [6] platform provides support for unicast distribution of application images to individual sensor motes. In contrast to Deluge this allows for the functionality of each mote to be individually customized. Multiple applications can also co-exist on the same mote, reducing the level of redundant functionality deployed, though this may still be inefficient where only a subset of application functionality needs to be modified. Unfortunately, the unicast distribution mechanism used in the Sun SPOT platform is inefficient for network-wide programming when compared to epidemic protocols such as Deluge [5]. Furthermore, while *application-based* approaches offer advantages over monolithic approaches, relationships between applications are opaque and may not be reconfigured.

Maté [14] is a *script-based approach* that augments the monolithic TinyOS [3] environment with fine-grained reconfiguration. Pre-deployed component functionality may be tailored using lightweight scripts which are distributed in an epidemic manner, similar to that used in Deluge [5]. While script-based approaches allow network-wide functionality to be tailored on a fine-grained level, they do not allow for the injection of new functionality into the network, for example to upgrade application functionality. DAViM [15] removes this limitation by combining the application-based and script-based approaches.

The *dynamic component-based* reconfiguration approach employed in OpenCOM [4], RUNES [2] and LooCI [16] provides rich support for reconfiguration, which may involve deploying or updating components as well as modifying component relationships. In these cases, components may be distributed using a diverse and extensive range of networking protocols including unicast, broadcast and group multicast. While this approach offers a high level of flexibility, the flat nature of code dissemination protocols, wherein code is transmitted from a single source to all nodes requiring reconfiguration is not scalable [4]. We improve upon these approaches using a hierarchical software deployment architecture that allows for decentralized optimization and enforcement of reconfiguration plans. Section 3 discusses how work from the field of scientific work-flows may be applied to address this problem.

2.3 Adaptive Workflow Techniques

A workflow represents a group of interdependent tasks, wherein each task may only execute once all dependent tasks have successfully completed. Each task is held back until the task(s) that it is dependent on has reported completion; tasks with no dependencies can be started immediately when the workflow is executed (commonly the first task in the workflow). When all tasks and their dependencies have completed, the workflow itself is judged to have completed.

A particularly useful abstraction for the domain of WSN reconfiguration is that of grid-based scientific workflows [7]. Wherein, the high level goals of a scientific workflow may be described in an abstract form, as a Directed Acyclic Graph (DAG). Such a workflow includes logical entities such as execution locations, logical data (e.g. a logical entity that will later be mapped to a physical location) and logical transformations (e.g. referring to the transformation of logical data to other logical data).

A two-stage process is used to convert this high-level abstract workflow description to an executable form; this is illustrated in Figure 2. A *first stage* called *Mapping* combines this abstract workflow with specific mappings to locations, files and components. A second stage called *Scheduling* decides when and where to execute workflow tasks on grid resources.

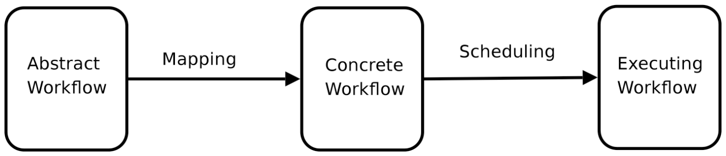

Figure 2: Abstract Workflow Compilation [24]

This flexible process inherently provides support for many adaptive techniques. [24] describes adaptations in both the *mapping* phase and the *scheduling phase* of the compilation process. Mapping adaptations are adaptations where the mapping from the abstract workflow to the concrete workflow changes depending on either changes in the environment or changes in the abstract workflow. Scheduling adaptations involve the alteration of the scheduling policy based on changes in the environment.

Mapping adaptations can involve the changing of the mapping of an abstract node in an abstract workflow to a concrete node in a concrete workflow due to environment changes. It can involve the reduction in the number of tasks in the concrete workflow, splitting tasks thereby increasing the number of tasks or adding additional tasks to perform new functions needed to support the workflows greater goals. Furthermore, as environmental conditions change the data source or sink for tasks in the concrete workflow might become inappropriate and thus adaptations might be necessary.

