

Dynamic Software Reconfiguration in Sensor Networks^{*}

Sachin Kogekar, Sandeep Neema, Xenofon Koutsoukos

Institute for Software Integrated Systems (ISIS)

Department of Electrical Engineering and Computer Science

Vanderbilt University

{Sachin.Kogekar,Sandeep.Neema,Xenofon.Koutsoukos@vanderbilt.edu}

Abstract

Reconfiguration and self-adaptation are vital capabilities of sensor networks and networked embedded systems that are required to operate in dynamic environments. This paper presents an approach for software reconfiguration based on exploration of the design space of the application. The design space is represented by formally modeling all the software components, their alternative implementations and their interactions. Reconfiguration is triggered by monitoring the system and is performed by transitioning to a new configuration that satisfies the system constraints. The approach is demonstrated using experimental results for a representative tracking application.

1. Introduction

Wireless sensor networks consist of a number of nodes spread across a geographical area, deployed in an ad hoc manner. The nodes are equipped with sensors, wireless communication, occasionally actuators, and computation capabilities. Applications are characterized by dynamic functional and performance requirements due to the uncertainty and variability of the environment.

Reconfiguration and self-adaptation are vital capabilities of sensor networks that are required to operate in dynamic environments. Dynamically adaptive software consists of tasks that detect system changes, reflect on the event occurrences, and adapt to the new operating conditions. Runtime technologies that allow software to evolve as system requirements change are critical because they enable such systems to operate under multiple conditions.

One type of reconfiguration is already implemented in wireless sensor networks using dynamic and ad hoc routing. We are concerned with a richer form of reconfiguration which allows

changing the functionality of individual nodes in the network in response to changes in the environment and/or usage. More importantly, we argue that the reconfiguration architecture should allow deriving a suitable configuration on-line as it would be infeasible to pre-compute all operating conditions, and thus all viable system configurations.

This paper presents an approach for constraint-based dynamic software reconfiguration in sensor networks. We have prototyped a software architecture using an 8-node sensor network and we present experimental results to evaluate the approach. The key components of our approach include: (i) A domain-specific modeling environment, designated the “Sensor Network Reconfiguration Architecture Modeling Language” (SNRAMoLa), instantiated in the meta-programmable generic modeling environment GME [6], (ii) A constraint-based design-space exploration tool, designated “DESERT”, that allows derivation of feasible configurations, and (iii) A suite of runtime components and services that allow monitoring the operating conditions and enacting the reconfiguration instructions.

Early results of this work have been presented in [7] where we investigated dynamic software reconfiguration for sensor networks based on the Berkeley MICA motes and TinyOS [5]. Developing the reconfiguration infrastructure on the motes was not possible because of the severe hardware constraints and the static nature of TinyOS and the approach was demonstrated only using simulation results. This paper focuses on the implementation of the reconfiguration approach for a sensor network consisting of 8 Linux-based sensor nodes equipped with cameras and communicating via 802.11b.

The paper is organized as follows. Section 2 discusses related work. The reconfiguration architecture is presented in Section 3. Section 4 presents the modeling environment and Section 5

^{*} This work is supported in part by NSF Career grant CNS-0347440 and a grant from Xerox Corp.

briefly discusses design space exploration. The reconfiguration infrastructure is presented in Section 6 and a case study using a simple tracking application is presented in Section 7. Conclusions are discussed in Section 8.

2. Related work

A generic architecture for adaptation in pervasive networks using a client/server networking example is presented in [3]. The work in [2] presents a lightweight infrastructure for managing dynamic reconfiguration in component-based, distributed software systems. An approach to carry out reconfiguration for fault tolerance is suggested in [4]. A framework for supporting the construction and dynamic reconfiguration of distributed multimedia applications is presented in [9].

The approaches summarized above focus primarily on the constructs and mechanisms for enacting a reconfiguration. However, there is very little emphasis on determining what the next configuration should be. Most of the approaches pre-compute the configurations and at runtime do a simple table lookup to decide the next configuration. In sensor network applications the number of potential configurations are simply too many to pre-compute. Our emphasis therefore is on scalable constraint-based techniques for determining network configuration during runtime.

