A Programming Model for the Automatic Construction of USN Applications based on Nano-Qplus

Kwangyong Lee[1], Woojin Lee[2], Juil Kim[2], and Kiwon Chong[2]

[1]Ubiquitous Computing Middleware Team, ETRI, Daejeon, Korea
kylee@etri.re.kr
[2]Department of Computing, Soongsil University, Seoul, Korea
{bluewj, sespop}@empal.com, chong@ssu.ac.kr

Abstract. A programming model for the automatic construction of USN applications based on Nano-Qplus is proposed in this paper. Nano-Qplus is a sensor network platform developed by ETRI. Programs of nodes such as sensors, routers, sinks and actuators in a sensor network are automatically generated through the technique of this paper. Developers can implement USN applications from models of sensor networks. The configuration information of each node is automatically generated from a model. Then, the execution code is automatically generated using the configuration information. Through the technique of this paper, developers can easily implement USN applications even if they do not know the details of low-level information. The development effort of USN applications also will be decreased because execution codes are automatically generated. Furthermore, developers can consistently construct USN applications from USN models using the proposed tool.

1 Introduction

Recent advances in wireless communications and electronics have enabled the development of lowcost, low-power, multifunctional sensor nodes. These sensor nodes, which consist of sensing, data processing, and communicating components, leverage the idea of sensor networks [1]. Ubiquitous sensor network (USN) is a wireless network which consists of a lot of lightweight, low-powered sensors. A lot of sensors which are connected to a network sense geographical and environmental changes of the field. Through USN, things can recognize other things and sense environmental changes, so users can get the information from the things and use the information anytime, anywhere. The sensor networks can be used for various application areas such as military, home, health, and robot.

However, it is difficult to construct USN applications. Resources of nodes in a sensor network are limited and wireless communication between nodes is unreliable. Nodes should also perform low-power operations. Developers should consider these facts, so it is very difficult to construct USN applications. Therefore, it is need to make developers can simply design USN applications by abstracting the details of low-level communication, data sharing, and collective operations.

* This work was supported by the Soongsil University Research Fund.

Accordingly, a programming model for automatic construction from a model of USN application is proposed in this paper. Programs of nodes such as sensors, routers, sinks and actuators in a sensor network are automatically generated from an USN model. Therefore, developers can easily develop USN applications even if they do not know the details of low-level communication, data sharing, and collective operations. The technique of this paper brings focus to USN application on a sensor network platform known as Nano-Qplus [2, 3]. Nano-Qplus is a sensor network platform developed by ETRI. It is a scalable and reconfigurable Nano-OS. It supports a variety of scheduling methods and various energy-efficient power management schemes in order to meet application specific goals.

2 Related Works

In this section, existing works for generation of USN applications are described. The difference between existing works and the technique of this paper is also described.

Cheong et al. [4] have proposed TinyGALS which is a globally asynchronous and locally synchronous model for programming event-driven embedded systems. This programming model is structured such that all asynchronous message passing code and module triggering mechanisms can be automatically generated from a high-level specification. They have implemented the programming model and code generation facilities on a wireless sensor network platform known as the Berkeley motes. Welsh et al. [5] have simplified application design by providing a set of programming primitives for sensor networks that abstract the details of low-level communication, data sharing, and collective operations. Newton et al. [6] have proposed a functional macroprogramming language for sensor networks, called Regiment. The goal of Regiment is to write complex sensor network applications with just a few lines of code.

Boulis et al. [7] have proposed a framework to define and support lightweight and mobile control scripts that allow the computation, communication, and sensing resources at the sensor nodes to be efficiently harnessed in an application-specific fashion. Their framework, SensorWare, defines, creates, dynamically deploys, and supports such scripts. The SensorWare architecture is based on a scriptable lightweight run-time environment, optimized for sensor nodes that have limited energy and memory.

Greenstein et al. [8] have proposes a new configuration language, component and service library, and compiler that make it easier to develop efficient sensor network applications. Their goal is the construction of smart application service libraries: high-level libraries that implement concepts like routing trees and periodic sensing, and that combine automatically into efficient programs. Their language, library, and compiler are collectively called SNACK (Sensor Network Application Construction Kit). Ramakrishna Gummadi et al. [9] have proposed Kairos. Kairos is a natural next step in sensor network programming in that it allows the programmer to express, in a centralized fashion, the desired global behavior of a distributed computation on the entire sensor network. Kairos' compile-time and runtime subsystems expose a small set of programming primitives, while hiding from the programmer the details of

distributed-code generation and instantiation, remote data access and management, and inter-node program flow coordination. Kairos is a simple set of extensions to a programming language that allows programmers to express the global behavior of a distributed computation. Kairos extends the programming language by providing three simple abstractions.

