SNAP - Sensor Network Application Plugin

WeeSan Lee
University of California, Riverside
weesan@cs.ucr.edu

Abstract

This paper presents SNAP, Sensor Network Application Plugin. It consists of a collection of C++ classes that wrap around TinyOS's [5] system components and a macro `PLUGIN` to instantiate the application. With this architecture, it is shown that writing embedded code for Sensor Networks is not much different than writing an ordinary C++ code using multiple inheritance. The process of learning a new language is avoided. Like TinyOS, applications using SNAP would include pertinent components only into the final executables. The size of final executables is only about 0.8K larger in average than the ones written in C. However, the application code using SNAP is much shorter, concise and easier to understand. Another advantage of using this architecture (the `PLUGIN` macro to be specific) is the forward compatibility which allows the same set of applications become plugins in the future on supported hardware and software by simply recompiling them without any code modification.

Keywords

Plugin, Sensor Networks, C++

1 Introduction

Sensor networks consist of a number of small, energy-constrained, embedded and distributed sensor nodes. Once deployed, they will self-organize among themselves to form a network and execute a specific pre-programmed task.

The soul and driving force behind those devices is an component-based operating system called TinyOS [5]. Since most of the sensor nodes are small and battery powered, the footprint of the TinyOS has to be small, in the order of several K bytes. For the same reason, an event-driven model is used so that devices can spend most of the time idle (in order to save power) and wait for the interesting events to happen. In order to maintain small footprint and achieve code re-usability, TinyOS provides the ability to wire all the pertinent components, either user applications or existing system components, together at compile time via a description/configuration file to generate a final monolithic executable that can be later uploaded to the sensor nodes.

In TinyOS-0.6.1, all components were written in C wrapped around with extensive C macros, a component interface (.comp file) and a graph of components (.desc file). It was not very intuitive and difficult to maintain the consistency among those files. Also, all components including their variables are instantiated within frames. A macro, called VAR, has to be used to access those variables in the code to hide the name of the frames. While debugging, the frame names have to be known in order to access the variables as well. Later, TinyOS-1.X.X was released to address some of the problems in previous version. Instead of using C macros, a new language along with a new compiler, called nesC [4], was introduced. In fact, nesC is just a C preprocessor that converts the nesC code into ordinary C code and then calls GCC to finish up the final compilation. The problem with this approach is that users have to re-learn a new computer language. There could be some steep learning curves before ones can actually master the new language. Although a lot similar to C, nesC differs C in many subtle ways. For example, like its predecessor, debugging nesC programs is not as intuitive as on C programs because GDB has not yet supported nesC natively. All variables used in nesC are mangled with a frame name and dollar signs in the final executables. In order to access these variables in a GDB session, the developers need to be able to unmangle the variables on-the-fly in order to access them. This would cut productivity in developing applications on the sensor nodes.

To address these problems, this paper proposes SNAP, Sensor Network Application Plugin. The contributions of this architecture is twofold: 1) it provides a thin layer, C++ classes, to wrap around system components such that "wiring" those components can be as simple as doing multiple inheritances for the application in C++. No extra description/configuration files are needed, thus, eliminate the inconsistency problem. In addition, event handlers are actually virtual functions override. For example, in order to handle system clock timeout event, one could simply override the virtual member function `void fire(void)`. This, in fact, simplifies the development process; 2) it provides a macro called `PLUGIN` to instantiate each application. Although it does not do much in the current architecture other than instantiate the applications, it does eliminate the mangled variables problem. Moreover, as the technology evolves, like many others, I envision that the future sensor nodes would equip with a more powerful CPU and plenty of memory for both program and data. Although limited, the application running on the nodes would be able to allocate/free memory from/to the heap at will without too much overhead. DLL (Dynamic Link Library) would also be supported. With these new two key features, application level DLL will become possible. With this vision in mind, the `PLUGIN` macro, also provides the applications the forward compatibility to become an application level DLLs, also as known as, **plugins**, without any code modification.