Scheduling adaptations can involve the movement of a task or service between available execution nodes in the event of environment changes. Depending on the platform support or the workflow properties, scheduling adaptations can also involve the increasing or decreasing of the parallelism of the scheduled task. Further adaptations can involve the pausing, stopping or resubmitting of a concrete task for fault tolerance if incorrect results are being produced or if new intermediate results need to be taken into account.

In addition to adaptations based on the properties of the compilation process, workflow partitioning can be used to introduce support for further adaptations. Partitioning is the process in which the whole workflow is split into multiple smaller workflows that can be compiled and scheduled separately. This partitioning process can be spatial meaning that it could be designed to ensure tasks are grouped together for execution on the same set of nodes or temporal meaning that tasks are grouped together to be compiled at the same time. Partitioning is particularly suitable for supporting fault tolerance; if a group of tasks in a workflow fail the partition that contains them can be resubmitted. Temporal partitioning which delays scheduling for groups of tasks can be useful in an environment where results are appearing from different sources, and such later compilation may lead to a different concrete workflow.

For adaptations to be possible data needs to be collected from sensors (software or hardware) in an ongoing way to allow decisions to be made about when and how to adapt. Such data is commonly in the form of events such as the progress, completion, data consumption rate, data production rate of a task or service; or the availability, load, network usage and memory usage of execution nodes. Architectural support can include splitting the adaptation process into different functional units such as Monitoring, Analysis, Planning and Execution [24], allowing decisions about adaptations to be scripted and structured.

Describing WSN reconfigurations as a workflow allows us to apply the mature abstractions and conceptual mechanisms of the workflow-processing domain to address the problem of WSN reconfiguration. This is explored in detail in the next section.

3. Modeling Reconfigurations as an Adaptive Workflow Problem

Describing reconfiguration as a high level workflow has a number of critical advantages compared to directly calling reconfiguration functionality. These are enumerated below:

- ***Platform and application independence:*** an abstract workflow provides a simple and generic mechanism to specify any form of reconfiguration, for any WSN platform using the DAG abstraction.
- ***Partitioning and decentralized operation:*** the master workflow may be partitioned using the techniques described in section 2.3 and each section of the workflow will be distributed to the entity with responsibility for enacting that reconfiguration.
- ***Workflow Optimization:*** the master workflow may be statically optimized to improve efficiency. Furthermore, as workflow partitions are executed by entities with knowledge of the target deployment location, site-specific optimization may also be performed.
- ***Scalability:*** due to the partitioning and optimization properties described above, a hierarchical workflow based approach to enacting software reconfiguration is inherently more scalable than a centralized approach, as control messages and monitored data do not need to travel between each deployment target and a back-end entity responsible for the whole reconfiguration. Instead the enforcement of each partition can be handled by a reconfiguration entity that is local to the WSN.

The enactment of a workflow occurs as follows: at the *compilation stage*, the compiler ensures that the reconfiguration is enacted in the most efficient way by considering current contextual data. At this point, implicit intermediate tasks may be reified (for example moving components to the location where they should be deployed). Redundant tasks may also be removed from the workflow.

Following compilation of the *reconfiguration graph*, workflow partitioning [9] will be used to split the master reconfiguration graph into smaller *partitions*, which are rendered concrete and distributed to *Action Executors* which have responsibility for achieving reconfigurations in a specific location. Action Executors serve two basic roles. Firstly, they act as units of *virtual synchrony* [11], arbitrating the success or failure of reconfiguration actions based upon locally gathered contextual data. Secondly, based upon local context they may perform site-specific optimization of the partition for which they have responsibility, for example by removing redundant deployment operations for software components that are already deployed at the target location.

During *execution* of the reconfiguration graph, context-based adaptation occurs at two levels. At the network level of the Action Executor, contextual data relevant to a specific network is used to arbitrate the success or failure of reconfiguration actions. For example, based upon previous performance, the Action Executor may modify the time-out period it applies before judging that a reconfiguration message has not been received. Secondly, each Action Executor will select appropriate methods for enacting a reconfiguration on the specific software and hardware technologies used in the WSN.