Developers who use the technique of Cheong et al. [4] should write high-level specifications in order to generate sensor network applications. Developers who use the technique of Welsh et al. [5] should develop sensor network applications using the given APIs. Developers who use the techniques of Newton et al. [6], Greenstein et al. [8] and Greenstein Ramakrishna Gummadi et al. [9] should develop sensor network applications using the given languages. Developers who use the technique of Boulis et al. [7] should write script codes in order to generate sensor network applications. On the other hand, developers who use the technique of this paper can automatically generate sensor network applications from models of the applications. They only write USN models using a tool. Therefore, they can easily develop sensor network applications. But, the technique of this paper supports only sensor network applications based on Nano-Qplus platform.

3 A Programming Model for the Automatic Construction of USN Applications

A programming model to construct USN applications based on Nano-Qplus is presented in this section. It is compared to the existing programming models for USN applications. Moreover, the modeling & design of an application using a tool is presented. The algorithm for automatic construction of the application is also presented.

3.1 Concepts of the USN Programming

Figure 1 presents the concept of USN programming described in existing works [4, 5, 6, 7, 8, 9].

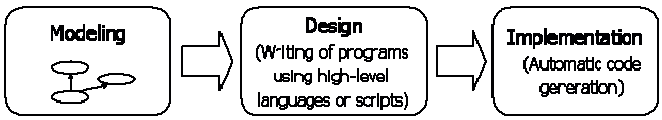

Fig.1. The concept of USN programming in the existing works

A modeling is done and a simple program based on the model is written using the high level language or the simple script. Then the code is automatically generated according to the program. It is important that the program is written using the high level language or the script. The high level language or the script helps users to construct applications, even though they do not know the details of low-level information of USN. A specification-level language, a script language, or APIs were proposed in order to abstract the low-level information in the related works. However,

users should learn the proposed language, the script language or APIs in order to develop USN applications using these techniques.

A technique to complement the existing techniques for the construction of USN applications is proposed in this paper.

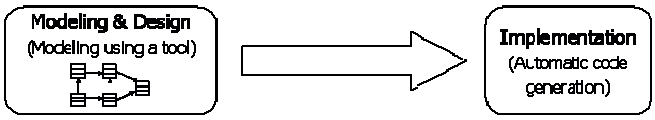

Fig.2. The concept of USN programming in this paper

Figure 2 presents the concept of USN programming proposed in this paper. Developers can implement USN applications by automatically generating execution code of each node in the sensor networks after they do modeling and design the sensor networks using a tool. The execution code is automatically generated from the model. Therefore, users can construct USN applications without learning a language or APIs.

3.2 The Modeling & Design of USN applications

The following is the process for the modelling & design of USN applications.

Step 1 – Write an USN model for an USN application using a tool.

Step 2 – Set up attribute values of nodes in the model using a tool.

Step 3 – Generate the model information using XML in order to automatically generate the configuration information of nodes.

Step 4 – Generate configuration information of nodes from the XML in order to automatically generate execution codes of nodes.

Figure 3 presents the process for the modelling & design of USN applications.

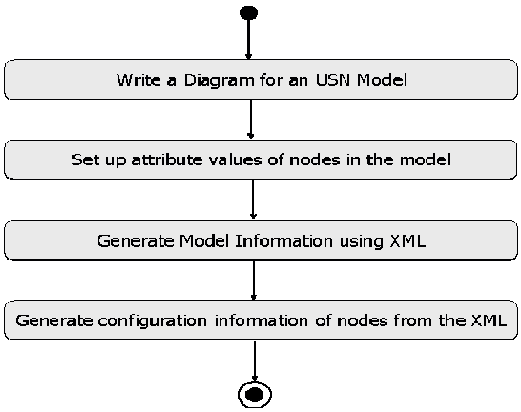

Fig.3. The process for the modelling & design of USN applications

The tool presented in figure 4 is proposed in this paper in order to model and design of USN applications. The user can write a diagram for an USN model and set attribute

values of each node in the model using the tool. The tool generates a XML file which stores the model information. Figure 5 shows the XML file generated by the tool.