Like TinyOS's traditional component-based approach, SNAP-based applications preserves the ability to include pertinent components only into the final executables to achieve minimal code size. Like any C++ programs, applications using SNAP would introduce some overheads in term of size, for example, vtable for virtual functions. However, I will show that the code size is not much larger than those generated from the C code. I will also show that the SNAP-based code will be smaller, more concise and self-explanatory. Last but not least, there are other advantages of using SNAP, mainly from the C++ language itself including Object-Oriented Design, code re-usability, encapsulation/information hiding, polymorphism and etc [6, 7].

The rest of this paper is organized as follows: Section 2 discusses the design trade-offs and implementation issues. Section 3 compares SNAP with TinyOS. Section 4 presents some related works. Finally, I conclude my work and discuss future directions in Section 5.

2 Design

This section discusses the SNAP design decisions and trade-offs as well as its implementation issues.

2.1 To C++ or not to C++

C++ is chosen simply because of its object-oriented design and methodology [6, 7]. It has become a very popular choice for implementing plugins for various applications [3]. Also, it is a trend to shift from C to C++ even for embedded programming [8, 2]. However, while there are some C++ features that are very useful for large scale applications, they are not compelling in an embedded environment. Thus, it should be used with care especially on CPU and memory constrained devices such of sensor nodes. The following C++ features are avoided in SNAP due to runtime overhead in terms of speed and size:

- Run-time type identification. RTTI facility provides type information about every class with virtual functions on runtime for type identification features such as `typeid` and `dynamic_cast`. It would increase the application size even if those features are not being used. So, it should be better turned off to save some space. In GNU C++, the compiler flag to turn off this feature is `-fno-rtti`.
- Exception handling. Although it is useful for dealing with errors, it imposes unnecessary overhead such as code size increased and speed degraded. It is better turned as well. In GNU C++, the compiler flag to turn off this feature is `-fno-exceptions`.
- Templates and STL. Templates are useful for making generic function or classes. However, templates can easily lead to code explosion. It is better to avoid using that. Since STL utilize templates, it is better off to avoid it all together.

2.2 To TinyOS-0.6.1 or not to TinyOS-0.6.1

TinyOS comes in two favors: 1) TinyOS-0.6.1, written in C with extensive C macros; 2) TinyOS-1.X.X, written in a new programming language especially designed for sensor network systems called nesC [4]. Since nesC is an extension to C and can not be compiled by a C compiler without prior being compiled/preprocessed into C code via the nesC compiler, there is no way SNAP can be added on top of that even though TinyOS-1.X.X back ported a lot of components from its predecessor. Moreover, it is easier to wrap around C code with C++, so, it does not make sense to implement SNAP on TinyOS-1.X.X on which nesC is used and incompatible with C/C++. As a result, SNAP is based on TinyOS-0.6.1 instead. In the rest of this paper, TinyOS implies TinyOS-0.6.1 unless stated otherwise.

2.3 Implementation

TinyOS supports multiple platforms: PC[1], MICA, MICA2 and etc. A `Makefile` that takes the name of the platforms as a parameter is in place to support this. For example, if users would like to generate executables for MICA platform, they could type: `make mica`. Likewise, for PC emulation executables, they could type: `make pc` and etc. In order to minimize the changes as much as possible in current source tree and preserve the same syntax as well as the ability to build original TinyOS executables, new platforms are introduced to support executables using SNAP. For each original platform, a "++" is appended after the name of the platform for executables using SNAP. For example, `make mica++` and `make pc++` would generate executable for MICA and PC using SNAP respectively[2].

Unlike TinyOS-0.6.1 that compiles all pertinent components into object files and later link them with the application, and, TinyOS-1.X.X that combines application and pertinent system components into a single C file and later compile it using GCC, SNAP provides C++ classes that wrap around all system components and are needed to be compiled and made libraries in advance before they can be linked with the application. One of the advantage of using libraries is that only components that are needed by the applications will be linked together with the applications into the final executable. The outcome of this is compact executable size, and, shorter compilation and linkage time subsequently during development process. For each platform, there is a library, namely `libplatform.a`, associated with it under each `platform` directory in the source tree. In addition to those platform libraries, under `system` directory, there are various system libraries corresponding to each platform built from the same source code. For example: `libsystem_mica++` and `libsystem_pc++` are for MICA and PC platform respectively. Both platform and system libraries are needed in order to generate the final SNAP-based executables. The task of generating those libraries is taken care by the modified `Makefile` automatically. No extra work needs to be done from the developers.