3. Reconfiguration architecture

Central to our approach is the ability to explicitly represent and manipulate the *design space* of the embedded application, which we also term the *operation space* when embedded in the running application. We define reconfiguration as the process of transitioning from one point in the operation space to another. This space is captured by formally modeling all the software components, their alternative implementations, and their interactions that together constitute an application. The applications are modeled using the Asynchronous Data Flow model of computation [8]. Components interact by exchanging data through input and output ports, which constitute the input and output interfaces of components. System requirements are expressed as formal constraints on operational parameters such as power consumption, latency, accuracy, and other Quality-of-Service (QoS) properties that are monitored at runtime. Object Constraint Language (OCL) [11], a standard-based declarative language is used for expressing constraints.

Determining the new configuration is a search problem in the operation space. The exploration of the operation space is a challenging problem since it must be performed within stringent time bounds and resource constraints. An efficient approach for performing this search is based on (1) parameterized constraints captured in the embedded models and (2) online constraint solving using a combination of symbolic constraint satisfaction and linear programming. Once a new configuration that satisfies all the constraints is found, the reconfiguration can be accomplished by online software synthesis targeting either an interpreted language or a command interface.

Reconfiguration thus involves two major tasks: (i) finding the new configuration and (ii) switching or reconfiguring the components that are actually executing on the individual sensors. These tasks are performed by architectural components, as depicted in Figure 1. The first task is performed by a component that runs on the base station while the second task is performed on individual sensors by specialized switching components.

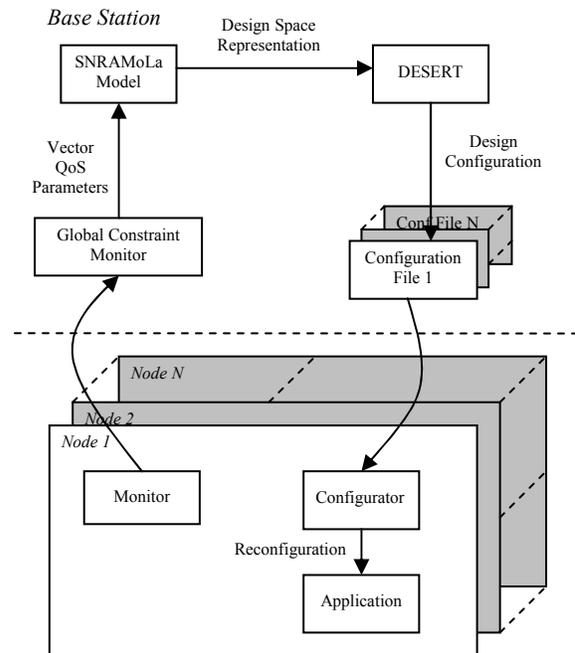


Figure 1. Reconfiguration architecture

During design time, the entire application is modeled using SNRAMoLa. The application model captures the system design space, constraints, and QoS attributes in the models. During operation, the Global Constraint Monitor (GCM), executing on the base station, monitors the sensor nodes using the Monitor components executing on individual nodes. The collated monitor data is used to update the QoS

attributes captured in the application models. The GCM is also tasked with characterizing the QoS parameter changes, and any change bigger than predetermined QoS thresholds causes the GCM to initiate the reconfiguration process.

The first task of determining a new configuration is performed by invoking the design exploration tool DESERT [10]. DESERT prunes the design space retaining only the configurations that are valid with respect to the constraints. The output of DESERT is used to generate a set of configuration files, one for each sensor.

The next task is performed by dispatching the new configuration files to the nodes over the ad hoc wireless network. A Configurator component on individual nodes executes the reconfiguration instructions to stop, rewire and/or start active and dormant application components. In our current architecture the software for alternate components is already present on the sensor nodes to lower the reconfiguration latency, however this does not preclude downloading new component binary execution code on to the sensors.