At the global level, the master workflow executor is informed of reconfiguration progress in terms of the success or failure of each concrete action (i.e. partition), and based upon the success or failure of reconfiguration actions reported by Action Executors, the Compiler may adapt in a number of ways including: Selecting alternative reconfiguration targets, recompiling the graph using new context data or introducing fault tolerance and redundancy into the deployment [10].

4. System Architecture

This section describes an architecture designed to support the adaptive enactment of reconfiguration graphs in WSN environments as described in Section 3. This architecture is illustrated in Figure 3. The reconfiguration process is as follows (the reader should refer to Figure 3). The high-level, abstract reconfiguration graph is compiled to a concrete change graph by the top-level *Compiler*. A *Partitioner* then splits the concrete graph into concrete partitions, which are then transferred to *Action Executers* at the edge of the network, which each have responsibility for enacting changes at a specific location. The functionality and operation of the Compiler, Partitioner and Action Executor are described in detail in section 4.1 to 4.3 respectively.

4.1 Compiler

The *Compiler* takes a high-level, abstract change graph, specified as a Directed Acyclic Graph and encoding in XML (XML-DAG). This abstract graph is then rendered 'concrete' by the Compiler: a process in which each abstract action is reified into executable commands, as described in [24].

- ***Reification of implicit operations***: abstract operations may actually subsume a number of implicit steps. For example, 'REMOVE_COMPONENT' may actually subsume the following implicit operations: 'MAKE_QUIESCENT' and 'STOP' before the component can be safely removed.
- ***Conversion of abstract targets to concrete targets:*** abstract targets specified in the graph for example 'WAREHOUSE_A' may be reified to a concrete location identified by a network address or other unique identifier. Each concrete location should denote a valid action executor.
- ***Removal of redundant operations***: redundant operations may also be removed at the compilation stage based upon static inspection and optimization of the composition graph.

4.2 Partitioner

The *Partitioner* has responsibility for the spatial and temporal partitioning of the concrete reconfiguration graph:

- ***Spatial Partitioning***: spatial partitioning of the concrete graph creates a number of sub-graphs, one for each deployment target (i.e. Action Executor), for efficient parallelized implementation of the reconfiguration graph.
- ***Temporal Partitioning:*** temporal partitioning may occur within a single spatial graph, or across multiple spatial graphs which are to be executed at distinct target locations. During temporal partitioning, sub-graphs are partitioned into a number of distinct steps, based upon ordering constraints encoded in the graphs. The *Partitioner* will then distribute each temporal partition for execution on the associated Action Executer as the constraints for execution are met. For example,

a ‘START_COMPONENT’ command may only be issued after the associated ‘DEPLOY_COMPONENT’ command has been completed.

4.3 Action Executer

The *Action Executor* builds on the approach described in [9], monitoring the state of the WSN using context-sensors (lightweight software components that provide status information), which are used to inform the adaptive behavior of the Action Executor, which may include:

- ***Target selection***: even when concrete, a target may specify that reconfiguration should be enacted on one of multiple physical nodes. Where this is the case, contextual information such as available battery level may be used to select the most appropriate node.
- ***Redundant action removal:*** concrete partitions may include unnecessary operations (e.g. deploying a component to a location where an equivalent component is already deployed). The Action Executor will inspect the current network configuration, and where redundant operations are detected, they will not be enacted.
- ***Providing Fault tolerance***: based upon previously observed failure rates, an Action Executor may choose to repeatedly apply reconfiguration actions in order to ensure they are successful.
- ***Enforcing Reconfiguration Policies***: the owner of each Action Executor will specify a reconfiguration policy that will restrict the reconfigurations that 3rd parties may perform on their WSN infrastructure.

4.4 Recompilation and Repartitioning

Where the execution of any concrete partition fails, this will be reported to the Compiler, which will initialize a recompilation of the remaining reconfiguration tasks, incorporating the context data provided by the failure into the compilation process. For example, a recompilation that occurs in response to the failure of a concrete partition at location X may not use this location when abstract target locations are reified in the recompilation. Further optimization may also be performed at this stage.