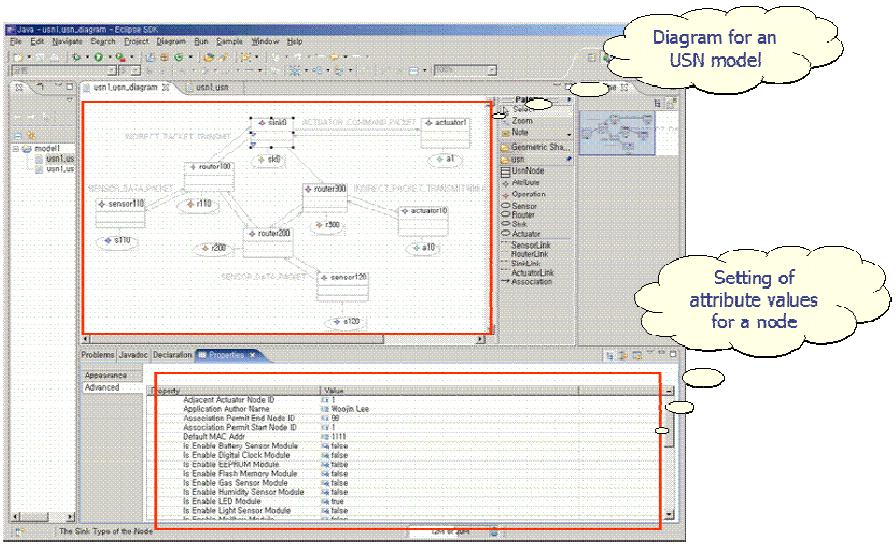

Fig.4. The modeling & design using a tool

```
<?xml version="1.0" encoding="UTF-8"?>
<usn:Usn xmlns:usn="http://www.example.eclipse.org/Usn">
  <nodes>
    <name>sensor110</name>
    <sensorType>#//@sensors.0</sensorType>
    <target>#//@nodes.1</target>
  </nodes>
  <nodes>
    <name>router100</name>
    <routerType>#//@routers.1</routerType>
    <target>#//@nodes.2</target>
    <target>#//@nodes.4</target>
  </nodes>
  <nodes>
    <name>sink0</name>
    <sinkType>#//@sinks.0</sinkType>
    <target>#//@nodes.7</target>
    <target>#//@nodes.3</target>
  </nodes>
  <nodes>
    <name>actuator1</name>
    <actuatorType>#//@actuators.1</actuatorType>
  </nodes>
  <sensors>
    <name>s110</name>
    <applicationAuthorName>Woojin Lee</applicationAuthorName>
    <nodeID>110</nodeID>
    <isEnableNON_BEACON>true</isEnableNON_BEACON>
    <defaultMACAddr>1111</defaultMACAddr>
    <scheduler>Preemption-RR</scheduler>
    <zigbeeRFModule>Star-MeshRoute</zigbeeRFModule>
    <isEnableLightSensorModule>true</isEnableLightSensorModule>
    <isEnableGasSensorModule>true</isEnableGasSensorModule>
  </sensors>
  <routers>
    <name>r100</name>
    <applicationAuthorName>Woojin Lee</applicationAuthorName>
    <nodeID>100</nodeID>
    <isEnableNON_BEACON>true</isEnableNON_BEACON>
```

Fig.5. An example of a XML file generated by the tool

The tool generates .config files from the XML file. The .config files store the configuration information of nodes, and the files are used to automatically generate an USN application. The following is the process for transformation XML to configuration information.

Step 1 – Parse the XML file. Parser generates the parsing tree based on the XML file.

Step 2 – Get the information of each node from the parsing tree.

Step 3 – Generate the configuration information of each node. The information of the parsing tree is transformed to the configuration information. As a result of transformation, .config file for source code generation of each node is generated.

Figure 6 presents the process of transforming XML to configuration information.