4. Modeling reconfigurable applications

We have developed SNRAMoLa, a graphical modeling language, to model the design space of the sensor network application. In the meta-programmable modeling environment GME [6], modeling languages are defined with UML class-diagrams. Commonly referred to as a “meta-model”, these capture the abstract and concrete syntax of a modeling language, while structural semantics are captured with OCL constraints. Note that this use of OCL constraints is different from the one mentioned earlier to express the operational constraints.

In addition to the syntax, and structural semantics, the operational semantics of a modeling language are indicated by selecting a model of computation. In SNRAMoLa, we choose the Asynchronous Data Flow (ASDF) model of computation [8]. In ASDF, algorithms are described as directed graphs where the nodes represent computations (or functions) and the arcs represent data paths. Any node can *fire* (perform its computation) whenever input data is available on its incoming arcs. A node with no input arcs may fire at any time. This implies that many nodes may fire simultaneously, and hence represent concurrency.

SNRAMoLa enables the user to represent sensor network applications in the form of a dataflow graph. The application graph is composed of components that exchange data through ports. Figure 2 shows the meta-model of SNRAMoLa. The core concepts in the SNRAMoLa are *Component-s*, *InPort-s*, *OutPort-s*,

DataFlow connection-s, *Choice-s* and *Condition-s*. The *Sensor* and *SensorFolder* objects contain the application graph, which is composed of the core objects. The *ComponentsFolder* object contains all the non-reconfigurable Component objects. Non-reconfigurable Component objects cannot be replaced by any other components in the application by DESERT during the reconfiguration process and are always included in the application configuration.

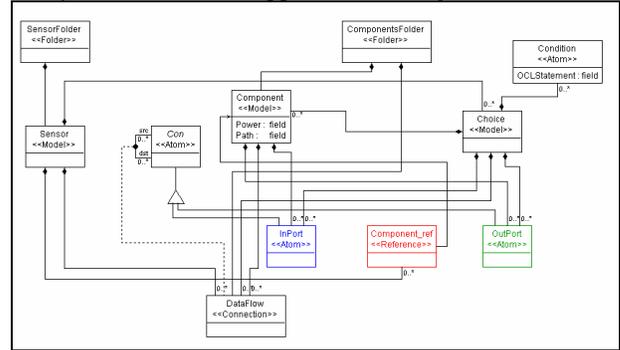


Figure 2. SNRAMoLa meta-model

Each SNRAMoLa model contains exactly one *SensorFolder* object. The *SensorFolder* acts as a container for all the *Sensor* objects, which model sensor node devices. Each application model also contains exactly one *ComponentsFolder* object. This object contains all the non-reconfigurable *Component* objects that are included in the application. The components are then only referred in the actual application graph built in the *Sensor* objects.

Separate *Sensor* objects for each sensor are declared inside the *SensorFolder* object. *Sensor* objects represent the actual sensor nodes in the sensor network. This enables the user to model different applications (applications composed of different components) for different sensors in the network. The graphs for the application executing on each sensor are then built inside these *Sensor* objects. The application graph is composed of the core *Component*, *Choice* and *Condition* objects.

A *Component* object represents a separate process in the application executing on the sensor devices. Components may contain *InPort(s)* and *OutPort(s)* if they exchange data with other Components. A Component is reconfigurable if it can be replaced by another Component in the application graph during the reconfiguration. Non-reconfigurable components are declared in the *ComponentsFolder* and referenced in the application graph built on individual *Sensor* objects while reconfigurable components are declared in the *Choice* objects, which form containers for alternative Components. Each Component also has an attribute called ‘Path’ which identifies the physical path of the executable that is invoked when starting

the Component. This attribute is used by the Configurator component on the individual nodes to execute the corresponding process represented by that Component.

An *InPort* object represents an input port of a Component object. It is declared inside a Component object and used to accept data from another Component object. A Component object can have any number of InPort objects but each InPort object can be connected to at most one corresponding OutPort object declared inside another Component or Choice object using the DataFlow connection object.

An *Outport* object represents an output port of a Component object. It is also declared inside a Component or Choice object and used to send data to another Component object. A component can have any number of OutPort objects but each OutPort object can be connected to at most one corresponding InPort object, declared inside another Component object, using the DataFlow connection object.