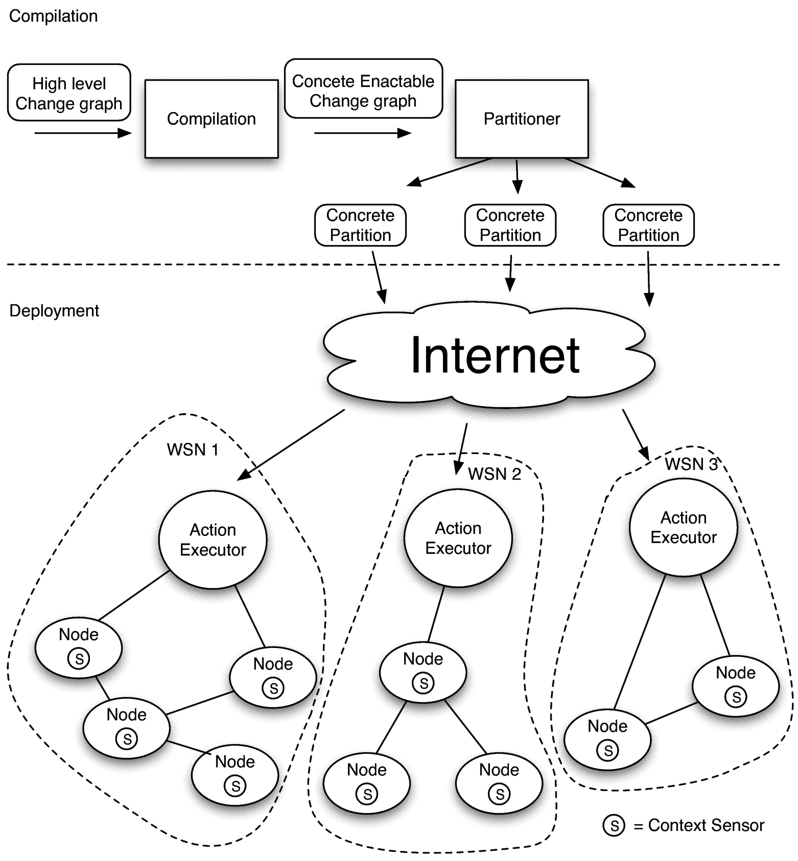

Figure 3: System Architecture

5. Case Study Evaluation

This section illustrates the appropriateness of the previously introduced architecture for supporting reconfiguration of WSN through a case-study scenario. This case-study shows how adaptive processing of the reconfiguration graph can be used to: *i.)* provide fault tolerance, *ii.)* eliminate redundant reconfiguration actions and *iii.)* modify the relationship between existing components. Section 5.1 provides details of the case-study scenario. Section 5.2 discusses static composition optimization. Section 5.3 shows how adaptive graph processing at the global level can provide fault tolerance. Section 5.4 shows how adaptive graph processing by Action Executors can remove redundant reconfiguration actions. Finally, section 5.5 shows how this architecture can be used to modify the relationships between deployed components. In each case, the XML reconfiguration graph is provided and the reconfiguration process is described in detail.

5.1 Case-Study Scenario

This appropriateness of our reconfiguration architecture will be illustrated through a stock tracking scenario. In this scenario, STORAGE_CO deploys sensor nodes in each of the packages that they are contracted to store. Packages are stored on pallets, which may be inspected at any time by customs officials equipped with mobile devices. Each pallet features one gateway node which runs an action executor and is responsible for all sensor nodes stored in the associated packages. Regulations state that at least half of all pallets stored in the warehouse should report environmental conditions when inspected.

5.2 Static Optimization of the Reconfiguration Graph

Prior to the concretization and partitioning of any change graph, it will first be subject to static optimization. This may involve (i.) the removal of redundant reconfiguration options and (ii.) component substitution, where a superior component than that specified in the reconfiguration graph exists in the repository. Here the definition of 'superior' is dependent upon the reconfiguration scenario. For example, in a bandwidth and power constrained scenario, a functionally-equivalent component that is smaller may be considered superior.

