American Journal of Computation, Communication and Control
2015; 2(1): 1-6
Published online February 20, 2015 (http://www.aascit.org/journal/ajccc)
ISSN: 2375-3943

# Object-Oriented Publish/Subscribe Programming Language for Wireless Sensor Networks Reprogramming

**Biao Dong**

School of Computer and Software, Nanjing Institute of Industry Technology, Nanjing, China

## Email address

dongb@niit.edu.cn

## Citation

Biao Dong. Object-Oriented Publish/Subscribe Programming Language for Wireless Sensor Networks Reprogramming. *American Journal of Computation, Communication and Control.* Vol. 2, No. 1, 2015, pp. 1-6.

## Keywords

Wireless Sensor Networks,
Reprogramming,
Publish/Subscribe,
Object Oriented

Received: January 23, 2015
Revised: February 2, 2015
Accepted: February 3, 2015

## Abstract

This paper presents an object-oriented publish/subscribe(Pub/Sub) programming language, called OPS, for modeling and implementing the architecture of wireless sensor networks (WSNs) reprogramming applications. Considering the ability to efficiently update applications running on sensor nodes which is necessary for WSNs reprogramming, event and subscription models provide suitable programming abstractions by integrated Pub/Sub with object-oriented environment. Focusing on the modifications that should be easy to reflect on the sensor nodes, we select an abstraction level for sending the compiled updates, and make a trade-off between WSNs operation costs and reprogramming costs. We design and implement a prototype system on OPS. Simulation experiments imply that OPS is simplicity, while ensuring good flexibility in updating code.

## 1. Introduction

WSNs are composed of a large number of communication nodes with limited sensing, processing and computational capabilities. These nodes, developed at a low cost and in small size, can be randomly deployed across the monitored area, providing dense sensing close to physical phenomena, processing and communicating this information, and coordinating actions with other nodes. Due to the dynamic nature of WSNs, environmental change and other factors, WSNs often need to update the codes running at the sensor nodes. These updates may include bug fixes, the perception for new information as well as the use of more efficient algorithms. WSNs have many characteristics such as large amount of nodes and wide distribution. Sometimes, these nodes can be deployed in the places that are difficult to reach. Therefore, it isn’t realistic to manually update the codes on the nodes. Usually, the updated code needs to be distributed to each sensor node in a wireless way. This behavior of modifying the application and sending the compiled updates to sensor nodes is called reprogramming.

Pub/Sub is an asynchronous communication paradigm that supports many-to-many interactions between a set of clients. A client can be an information publisher, an information subscriber, or both. Client interactions are data-centric: publishers describe their publishable events, subscribers express their interest in events, and Pub/Sub protocol delivers the published events to their corresponding event subscribers. The loose coupling of clients eliminates the burden of context information gathering and processing by resource constrained devices. Existing methodologies based on the request/response model for system software design, rooted in principles of object-oriented design, lead to tightly-coupled interactions, and lack coordination capacities. Pub/Sub paradigm is particularly suitable for loosely- coupled communication environment. Our design of OPS is motivated by the idea of using object-oriented Pub/Sub environment for WSNs reprogramming.

The rest of the paper is organized as follows. In Section 2, we review some related works. Our main methods including Pub/Sub for OPS, event model, and subscription model are presented in Section 3. The prototypical implementations including compiler and interpreter are given in Section 4, followed by the performance evaluations in Section 5. Finally, the paper is concluded in Section 6.