A *DataFlow* object is a connection object that links an output port of a Component object represented by an OutPort object with an input port another Component object represented by an InPort object. It models the asynchronous flow of data from one application component to another. The DataFlow object along with the InPort and OutPort objects is implemented in the reconfiguration software infrastructure using shared memory.

As the name suggests, a *Choice* object facilitates the user to model reconfigurable or mutually replaceable Component objects in the application graph. At any given instance, only one process from the collection of processes represented by the Component objects declared in a given Choice object actually executes in an application. A Choice object also contains a Condition object, which specifies the condition expressed in OCL in its Expression attribute. During the reconfiguration process, DESERT evaluates all the constraints modeled as Condition objects over all the Component objects declared in the respective Choice objects and selects only one Component object to be included in the final application graph from each Choice object. The selection is based on the value of the QoS attributes of the Component objects.

The SNRAMoLa paradigm enables the user to model complex component based sensor network applications in an intuitive manner. The communication links between various components are clearly expressed using the InPort, OutPort and DataFlow objects. The paradigm enables the user to model components that can be replaced by others during runtime along with the constraints that govern the selection of the appropriate components from the

collection of alternatives. The user can visualize the applications executing on individual sensor nodes along with all the active and passive components of the application and maintain different versions of the application on individual sensors if needed. An example of a reconfigurable application model in SNRAMoLa is described in Section 7.

5. Design space exploration

This section briefly elaborates upon the design space representation and exploration techniques, as implemented in the tool DESERT [10].

Formally, a design space is a set and can be symbolically formulated as follows. A configuration is a particular selection of choices in the space. Let $Configs(d)$ be the set of all configurations that include an element d , and $\chi(d)$ be the set of children of d . Also let D_j be the set of values of property j , and let $P(l)$ be the set of properties in a leaf element l . Then, the set of possible instantiations $PS(l)$ of the leaf element l can be defined as:

$$PS(l) = \prod_j^{P(l)} D_j \quad (1)$$

The set of configurations can be constructed recursively, depending on element decomposition, as follows:

$$Configs(d) = \begin{cases} PS(d) & \text{LEAF} \\ \prod_{x \in \chi(d)} Configs(x) & \text{AND} \\ \bigcup_{x \in \chi(d)} Configs(x) & \text{OR} \end{cases} \quad (2)$$

Let, \mathfrak{R}_k be the root element of the k -th space, then $Configs(\mathfrak{R}_k)$ is the set of all configurations in the k -th space. The aggregate design space can now be defined as:

$$DS = \prod_k Configs(\mathfrak{R}_k) \quad (3)$$

Design-spaces can be combinatorially large, rendering all enumerative techniques infeasible. Fortunately, the structuring of design space as sets, lends itself suitable to application of symbolic techniques. We employ a binary encoding for the elements of the design space (see [10] for details), and construct the entire space symbolically as Boolean functions represented with Ordered Binary Decision Diagrams [1], a powerful and scalable tool for manipulation of Boolean functions.

There are two basic categories of structural constraints that DESERT can compute efficiently. *Compatibility and Inter-space constraints* specify relations among subspaces in the overall design space expressing semantic compatibility between different elements. Symbolically, these constraints can be represented as a Boolean expression over the Boolean representation of the elements of the design-space. *Property constraints* specify bounds on the composite properties of elements in the composed system. The important challenge for the property constraints are that they are derived from structural characteristics of designs. A combination of additive, min, and max type of composition function can be used expressed such constraints in DESERT.

The primary advantage of the symbolic design space pruning approach is that it is exhaustive, i.e. the pruned space includes all of the designs which meet the applied design constraints. In our approach, the first amongst all the valid configurations generated by DESERT is selected. A significantly simpler, but still useful alternative approach to design space pruning could be to find a single design configuration (not all), which satisfies the selected design constraints.