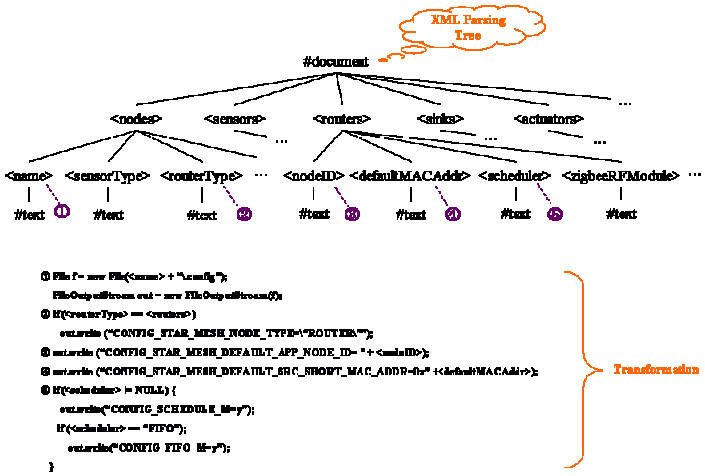

Fig.6. The process of transforming XML to configuration information

The configuration information showed in figure 7 has been automatically generated by the tool in order to generate the source code of a node.

```
#
# ETRI-SSN(or MINI) Menu
#
# CONFIG_EEPROM_M is not set
# CONFIG_FLASHMEM_M is not set
CONFIG_TIMER_M=y
# CONFIG_DIGITAL_CLOCK_M is not set
CONFIG_UART_M=y
CONFIG_PRINTF_M=y
# CONFIG_SCANF_M is not set
CONFIG_ACTUATOR_M=y
CONFIG_LED_M=y
# CONFIG_ADC_M is not set
CONFIG_SCHEDULE_M=y
# CONFIG_FIFO_M is not set
CONFIG_PREEMPTION_RR_M=y
# CONFIG_PWM_M is not set
CONFIG_ZIGBEE_RF_M=y
CONFIG_ZIGBEE_M=y
CONFIG_IEEE_802_15_4_MAC_M=y
CONFIG_STAR_MESH_ROUTE_M=y
CONFIG_STAR_MESH_NODE_TYPE="SINK"
CONFIG_STAR_MESH_DEFAULT_RF_CHANNEL="25"
CONFIG_STAR_MESH_DEFAULT_APP_NODE_ID="0"
CONFIG_STAR_MESH_ADJACENT_ACTUATOR_NODE_ID="1"
CONFIG_STAR_MESH_THIS_NODE_PAN_COORDINATION_ENABLE="TRUE"
CONFIG_STAR_MESH_COORDINATOR_TYPE="NON_BEACON_ENABLE"
CONFIG_STAR_MESH_DEFAULT_SRC_SHORT_MAC_ADDR="0x1111"
# CONFIG_STAR_MESH_USE_DEFAULT_EXTENDED_MAC_ADDR is not set
CONFIG_START_MESH_ASSOCIATION_PERMIT_NODEID_START="1"
CONFIG_START_MESH_ASSOCIATION_PERMIT_NODEID_END="99"
# CONFIG_RSSI_M is not set
# CONFIG_ITC_M is not set
# CONFIG_UTILITY_M is not set
CONFIG_LOG_M=y
```

Fig.7. An example of automatically generated configuration information of a node

3.3 The Automatic Construction of USN applications

The following is the process for generating source code to control each node.

Step 1 – Read Config_Info(.config) file in order to get the attribute values of a node.
Step 2 – Parse Config_info(.config) file and find out selected modules. Then read headers, data and function codes from the DynamicTemplate class according to the selected modules and save them to the template.
Step 3 – Read main code from the HashTable_Main class based on selected modules and save it to the template.

```
Templet Transformation(Config_Info config_Info) {
    templet = getTemplet(config_Info.NODETYPE);
    templet.setAttribute(config_Info.Attribute);

    DynamicTemplate dynamicTemplate = new DynamicTemplate();
    HashTable_Main hashTable_Main = new HashTable_Main();

// Set the NODETYPE of HashTable_Module
    dynamicTemplate.setType(config_Info.NODETYPE);

// Construct code according to the config_Info of each node
    Iterator iterator = Parser.getIterator(config_Info);
    while( iterator.hasNext() ) {
        // Set the name of selected module
        dynamicTemplate.setModuleName(iterator.ModuleName);

        templet.addHeader(dynamicTemplate.getHeader() );
        templet.addData(dynamicTemplate.getData() );
        templet.addFunction(dynamicTemplate.getFunction() ) ;
        templet.addMain( hashTable_Main.getMain() );
        iterator.next();
    }
}
```