## 2. Related Work

In order to solve this problem and put our work in perspective, we give a brief overview of related works. A major approach is based on embedded operating system (OS) that allows for WSNs reprogramming applications. TinyOS (Gay D et al., 2005) supports for reconfigurability can be attributed to the encapsulation-by-modules feature in nesC that provides a unified interface [1]. Contriki (Dunkels A et al., 2004) employs advanced reprogramming support in the form of loadable modules as well as an abstract programming interface that applications can use to perform actual transmission and routing of data message [2]. LiteOS (Cao Q et al., 2008) provides Unix-like abstractions for WSNs, and supports software updates through a separation between the kernel and user applications, which are bridged through a suite of system calls [3]. Lin Gu et al. (2006) design a new OS kernel, the t-kernel, to perform extensive code modification at load time. The modified code and the OS work in a collaborative way supporting the aforementioned features [4]. MANTIS OS (Bhatti S et al., 2005) is flexibility in the form of cross-platform support and testing across PCs, PDAs, and different micro sensor platforms. A key design feature is support for remote management of in-situ sensors via dynamic reprogramming and remote login [5].

Query processing in sensor networks is another relevant area to our work. Researchers have noted the benefits of a query processor-like interface to sensor networks and the need for reprogramming to limited power and computational resources. TinyDB (Madden S R et al., 2005) is a distributed query processor that runs on each of the nodes in a sensor network, also incorporates a number of other features designed to minimize power consumption via acquisitional techniques[6]. Yao Y et al. (2002) introduce the Cougar approach to tasking sensor networks through declarative queries. Since queries are asked in a declarative language, the user is shielded from the physical characteristics of the networks[7]. Tavakoli A et al. (2007) propose Snlog, a programming paradigm for declaratively specifying sensor networks systems. Snlog allows the user to specify an application using a high-level language, which is subsequently fed to a compiler which builds a runtime query processor that executes on each node[8].

There are other ways of achieving WSNs reprogramming. Taherkordi A et al. (2013) consider WSN programming models and run-time reconfiguration models as two interrelated factors, and present an integrated approach for addressing efficient reprogramming in WSNs [9]. Maia G et al. (2013) propose a multicast-based over-the-air programming protocol that considers a small world infrastructure [10]. Krontiris I and Dimitriou T (2006) present Scatter, a secure code dissemination protocol that enables sensor nodes to authenticate the program image efficiently [11]. A multi-block forward error correction technique is used (Park T et al., 2013), in which blocks were encoded using a rateless code, such as a Luby transform code [12]. Ortega-Zamorano F et al. (2014) presents an alternative based on an on-chip learning scheme in order to adapt the node behaviour to the environment conditions [13]. Mazumder B and Hallstrom J O (2013) present an incremental code update strategy used to efficiently reprogram WSNs, and adapt Hirschberg's algorithm for computing maximal common subsequences to build an edit map specifying an edit sequence [14]. Semparuthi R and Yuvaraj R (2014) propose a secure distributed reprogramming protocol named SDRP with node classification algorithm. Based on the evaluation results, all nodes will be registered as users before giving the reprogramming authorization [15].

The above researches effectively improve the efficiency of software development for WSNs and running quality. But with the expansion of WSNs applications and the growth of WSNs scale, reprogramming makes the following demands on the programming languages. Firstly, languages need to provide suitable programming abstractions, help programmers to efficiently build programs which have a reasonable structure, and are easy to be modified and reused. Secondly, modifications to source code should be easy to reflect on the sensor nodes, so the execution mechanism of language needs to select an appropriate abstraction level for sending the compiled updates. Finally, the execution mechanism of language faces the tradeoff between the WSNs operation costs and the reprogramming costs. This paper proposes an object-oriented Pub/Sub programming model for WSNs reprogramming, and addresses two issues:

(1) The establishment of event model and subscription model in object-oriented environment.

(2) The design and implementation of compiler and interpreter that support for WSNs reprogramming.

Compared with other works on reprogramming on WSNs based on high-level abstract, several advantages to using OPS in WSNs reprogramming can be discussed as follows.

(1) Query processing approaches, such as TinyDB and Cougar, regard WSNs as relational database table, and provide SQL-like language for users to query data. Due to a too high programming abstraction level, they can only support part of WSNs applications. While based on object-oriented language environment, OPS supports most of WSNS applications.