While most of the changes are done by adding new C++ files which later be compiled as libraries, there are a couple of exceptions that need to actually add code into existing C files guided by a SNAP specific directive, `__tosplusplus`. Scheduler, for example, is such exception case. In order to support task posting inside the derived classes, an extra context pointer is added into scheduler's internal data structure as a placeholder for the

[1]This is a PC-mode simulation.

[2]Currently SNAP is supported only partially on MICA and PC platform, more code will be added when time permits. The main purpose of supporting PC platform at the first place is to verify the correctness of SNAP's implementation while supporting MICA platform is to find out the size overhead introduced by C++.

instance of the class that posts the task. And because of this, one of the task API needs to be changed to accommodate this as well.

A special class, called `Plugin`, is introduced and it is required for each application to at least inherit from this class. Depending on the applications, if clock service is needed, the applications also need to inherit from a class `Clock` as well and etc. Essentially, each system service will be a class by itself. It is very intuitive and easy to recognize those services simply by their class names. For example, class `Leds` is for accessing LEDS, `Photo` is for light sensing service, `Task` is for posting tasks inside the class and so on. The advantage of using this multiple inheritances for the applications is to avoid using extra description file which might eventually lead to inconsistency between the configuration file and the C code.

After an application class declaration and implementation inside the class, a new macro called `PLUGIN`, has to be used to instantiate the application. What the macro does right now is to instantiate an object for the class as well as fulfill some requirements that are needed by the C++ compiler to generate the final executable. It also serves the purpose of abstracting the process of application instantiation. By doing that, it provides the application the forward compatibility in supporting plugins on newer platforms in the future without any code modification.

In order to keep the main program, `MAIN.c`, for each platform untouched so that porting efforts to a new platform become easier, instead of changing the code inside the main programs, macro `PLUGIN` is defined such that it provides both `void MAIN_SUB_INIT_COMMAND()` and `void MAIN_SUB_INIT_CMMAND()` that are needed and called by the main program although only one of them is needed. The reason being that, with SNAP, the application initialization is done by the constructor and its parent's constructors when the object is being instantiated. The application can, in fact, be started right away inside the constructor without violating the init and start order for the application. This comes for free from the C++ syntax. So, what macro `PLUGIN` does now is simply instantiating the object in the init command and does nothing in the start command as shown in the code snippet below:

```
#define PLUGIN(P)                              \
    static char appFrame[sizeof(P)];           \
    void *operator new(unsigned int size) {\
        return (appFrame);                     \
    }                                          \
    void operator delete(void *p) {            \
        /* Do nothing here */                  \
    }                                          \
    void MAIN_SUB_INIT_COMMAND() {             \
        new P;                                 \
    }                                          \
    void MAIN_SUB_START_COMMAND() {            \
        /* Do nothing here */                  \
    }                                          \
    ...
```

In order to keep the original instantiation order[3] without changing the main program, and to instantiate the object when the init command is called, the object has to be instantiated *inside* the init command. Statically instantiation is not an option because statical objects are always instantiated first even before the `int main(void)` is executed which is undesired. The trick used in the macro `PLUGIN` is to "dynamically" allocate the object without actually allocating memory for the object. From the macro `PLUGIN` code above, an array with the size of the object is statically allocated, then override the `new` operator to return that array. By doing that, the object can be safely allocated "dynamically" *inside* the init command. Also, `delete` operator has to be overrided as well by doing nothing. Like TinyOS, SNAP is not doing any memory allocation anywhere within the code except for the trick presented above, so, it is safe to perform the trick here.