6. Reconfiguration infrastructure

Once the application is deployed the tasks of design space exploration, communication of the configuration to the sensors, monitoring the sensors and updating QoS parameters in the models are performed in a cyclical manner. During the reconfiguration process, the application model is converted to a format acceptable to DESERT by the SNRAMoLa to DESERT Interpreter (for details of this mapping see [7]). The converted data is fed to DESERT as an XML file. DESERT applies the constraints present in the model and generates another XML file, which enumerates the design configurations in the pruned space. The DESERT to Configurator interpreter then generates a configuration file for each Sensor object present in the SNRAMoLa model, for the selected design configuration.

The *DESERT to Configurator Interpreter* reads from the SNRAMoLa model file and the DESERT output file (XML file) and creates individual configuration files for each Sensor object declared in the SNRAMoLa model. These configuration files are identified by the sensor nodes which are identified by the Sensor objects in the SNRAMoLa model. The configuration files are then physically transported to the nodes over the wireless network.

The *Configurator* is the most important component of the software reconfiguration infrastructure. A copy of the Configurator executes on all the sensor devices. The Configurator implements the reconfiguration infrastructure by maintaining two link-list data structures and a memory ID counter. The *Processes link-list* stores information about all the processes that are currently executing on the node. When a new component is added in the configuration file, the Configurator adds a new process in the list before executing it. The *Links link-list* stores all the information about the shared memory that is used to pass data between two processes. The Configurator creates and maintains connections between the processes (using shared memory) so that the source and destination processes can exchange information.

During initialization, the Configurator opens a socket to listen for incoming signals from the base station. After initialization, the Configurator goes in an infinite loop where it continues to listen for new messages on the open port coming from the base station. Upon receiving a message, it performs reconfiguration activities and then goes back to listening for new messages.

The *Monitor* components execute on each node and monitor QoS parameters of itself and its immediate neighbors. The GCM executes on the base station and receives messages with the QoS parameters of each node. The GCM updates the QoS parameters in the SNRAMoLa models of the application and then invokes the reconfiguration process on the base station. The configuration files are then sent to the nodes where the actual reconfiguration takes place.

7. Case study

Our sensor network test-bed consists of eight Red Hat Linux OpenBrick-E wireless devices [12] and a Windows XP base station. The OpenBrick-E has a small form factor and it includes a USB-based 802.11b wireless LAN with a 2 dbi antenna. Each node is equipped with a Logitech QuickCam Pro 4000 webcam. The base-station is connected to one sensor node through a wired 802.3 LAN connection and is used to carry out computation intensive tasks in the reconfiguration process. The eight nodes are configured to form a private ad hoc wireless network as shown in Figure 3. The reconfiguration software and the applications generate UDP packets and route them to the destination using IP. An implementation of the Ad hoc On Demand Distance Vector (AODV) provided by NIST [13] is used for routing.

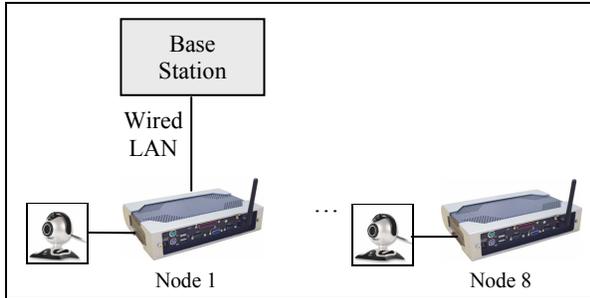


Figure 3. Sensor network setup

The distributed aislemonitor application is designed to perform one-dimensional tracking of people walking in an aisle. Figure 4 shows the operational setup for this application. The collective range of the sensor network is 37 feet. The nodes are kept equidistant from each other along a straight line in the aisle so that the fields of view (FOV) of their webcams overlap. The application tracks people walking using the webcams and estimating their position in the aisle. When a person moves from the FOV of one device to another, the device communicates with its neighbor and hands over the track. Although there may be multiple people in the aisle, for simplifying the tracking it is assumed that there is a single person in the FOV of each node at a given time. The sampling frequency for the study was set to 4 frames per second.