(2) A logical language, such as Snlog, abstracts WSNs as globally deductive databases that can be logically programmed. Therefore, these aren't conducive to WSNs reprogramming. By reducing the size of the updated code, the OPS interpreter efficiently support reprogramming.

## 3. Our Methods

### 3.1. Event and Subscription Model

In an object-oriented environment, communications between different objects are carried out by the event delivery. When an object receives an event, a method is called, therefore, the method calls can be treated as events. Pub/Sub events are considered as value events, a value event can be mapped to one or several method events.

Definition 1 method event. Method event in an object-oriented environment is defined as follows:

Create Event <Event> <EventClause>

<Event Clause>=[Before|After] <Name.MethodName>

The Event is a string uniquely identifying event, the Before and After events are produced before and after method execution, the Name is a class or object name, and the MethodName is a method name.

In an object-oriented system, method events are divided into pre-execution and post-execution method events, and include two kinds of method events: class method events and general method events. Class method events are generated before and after invoking class methods, such as creating an instance, deleting an instance. Similarly, general method events are generated before and after executing object methods.

Definition 2 subscription. Subscription in an object-oriented environment is defined as follows:

Create Sub <Subscription-Name>

On <Event-Trigger >

Condition: <Condition>

Act: <Action>

The Subscription-Name is a string uniquely identifying a subscription, the Event-Trigger is a list of Method events, the Condition is a Boolean function, and the Action denotes a procedure defined by an application program. As used in this definition, the subscription means as follows: when a method event in an event-trigger occurs, the condition is evaluated, and then if the condition is true, actions are executed.

### 3.2. Operational Semantics

In order to understand OPS more clearly and guide the implementation of OPS language processing system, we learn from the calculus of communication system [16], and give the operational semantics of event and subscription.

Definition 3 substitution. A substitution θ is defined as the set of variables and its values, and is denoted as $\theta=\{c_1/v_1,\ldots,c_n/v_n\}$. Where $v_1,\ldots$, and $v_n$ represent different variables, $c_1,\ldots,$and $c_n$ represent constants. The symbol $\perp$ indicates an invalid substitution.

Given an event Event($v_1,\ldots,v_n$). A substitution $\theta=\{c_1/v_1,\ldots,c_n/v_n\}$ will be returned when the event happens.

Definition 4 subscription semantics. Suppose there is a subscription $Sub_{id}$: On <$E_{id}$ >, Condition: <$Con_{id}$>, Act: <$Act_{id}$>. When the message channel β receive a triggering message, OPS detects whether the message meet the condition $Con_{id}$. If the condition is met, the method corresponding to $E_{id}$ is substituted into θ. Subsequently, OPS uses θ to assign values to parameters, executes $Act_{id}$. The subscription semantics of $Sub_{id}$ is expressed as the following. Stask denotes the sequence of tasks to be executed, <$Sub_{id}$, Stask> represents an ordered pair that defines structural pattern of sentence, and means that $Sub_{id}$ will be executed in the current task sequence. $act_{id}^{1}(v\theta)\triangleright\cdots\triangleright act_{id}^{m}(v\theta)$ represents that all actions in $Act_{id}$ are carried out in sequence.

## 4. OPS Language Processing System

### 4.1. OPS Architecture

In conjunction with system software, programs written in embedded languages are compiled into the binary images. Since the binary images are usually larger, it's more costly to distribute the images through wireless transmission. At the same time, the application programs in the binary images tightly couple with system software, there are many restrictions in the process of incremental update to the application codes.

In order to reduce the cost of updating sensor code, OPS uses intermediate code to decouple the applications and the system software. Intermediate code contains only the codes associated with the relevant business logic, it has the advantages of small volume, and is easy to transport. OPS language processing system selects the architecture which includes the server-side compiler and the sensor-side interpreter, and as shown in Fig.1.