5.3 Providing Fault Tolerance

```
<?xml version="1.0" encoding="UTF-8"?>
       <reconfiguration-graph>
              <deploy id="1">
                     <component>PACKAGE_STATE</component>
                     <location>PALLET_3</location>
                     <location>PALLET_4</location>
                     </deploy>
              <connect id="2">
                     <origin>PALLET_3</origin>
                     <dest>BACK_END</dest>
                     <component>PACKAGE_STATE</component>
                     <component>STORAGE_MONITOR</component>
                     </connect>
              <connect id="3">
                    <origin>PALLET_4</origin>
                    <dest>BACK_END</dest>
                    <component>PACKAGE_STATE</component>
                    <component>STORAGE_MONITOR<component>
                     </connect>
              <child ref="2">
                     <parent id="1">
              </child>
              <child ref="3">
                     <parent id="1">
              </child>
       </reconfiguration-graph>
```

Listing 1: Reconfiguration Graph Illustrating Fault Tolerance

The first stage in this scenario is for STORAGE_CO to *deploy* a ‘PACKAGE_STATE’ monitoring component to a subset of pallets in the warehouse (3 and 4). The XML reconfiguration graph for this operation is shown above.

Upon execution of the reconfiguration graph, the abstract XML will be reified to a set of concrete actions. Specifically, the abstract location for ‘PALLET_X’ will be converted to gateway addresses and the abstract concept *deploy* will be converted to a platform-specific deployment action. The concrete graph will then be partitioned and deployed to appropriate Action Executors. Where an Action Executor reports failure, the global reconfiguration executor will adapt to this contextual information. Thus, the original high level reconfiguration graph will be recompiled, taking into account what actions have been performed successfully and producing a concrete graph that only performs the operations that previously failed. Following successful component deployment the components will then be bound to the back-end package monitoring software of STORAGE_CO.

5.4 Removing Redundant Reconfigurations

During the deployment process outlined above, the Action Executor may also remove redundant actions in the concrete reconfiguration partition. Before each reconfiguration action is enacted, the Action Executor will inspect the specified location and check whether the current state matches the successful outcome of the reconfiguration action to be executed. Where this is the case, the redundant reconfiguration action will simply be omitted. In this specific instance, redundancy may be removed where a matching component (i.e. PACKAGE_STATE) is found to exist, and thus the component will not be re-deployed, conserving valuable resources.

5.5 Modifying Component Relationships

When a customs official with a mobile device arrives to inspect the packages stored by STORAGE_CO, a new reconfiguration graph will be submitted to connect PACKAGE_STATE components within range of the device to the MOBILE_MONITOR component running on the customs official’s mobile device.

```
<?xml version="1.0" encoding="UTF-8"?>
       <reconfiguration-graph>
              <connect id="1">
                     <origin>GLOBAL</origin>
                     <dest>MOBILE_USR</dest>
                     <component>MOBILE_MONITOR<component>
                     <component>PACKAGE_STATE</component>
              </connect>
       </reconfiguration-graph>
```

Listing 2: Reconfiguration Graph Illustrating Component Rewiring

As before, upon execution of the reconfiguration graph, the abstract reconfiguration graph will be reified to a set of concrete actions and dispatched to the appropriate Action Executor to be enacted. New reconfiguration graphs will be dispatched by the custom official’s device as it comes within range of new packages / sensor motes in order to deal with mobility.

5.6 Discussion

While the case-study we have presented is simple, we believe that it illustrates the benefits of using a hierarchical, graph-based approach to achieving reconfiguration in WSN environments. In the above scenario, adaptive graph execution was used to provide fault tolerance, while local contextual data was applied to remove redundant reconfiguration operations. We expect that further benefits will be evident in multi-owner WSN scenarios, where our decentralized design will allow each WSN administrator to specify an appropriate reconfiguration policy.

The presented approach has concrete benefits in terms of relieving developers from the complexities of software deployment in unreliable network environments and, moreover, our XML reconfiguration specification language provides a simple, yet powerful mechanism for developers to specify their desired reconfiguration actions.