Fig.8. An algorithm for generating USN application

Figure 8 presents the algorithm for generating source code of each node. Headers, data and function codes are generated by calling the functions of the DynamicTemplate class according to the type of the target node.

```
public class DynamicTemplate {
        //&1 -- node type, &2 -- module name, &3 -- header, data, or function name
        private String moduleTemplate = "#1_#2_#3";
        private String moduleFileName = "";

        private final String HEADER = "H";
        private final String DATA = "D";
        private final String FUNCTION = "F";

        public void setType(String nodeType) {
                moduleTemplate = moduleTemplate.replaceFirst("&1", nodeType);
        }
        public void setModuleName(String moduleName) {
                moduleFileName = moduleTemplate.replaceFirst("&2", moduleName);
        }
        public String getHeader(String moduleName) {
                return moduleFileName.replaceFirst("&3", HEADER);
        }
        public String getData(String moduleName) {
                return moduleFileName.replaceFirst("&3", DATA);
        }
        public String getFunction(String moduleName) {
                return moduleFileName.replaceFirst("&3", FUNCTION);
        }
};
```

Fig.9. DynamicTemplate class

The DynamicTemplate class used in the algorithm is presented in figure 9. The class includes setType(), setModuleName(), getHeader(), getData() and getFunction() to generate the program for each node. These functions use the *moduleTemplate* field in order to get source codes. The DynamicTemplate class generates the proper source

code using the *moduleTemplate* field dynamically because the codes of headers, data and functions are dependent upon the type of node and module.
The type of the *moduleTemplate* field is "String", and the initial value is "&1_&2_&3". Strings such as "&1", "&2" and "&3" are dynamically replaced according to the type of module and node. When the type of each node is determined, the string "&1" is replaced with the type through the setType() method. The string "&2" is replaced with the name of a module provided by Nano-Oplus through the setModuleName() method. The string "&3" is replaced with "H" (means Header), "D" (means Data) or "F" (means Function) based on the type of required module. For example, the value of the *moduleTemplate* field is replaced as follows by calling functions of the DynamicTemplate class if the type of a node is SINK and Zigbee_Simple module for radio frequency communication of the node is selected.

setType("SINK"); → "SINK_&2_&3"
setModuleName("Zigbee_Simple"); → "SINK_Zigbee_Simple_&3"
getHeader("Zig_Simple") → "SINK_Zigbee_Simple_H"
getFunction("Zig_Simple") → "SINK_Zigbee_Simple_F"

"SINK_Zigbee_Simple_H" is the name of a file which contains header codes of the Zigbee_Simple module for a sink node, and "SINK_Zigbee_Simple_F" is the name of a file which contains function codes of the Zigbee_Simple module for a sink node.

The main function code of each node is generated by the HashTable_Main class. The HashTable_Main class generates the main function code using the hash table presented in table 1. The key of the hash table is the name of a module provided by Nano-Qplus. The main function code is generated according to the type of selected module using the key value of the hash table.

Table 1. Structure of hash table for the main function code

Key	Value
Zigbee_Simple	"mlme_start_request(MY_MAC_ADDRESS, rf_recv_data)"
Zigbee_MAC	"mlme_ll_link_start(NULL, rf_recv_data)"
Zigbee_MAC_StarMesh	"mlme_ll_link_start(NULL, rf_recv_data)"
Scheduler_FIFO	"(*start)((void *)0);"

Scheduler_PreemptionRR	"uint8_t int_handle;
	int_handle = thread_disable_int();
	thread_enable_ints(int_handle);
	pthread_create(NULL, rf_recv_data);
	start_threads();"

4 Case Study with Light Sensing System

An USN application for Light Sensing System such as figure 10 has been developed using the proposed technique in this paper. The Light Sensing System is composed of sensor nodes which sense light data, router nodes which transmit the received data to other nodes, a sink node which is connected to the monitoring system and determines the action command, and a light bulb which contains an actuator to turn the light bulb on.

A USN model was designed for the Light Sensing System. In the model, sensor nodes sense light data and transmit the data to router nodes. The router nodes receive the data and transmit it to the sink node. The sink node receives the data, computes it and transmits it to the actuator node. The actuator node performs an action according to the threshold value.