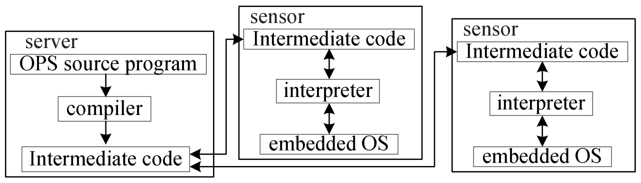

***Fig. 1.*** *Architecture of OPS processing system*

The compiler is responsible for converting the OPS source program to intermediate codes which are independent of the specific sensor platforms. The interpreter and embedded OS reside in the sensors, and execute the received intermediate codes by way of interpretation.

### 4.2. Compiler

The main task of the compiler is to generate intermediate code which contains only the business logic of the application, and reduces the amount of code transmission caused by code updates so as to improve the efficiency of reprogramming. At the same time, taking into account extra burden brought by the interpretation and execution mechanism executing certain application code, another important task of the compiler is to optimize intermediate code to improve the interpreter computing speed and reduce power consumption.

The computational demands for the interpreter in the sensors mainly originate from execution of external procedures. External procedures are virtual procedures that could be anything when compiled, including class methods, attributes, methods, events and procedures. External procedures are used to store class type information for the compiler subsystem. Fig.2. shows the compiler.

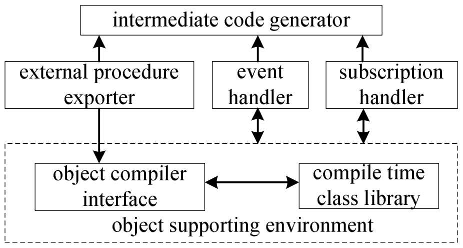
intermediate code generator
external procedure exporter
event handler
subscription handler
object compiler interface
compile time class library
object supporting environment

***Fig. 2.*** *OPS compiler*

The working process of the compiler is divided into three phases: the first phase is initialization, the second will focus on compiling, and the third phase is to generate intermediate code. In the initialization phase, the compile time class library establishes class information. In the compile phase, the object supporting environment generates push instructions, creates or finds the number of the called external procedures, and then, fills in the import descriptions of the external procedures import table. The import declaration used for the class library contains all information needed for external function registered to the compiler. In the code generation phase, the compiler completes two works. First of all, subscriptions are translated into the following procedure.

```
Procedure <Subscription name> (<Triggering event
  parameter list>)
Begin
  If <conditions> Then <actions>;
End.
```

Second, the association between event and subscription name is established.

### 4.3. Interpreter

The interpreter, together with system software to support its running, reside in the sensors, and are responsible for executing intermediate code by way of interpretation. The interpreter is implemented on the embedded OS, and as shown in Fig.3. Its core part includes external procedure which supports environment, scheduler, subscription library management, and communication handler.

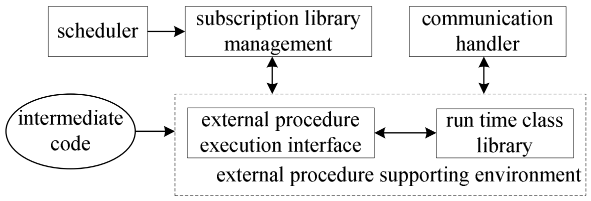
scheduler
subscription library management
communication handler
intermediate code
external procedure execution interface
run time class library
external procedure supporting environment

***Fig. 3.*** *Architecture of the OPS interpreter*

The execution processes of the external procedure supporting environment are as follows. Firstly, according to the external procedure type, the environment calls the corresponding classification function. Secondly, the classification functions generate the call parameters according to the characteristics of the call functions. Finally, according to the different calling conventions, method calls and parameter-passing modes, the environment handles the results returned by the underlying call functions. According to the type of event, the scheduler is responsible for calling different external process modules to perform subscription tasks.