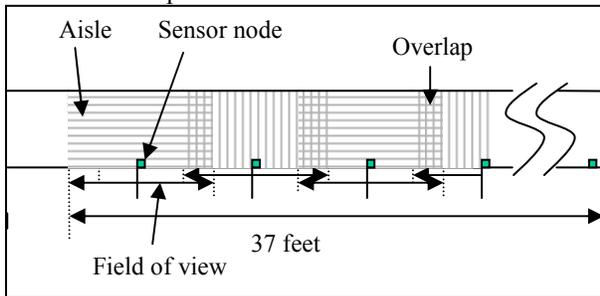


Figure 4, Application setup

Figure 5 shows the functional flow graph of the aislemonitor application for a single node. The key components are *ImageSensor*, *Receiver*, *Estimator*, and *DataCollector*. We have developed two alternative implementations of the Estimator and the DataCollector component, thus the application on a single node can execute in one of four possible configurations. The alternative implementations of the Estimator and DataCollector components are designed to work together. Thus, there are two valid configurations of the functional flow graph.

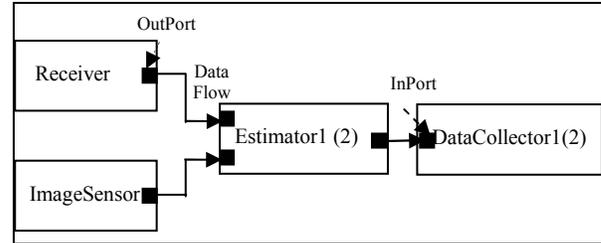


Figure 5. Aislemonitor functional graph

The *ImageSensor* component is responsible for detecting motion based on the difference between the current image and a background image of the aisle and computing the position of the center of mass of the person in the image. The *Receiver* component receives data packets from neighboring nodes by establishing a connection during initialization. The *Estimator1* component implements a Kalman filter, which takes the position and speed passed by the *ImageSensor* component as input to calculate the most likely position of the person in the aisle. The *Estimator2* component is similar to *Estimator1*. However, in addition to tracking people in its own FOV, *Estimator2* also implements a function for predicting the position and the speed if the person is in the FOV of a neighboring node that is disabled. The prediction function implements the dynamical model used by the prediction step of the Kalman filter initialized with the last available position and speed. The *DataCollector1* component receives the position and speed of the person from *Estimator1* and either records the values or hands over the track to the neighbor node. The *DataCollector2* component is also similar to the *DataCollector1* but it also distinguishes between the data obtained by prediction or Kalman filtering to enable the correct initialization of the track in the next node.

The *Receiver* and the *ImageSensor* components continue to execute even during reconfiguration. Only the *Estimator1* and *DataCollector1* components are stopped and their alternatives are started. The SNRAMoLa model of this application is shown in Figure 6. It depicts the dataflow graph of the application components, and the alternative implementations of the Estimator component. The model also captures constraints that are evaluated over the design space for the application, at design time and runtime (dynamic constraints). Some of the example constraints are listed below:

```

C1. (EstimatorChoice.ImplementedBy() =
    EstimatorChoice.Estimator1 implies
    DataCollectorChoice.ImplementedBy() =
    DataCollectorChoice.DataCollector1) and
    (EstimatorChoice.ImplementedBy() =
    EstimatorChoice.Estimator1 implies
    DataCollectorChoice.ImplementedBy() =
    DataCollectorChoice.DataCollector1)

C2. (power() < Pavailable)

```

C3. ($\text{accuracy}() > A_{\text{desired}}$)

where $P_{\text{available}}$ is currently available total power of the nodes, a runtime parameter that is computed by the monitors on the sensor and updated in the models by the reconfiguration controller. A_{desired} is the current accuracy requirements of the application, computed by the monitors on the sensor nodes depending upon the time of the day, the density of traffic in the monitored aisle, and user preferences.