Another critical advantage of our approach is that it allows for a clean separation of concerns between the *planning* of software deployments and their *realization*. The former should be based upon high level concerns and platform independent, while the latter should be tightly coupled to the specific WSN platform on which deployment is being enacted, such that contextual data can be efficiently exploited.

We also intend to explore the way in which more flexible notions of 'deployment quality' may be embedded into reconfiguration operations. This is particularly important in dynamic and multi-user sensor network scenarios where Action Executors may be expected to simultaneously honor the differing requirements of distinct applications. To accomplish this, we will draw on the techniques provided by the Quality Aware Reconfiguration Infrastructure (QARI) [25].

It is also important to note that, while our case study focused upon a component based reconfiguration approach, our hierarchical, graph-based approach could equally be applied to scripted, application-based or even monolithic reconfiguration approaches.

6. Future Work

In the short term, our future work will focus upon testing the architecture presented in this paper in a rich, multi-user WSN scenario, which will allow us to evaluate the performance of our architecture in an application scenario which allows for separation of infrastructure provision and application development. We will then quantitatively evaluate the potential benefits of this approach, including: static optimization, fault tolerance and contextual optimization by Action Executors in the context of this scenario.

Critically, the approach that we propose and evaluate in this paper promotes a separation of concerns between application development and infrastructure provision, which is consistent with current trends in the field of WSN [16] [17] and allows the creation of large scale sensor network applications, which may span multiple sensor networks and organization boundaries.

In the longer term, we intend to explore the extent to which context-awareness can be used to automatically optimize reconfiguration graphs (for example to exploit currently deployed components), to serve predicted future needs and minimize the overhead of reconfiguration operations in terms of bandwidth and power consumption.

7. Conclusions

This paper has advocated for the use of runtime reconfigurable component models to manage the dynamism of WSN environments. We presented a hierarchical, adaptive, graph-based approach to enacting reconfiguration. This approach draws on existing techniques from the dynamic work-flow processing domains. The appropriateness of this approach was illustrated through a detailed WSN case-study involving diverse reconfiguration actions.

In summation, we believe that an adaptive, graph-based approach to enacting reconfiguration in WSN holds great potential for lowering the burden on application developers, while allowing for optimization of reconfiguration graphs.

8. Acknowledgments

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, and by the Research Fund, K.U.Leuven.

9. References

[1] Coulouris G., Dollimore J. and Kindberg T., Distributed Systems: Concepts and Design, Fourth Edition, Addison-Wesley 2005.

[2] 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 proc. of the 5th Annual IEEE International Conference on Pervasive Computing and Comunications (PERCOM'07), White Plains, New York, 19-23 March 2007, pp. 69–78.

[3] Hill J., Szewczyk R., Woo A., Hollar S., Culler D., Pister K., System Architecture Directions for Networked Sensors, in ACM SIGPLAN, Vol. 35, No. 11, November 2000, pp. 93-104.

[4] Grace P., Coulson G., Blair G., Porter B., Hughes D., Dynamic Reconfiguration in Sensor Middleware, in the proceedings of the 1st International Workshop on Middleware for Sensor Networks (MidSens'06), Melbourne, Australia, November 2006, pp. 1 – 6.

[5] Hui J. W., Culler D., The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In proc. of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys'04), Baltimore, Maryland, USA, November 2004, pp. 81-94.

[6] Sun Microsystems, Small Programmable Object Technology, "Inspiring Java developers to create a whole new breed of devices and technologies - and accelerating the growth of the 'Internet of Things'", available online at: http://www.sunspotworld.com/vision.html

[7] Deelman E., Singh G., Sa M., Blythe J., Gil Y., Kesselman C., Mehta G., Karan V., Berriman G., Good J., Laity A., Jacob J., and Katz D., Pegasus: A framework for mapping complex scientific workflows onto distributed systems, in Scientific Programming, Vol. 13, No. 3, 2005, pp. 219–237.

[8] Gurmeet S., Kesselman C., Deelman E., Optimizing Grid-Based Workflow Execution, in Journal of Grid Computing, Vol. 3, No. 3-4, 2005, pp. 201-219.