The running state of the interpreter is determined by its core part. The external procedure supporting environment only involves six aspects of work related to executing external procedure, such as calling conventions, method call, returning results, constructors and destructors, virtual methods, and passing parameters. Regardless of the upper application logic of WSNs, the work is only related to sensor hardware and OS. That is, this part code of the interpreter codes can be relatively unchanged. When extending the capability of the interpreter, we just need to update the rest of the interpreter which relates to event and subscription management.

## 5. Simulation Experiment

By OPS, the simulation experiment realizes the following function: The temperature sensor collects temperature data once every one second. And then, the temperature data, together with the sensor node number, are sent to the base station. The parent node forwards the data packet of its child node. We use this application as a case to evaluate the OPS performance in the following aspects. Firstly, we demonstrate the simplicity of the language by using OPS to write the applications. And then, by comparing the size of the compiled code, we show the flexibility in updating the codes. In the evaluation of the OPS performance, we use a program written in nesC as a comparative target.

```
(1) main program
var app:TApplication; t:TTimer; r:TReceive;
begin
app:=TApplication.CreateNew(app,0);
t:=TTimer.CreateNew(app, sys_Initial, 1000, sys_Infinity);
r:=TReceive.CreateNew(app);
while app.Enabled do
Application.ProcessMessages;
app.free;
end.

(2) Define the events.
Create Event OnTimer After TTimer.OnTimer;
Create Event Get_Temperature After TReceive.Get;

(3) Define the subscriptions.
Create Sub Insert_Temperature       Create Sub Forwarding_Temperature
On TTimer.OnTimer                   On TReceive.Get
Condition: Aowner=app;              Condition: Aowner=app;
Act: Begin                          Act: Begin
Insert(Temperature);                        Send(Parent-ID, ID, Temperature);
End.                                     End.
```

***Fig. 4.*** *An OPS program for collecting temperature*

(1) The simplicity of OPS. An OPS program for collecting temperature is shown in Fig.4. The compiler generates intermediate codes for the main program and the subscriptions, and then establishes associations between the method events and the subscriptions. After the interpreter executes TTimer. Create New (app, sys_Initial, 1000, sys_Infinity), the event

On Timer is triggered once every one second. After the interpreter executes TReceive. Create New (app), the event Get_ Temperature is triggered when the object r receives the data packets. The elements on the top of the external procedure supporting environment stack stands for the actual value that the interpreter uses. Then the intermediate codes corresponding to the subscriptions are executed.

The program basically can be viewed as a simple translation of the application requirements. Programmers don't need to consider the implementation details which don't correlate with the application requirements, such as hardware equipment management, message buffer management, etc.

(2) The flexibility in updating code. The target platform is set to be Mica Z sensor. After the nesC program for collecting temperature is compiled, its binary image is 53180 bytes. While the intermediate code of the OPS program is 1805 bytes. The intermediate code is only 3%, compared with the binary image. The explanation behind this is as follows. The nesC program, together with the embedded OS, is compiled. Firstly, most of the code in the compiled binary image doesn't have direct relation with the business logic of the application, while the OPS intermediate code contains only the code related to the business logic. And then, compared with the binary image, OPS intermediate code has a higher level of abstraction.

(3) The costs related to the interpreter on each node. MicaZ can provide 128KB ROM and 4KB RAM. The interpreter needs 40375 bytes ROM and 3500 bytes RAM. Since the interpreter uses static memory allocation, so this situation is acceptable. We perform the experiments using MicaZ nodes for grid topology. For the grid network, a node situated at one corner of the grid acts as the base node, the transmission range $R_{tx}$ of a node satisfies $\sqrt{2d}<R_{tx}<2d$, where d is the separation between the two adjacent nodes in any row or column of the grid. In our experiments, if a node receives a packet from a non-adjacent node, it is dropped. We perform these experiments for grids of size 4x4. We run both versions 120s on ATEMU simulator [17], and come to the following conclusions. As the sensors are closer to the base node, the more packets are dropped. But the packet loss status of OPS interpreter is very close to that of nesC algorithm, and the nesC version cuts the total energy consumption by 10 percent than that of the OPS version. Therefore, in this application, the way of interpretation can meet the requirements.