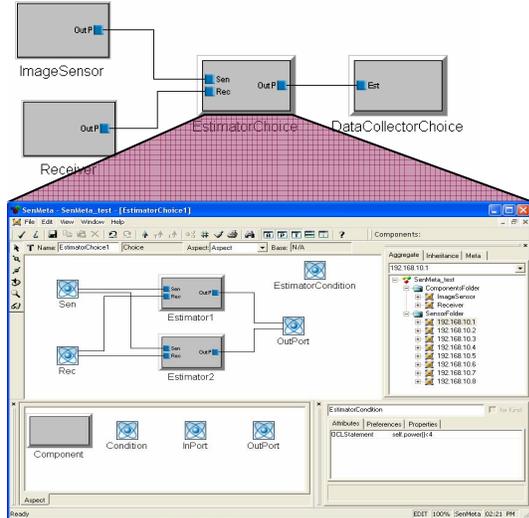


Figure 6. Application model

The application on each node can operate in one of two possible configurations, and additionally a sensor node could be turned off to save power. The overall system can thus operate in 729 (3^8) configurations. However, the application functionality requires that for each node that is turned off, its nearest neighbor must be in the second configuration (Estimator2 and DataCollector2), otherwise it should be in the first configuration (Estimator1 and DataCollector1), thus limiting the number of valid configurations to 69. The reconfiguration is driven by a QoS parameter vector $P_{av} = [p_1, \dots, p_8]$ where p_i indicates the available power on a node, and QoS parameter A_{desired} that indicates the desired accuracy. The monitor component on each sensor node monitors the available power (using a battery sensor) and updates the reconfiguration controller with this information periodically (every 10 sec in the experimental setup). The reconfiguration controller updates the model, and invokes the DESERT tool to solve these constraints on-line. We choose a simple linear composition for power, i.e. the power consumption of each node is a linear sum of the power consumption by each component, and the system wide power consumption is a linear sum of the power consumption of each node. The power consumption of the second

configuration is higher than the power consumption of the first configuration due to the higher computation complexity of the algorithms. Similarly, we chose a linear composition model for accuracy, i.e. the application accuracy is a linear function of number of nodes in the application that are turned on.

Given the runtime available power and the desired accuracy, DESERT returns a set of valid configurations which could include nodes that are operating in first configuration, second configuration, or are turned off. The reconfiguration controller simply returns a configuration and starts computing the instructions necessary to ‘reconfigure’ the system. In order to reconfigure nodes from the first configuration to second configuration, instructions for stopping *Estimator1* and *DataCollector1* components are sent, followed by instructions for starting *Estimator2* and *DataCollector2*. The *ImageSensor* and *Receiver* components are not affected by the reconfiguration. In order to reconfigure nodes from any of the configurations to ‘off’, instructions are sent to set the nodes in hibernation (can be woken by a command sent over the network). It should be noted that our approach is not limited to switching components alone but can handle cases where the new application graph will be different.

A series of experiments with 1, 2, and 3 people walking in the aisle were performed to evaluate the software reconfiguration approach. Figure 7 displays representative results for tracking a single person. The line ‘Configuration 1’ corresponds to the case all nodes are executing the first configuration. The ‘Sensing Gap’ is obtained with the 4th node disabled but still all the remaining nodes execute configuration 1. As it can be seen from the figure, there is a gap between 220in and 375in. Also the next node needs to reinitialize track after the gap which causes an additional error. The line ‘Configuration 2’ illustrates the case when the 4th node has been disabled but nodes 3 and 5 run the second configuration. A dynamical model is used to predict the positions in the gap and there is no need to initialize a new track.

The reconfiguration approach was evaluated by recording the times at which various activities took place. During the tests, time was recorded when (1) the Monitor on the third node sent a message to the GCM, which triggered the reconfiguration process, (2) new configuration files were sent by the base station to the third node, (3) configuration files were received by the Configurator on the third node, (4) reconfiguration commenced on the third node, and (5) reconfiguration was completed.