[9] Nichols J., Demirkan H., Goul M., Autonomic Workflow Execution in the Grid, in IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews, Vol. 6, No. 3, May 2006, pp. 353-364.

[10] Lee K., Paton N. W., Sakellariou R., Deelman E., Fernandes A. A. A., Mehta G., Adaptive Workflow Processing and Execution in Pegasus, to appear in Concurrency and Computation: Practice and Experience, 2009.

[11] Schiper A., Birman K., Stephenson P., Lightweight causal and atomic group multicast, ACM Transactions on Computer Systems, Vol. 9, No. 3, 1991, pp. 272-314

[12] Sun Microsystems, Java ME - the Most Ubiquitous Application Platform for Mobile Devices, available online at: http://java.sun.com/javame/index.jsp

[13] Dunkels A., Grönvall B., Voigt T., Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors, in proc. of 29th IEEE International Conference on Local Computer Networks (LCN'04), Tampa, FL, USA, November 2004, pp. 455 – 462.

[14] Levis, P.; Gay, D. & Culler, D., Active Sensor Networks, in proc. of the 2nd USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI'05), Boston, Massachusetts, USA, May 2005, pp. 343 – 356.

[15] Horré W., Michiels S., Joosen W., Verbaeten P., DAVIM: Adaptable Middleware for Sensor Networks, IEEE Distributed Systems Online, 2008, Vol. 9, No. 1

[16] Hughes D., Thoelen K., Horré W., Matthys N., Michiels S., Huygens C., Joosen W., LooCI: A Loosely-coupled Component Infrastructure for Networked Embedded Systems, in the proceedings of the 7th International Conference on Advances in Mobile Computing & Multimedia (MoMM'09), December 200.9

[17] Huygens C., Joosen W., Federated and Shared use of Sensor Networks through Security Middleware, in the proceedings of the 6th International Conference on Information Technology: New Generations, Las Vegas, Nevada, USA, April 27-29, 2009.

[18] Coulson G., Blair G., Grace P., Taiani F., Joolia A., Lee K., Ueyama J. and Sivaharan T, A Generic Component Model for Building Systems Software, in ACM Transactions on Computer Systems, Vol. 26, No. 1, Feb 2008.

[19] Rellermeyer J., Alonso G., Concierge: A Service Platform for Resource-Constrained Devices, in ACM SIGOPS Operating Systems Review, Vol. 41, No. 3, June 2007, pp. 245 – 258

[20] Gay D., Levis P., Von Behren R., Welsh M., Brewer E., Culler D., The NesC Language: A Holistic Approach to Networked Embedded Systems, in Proc. of the conference on Programming Language Design and Implementation, ACM SIGPLAN 2003, San Diego, California, USA, pp. 1 – 11.

[21] Grace P., Coulson G., Blair G., Porter B., Hughes D., Dynamic Reconfiguration in Sensor Middleware, in Proc. of the 1st International Workshop on Middleware for Sensor Networks (MidSens'06), Melbourne, Australia, November 2006, pp. 1 – 6.

[22] Hughes D., Greenwood P., Coulson G., Blair G., Pappenberger F., Smith P., Beven K., An Experiment with Reflective Middleware to Support Grid-based Flood Monitoring, in Wiley Inter-Science Journal on Concurrency and Computation: Practice and Experience, vol. 20, no 11, November 2007, pp 1303-1316.

[23] IONA et al., Service Component Architecture: Building Systems using a Service Oriented Architecture: www.iona.com/devcenter/sca/SCA_White_Paper1_09.pdf

[24] Lee K., Sakellariou R., Paton N. W., Fernandes A.A.A., Workflow Adaptation as an Autonomic Computing Problem, 2nd Workshop on Workflows in Support of Large-Scale Science (Works 07), In Proceedings of HPDC 2007, Monterey Bay California, June 27-29 2007

[25] Horré W., Hughes D., Michiels S., Joosen W., QARI: Quality Aware Software Deployment for Wireless Sensor Networks, accepted at the 7th International Conference on Information Technology : New Generations (ITNG'10).