## 6. Conclusions

In this paper, we propose OPS, an object-oriented Pub/Sub programming language, to provide the architecture of WSNs reprogramming. OPS is based on event and subscription model. We design and implement OPS language processing system, and derive conclusions from the simulation experiments. The results show the simplicity of the language, and the flexibility in updating code.

## Acknowledgements

This work was sponsored by Qing Lan project(Jiangsu province, china) and open fund project of Jiangsu provincial research and development center of intelligent sensor network engineering technology, china (ZK13-02-03, Software technology, platform and application for sensor network).

## References

[1] Gay D, Levis P, Culler D. Software design patterns for TinyOS. ACM SIGPLAN Notices, 2005, 40(7): 40-49.

[2] Dunkels A, Gronvall B, Voigt T. Contiki-a lightweight and flexible operating system for tiny networked sensors. Local Computer Networks, 2004. 29th Annual IEEE International Conference on. IEEE, 2004: 455-462.

[3] Cao Q, Abdelzaher T, Stankovic J, et al. The liteos operating system: Towards unix-like abstractions for wireless sensor networks. Information Processing in Sensor Networks, 2008. IPSN'08. International Conference on. IEEE, 2008: 233-244.

[4] Gu L, Stankovic J A. t-kernel: Providing reliable OS support to wireless sensor networks. Proceedings of the 4th international conference on Embedded networked sensor systems. ACM, 2006: 1-14.

[5] Bhatti S, Carlson J, Dai H, et al. MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms. Mobile Networks and Applications, 2005, 10(4): 563-579.

[6] Madden S R, Franklin M J, Hellerstein J M, et al. TinyDB: an acquisitional query processing system for sensor networks. ACM Transactions on database systems (TODS), 2005, 30(1): 122-173.

[7] Yao Y, Gehrke J. The cougar approach to in-network query processing in sensor networks. ACM Sigmod Record, 2002, 31(3): 9-18.

[8] Tavakoli A, Chu D, Hellerstein J M, et al. A declarative sensornet architecture. ACM SIGBED Review, 2007, 4(3): 55-60.

[9] Taherkordi A, Loiret F, Rouvoy R, et al. Optimizing sensor network reprogramming via in situ reconfigurable components. ACM Transactions on Sensor Networks (TOSN), 2013, 9(2): 14.

[10] Maia G, Aquino A L L, Guidoni D L, et al. A multicast reprogramming protocol for wireless sensor networks based on small world concepts. Journal of Parallel and Distributed Computing, 2013, 73(9): 1277-1291.

[11] Krontiris I, Dimitriou T. Scatter–secure code authentication for efficient reprogramming in wireless sensor networks. International Journal of Sensor Networks, 2011, 10(1): 14-24.

[12] Park T, Kim S Y, Kwon G I. Multi-block FEC for reprogramming wireless sensor networks. Electronics Letters, 2013, 49(14).

[13] Ortega-Zamorano F, Jerez J M, Subirats J L, et al. Smart sensor/actuator node reprogramming in changing environments using a neural network model. Engineering Applications of Artificial Intelligence, 2014, 30: 179-188.

[14] Mazumder B, Hallstrom J O. An efficient code update solution for wireless sensor network reprogramming. Proceedings of the Eleventh ACM International Conference on Embedded Software. IEEE Press, 2013: 4.

[15] Semparuthi R, Yuvaraj R. A Efficient QOS based User selection in Secure and Distributed Reprogramming Protocol for Wireless Sensor Networks. IJRCCT, 2014, 3(4): 521-525.

[16] Milner R. A Calculus of Communicating Systems. New York: Springer-Verlag, 1982.

[17] Polley J, Blazakis D, McGee J, et al. ATEMU: a fine-grained sensor network simulator. Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on. IEEE, 2004: 145-152.