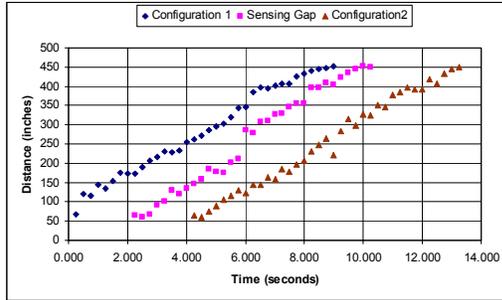


Figure 7. Tracking results

The overhead of the reconfiguration process was measured by performing 10 experiments. The reconfiguration process from the receipt of the message from the node to the dispatch of a new configuration file to it took in average 10 sec. The Configurator upon receipt of a new configuration file performs the actual software reconfiguration on the node in average 8 sec. The total reconfiguration process took in average 18 sec (28 sec if the time required for monitoring is considered). The results of the experiments are summarized in Table 1.

Table 1. Time required for reconfiguration

Component	Location	Time (sec)
Monitor	Sensor node	10
GCM and Controller	Base station	10
Configurator	Sensor node	8
Total Time		28
Total Reconfiguration Time		18

The experiments carried out for evaluation of the software reconfiguration architecture produced satisfactory results. Although the time required for reconfiguration is still considerable it is a major improvement than performing this activity manually. In addition, our software infrastructure allows the selective switching of components without affecting the entire application. For example, during the reconfiguration the ImageSensor component is still active and can store its output while switching and rewiring the Estimator component and therefore, except the delay introduced, the application can continue seamlessly.

8. Conclusions and future work

We have demonstrated an approach for constraint-based dynamic software reconfiguration in sensor networks. Although we used a sensor network consisting of Linux-based sensor nodes communicating via 802.11b, the approach can be modified for other sensor networks in a straightforward manner. In our case study, we didn't have any problems related to network connectivity. For large networks, it's likely that connectivity will affect the method and especially the time required for

reconfiguration. Since the design space exploration is performed in the base station, the approach is well-suited for small to medium sensor network applications. Scalability as the number of nodes and software components increases is a very significant issue. To address this issue, reconfiguration must be performed in-network and such methods are currently under investigation.

9. References

- [1] R. Bryant, "Symbolic Boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys*, 24(3), 293-318, 1992.
- [2] M. Castaldi et al., "A Lightweight Infrastructure for Reconfiguring Applications," *In Proc. of 11th Software Configuration Management Workshop*, Portland, Oregon, USA, May, 2003.
- [3] S-W. Cheng et al., "Software Architecture Based Adaptation for Pervasive Systems," *Int. Conference on Architecture of Computing Systems and Trends in Network and Pervasive Computing*, Karlsruhe, Germany, April 8-11, 2002.
- [4] M.D. Derk and L.S. DeBrunner, "Reconfiguration for Fault Tolerance using Graph Grammars," *ACM Trans. on Computer Systems*, 16(1), 41-54, 1998.
- [5] J. Hill and D. Cullar, "MICA: A Wireless Platform for Deeply Embedded Networks." *IEEE Micro*, 22(6), 12-24, Nov./Dec. 2002.
- [6] G. Karsai, J. Sztipanovits, A. Ledeczki, T. Bapty. "Model-Integrated Development of Embedded Software." *Proceedings of the IEEE*, 91(1):145-164, 2003.
- [7] S. Kogekar et al., "Constraint-Guided Dynamic Reconfiguration in Sensor Networks," *In Proc. of Information Processing in Sensor Networks, IPSN '04*, Berkeley, California, April 26-27, 2004.
- [8] E.A. Lee and T.M. Parks, "Dataflow Process Networks", *Proceedings of the IEEE*, 83(5): 773-801, May 1995.
- [9] D. Mitchell et al., "A QoS Support Framework for Dynamically reconfigurable Multimedia Applications," *In Proc. of the IFIP WG 6.1 Int. Working Conference on Distributed Applications and Interoperable Systems II*, p.17-30, June, 1999.
- [10] S. Neema et al., "Constraint-Based Design-Space Exploration and Model Synthesis." *LNCS 2855*, pp 290-305, Sept 2003, Springer.
- [11] J.B. Warner and A.G. Kleppe A, *The Object Constraint Language: Precise Modeling With UML*, Addison-Wesley, 1999.
- [12] <http://www.openbrick.org>
- [13] http://w3.antd.nist.gov/wctg/aodv_kernel