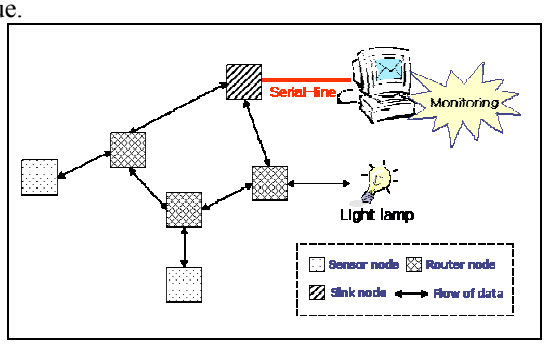

Fig.10. Block diagram of Light Sensing System

The system of figure 10 was developed after the application was automatically generated from the designed model. Result that applies, sensing light data was forwarded from sensor node to router node and router node sent forwarded data to sink node. An action command according to the light value was forwarded from sink node to router node and router node sent forwarded the action command to actuator. Actuator turned the light bulb on or off according to the action command.

Figure 11 is the source code of sink node automatically generated based on the configuration information presented in figure 7.

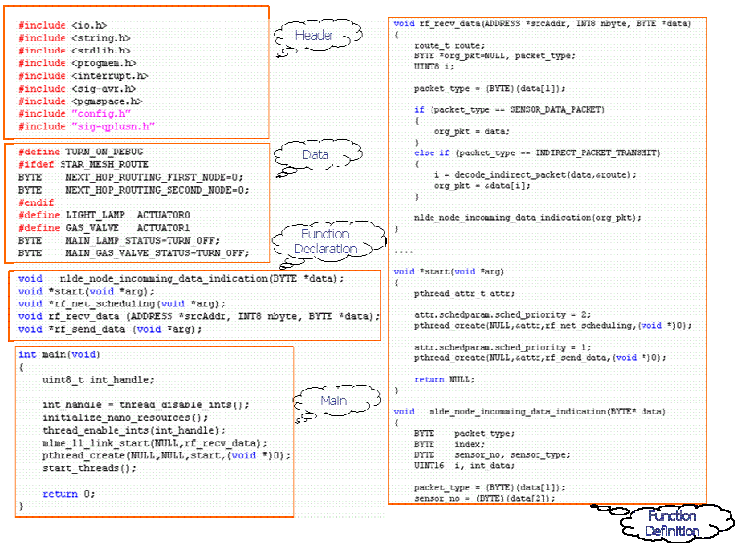

Fig.11. An example of automatically generated code

5 Conclusion

A programming model for the automatic construction of USN applications based on Nano-Qplus is proposed in this paper. Developers can implement USN applications by automatic generation of execution code of each node in the sensor networks after they make models of the sensor networks. The configuration information of each node is automatically generated from a model. Then, the execution code is automatically generated using the configuration information. The modelling tool to make an USN model and generate the configuration information of each node is proposed in this paper. The templates and an algorithm for automatic code generation are also presented. Through the technique of this paper, developers will easily implement USN applications even if they do not know the details of low-level communication, data sharing, and collective operations. The development effort of USN applications also will be decreased because execution codes are automatically generated. Furthermore, developers can consistently construct USN applications from USN models using the proposed tool.

References

[1] Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci, "A Survey on Sensor Networks," IEEE Communications Magazine, Volume 40 Issue 8, pp.102-114, August 2002.

[2] Kwangyong Lee et al., "A Design of Sensor Network System based on Scalable & Reconfigurable Nano-OS Platform," IT-SoC2004, October 2004.

[3] ETRI Embedded S/W Research Division, "Nano-Qplus," http://qplus.or.kr/

[4] E. Cheong, J. Liebman, J. Liu, and F. Zhao, "Tinygals: a programming model for eventdriven embedded systems," SAC, 2003.

[5] M. Welsh and G. Mainland, "Programming sensor networks using abstract regions," NSDI, 2004.

[6] R. Newton and M. Welsh, "Region streams: Functional macroprogramming for sensor networks," DMSN, 2004.

[7] A. Boulis, C. Han, and M. B. Srivastava, "Design and implementation of a framework for efficient and programmable sensor networks," MobiSys, 2003.

[8] B. Greenstein, E. Kohler, and D. Estrin, "A sensor network application construction kit (SNACK)," SenSys, 2004.

[9] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan, "Macro-programming Wireless Sensor Networks Using Kairos," LNCS 3560, pp. 126–140, 2005.