3 Evaluation

In this section, I compare TinyOS and SNAP in the following 4 aspects: 1) functionalities; 2) size comparison; 3) code comparison; and 4) compilation time.

3.1 Functionalities

Since TinyOS is implemented in C (procedure-based) while SNAP is in C++ (object-oriented based), in addition to the fact that the design and syntax are different, the program's initialization process as well as the flow of code is different. It is very important to make sure the code behave and function the same regardless of the implementations. One of the nice features on TinyOS is its PC-mode simulation with debugging facilities. By comparing the debug output from the PC-mode simulation from both implementations, I concluded that both implementations are identical.

3.2 Size comparison

One concern of C and C++-based programs are their executable size. C++ code are known to generate bigger executables. In this section, I compare the size of executables generated from TinyOS's component-based approach and those using SNAP. In Table 1 and Table 2, they show the size and the size difference of code and data section of AVR platform executables generated from both TinyOS's component-based and SNAP approach. Although the code and data size of SNAP is about 771 bytes and 35 bytes larger respectively, the final executables using SNAP are still small enough to fit into 8K bytes of program memory and 512 bytes of data memory [5]. In other words, the extra few bytes are only 9.4% and 6.8% out of the program and data memory respectively. I believe that the advantages of using SNAP are outweighed the size increased in the final executables.

It is also worth mentioned that since current SNAP implementation consists of a number of C++ classes wrapped around existing C functions, there are still rooms for improvement in term of executable size by implementing the whole TinyOS in C++ from scratch.

[3]Some system components need to be instantiated before the application.

Table 1: **Code size comparison:**

Application	**TinyOS's Size**	**SNAP's Size**	**Size diff**
blink	1078	1268	190
sense	1590	2562	972
sense2	1740	2770	1030
sens_to_leds	1652	2546	894
Average			**771**

Table 2: **Data size comparison:**

Application	**TinyOS's Size**	**SNAP's Size**	**Size diff**
blink	4	16	12
sense	14	52	38
sense2	14	66	52
sens_to_leds	14	52	38
Average			**35**

3.3 Code comparison

In Table 4, it shows the **blink** code[4] from TinyOS and SNAP approach side-by-side. What the code does is to make the red LED blink on the sensor node at 1Hz. Although there is no scientific method or objective way to compare both, it is obvious that the code using SNAP is shorter and the interfaces are clearer due C++ features such as information hiding, polymorphism, default value and etc [6, 7].

As shown on the right side of Table 4, it is application dependent to decide which classes it should inherit from. In this case, the **blink** application schedules the system clock to fire at the rate of 1Hz. When the timer expired, it will turn on or off the red LED. Obvious enough, the application needs to at least inherit from the class `Leds` and `Clock`. As mentioned in Section 2, every application must also inherit from the class `Plugin` which is a mechanism to enable all SNAP-based applications to become plugins on supported platform in the future without any code modification.

Unlike TinyOS, the clock initialization is part of the application constructor initialization which is being enforced by the C++ syntax. Thus, no extra function call or "command" is need to initialize system resources or start system services. As matter of fact, both are combined into the application constructor as mentioned in Section2. Also, the `void fire(void)` is a pure virtual function in class `Clock`. By definition, all inherited classes need to override them. In this case, the **blink** application overrides this function to deal with the clock events.

For completeness sake, the equivalent **blink** code from nesC is also included in Table 3. **BlinkM.nc** is the module implementation and **Blink.nc** is the configuration.

One thing worth mentioning is that both TinyOS versions mangle the variable names by either prepending the variables by a frame name or inserting dollar signs inside the variables. Either way, the outcome is to further complicate the already complex and difficult debugging task. SNAP, on the other hand, does not have this problem. Debugging a SNAP-based program behaves exactly like doing on any regular C++ programs.

[4]All comments are taken out and the filename is added as comment at the top of each file.

Table 3: **Code comparison.** *The equivalent blink code from nesC*

```
// BlinkM.nc
module BlinkM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Leds;
  }
}
implementation {
  command result_t StdControl.init() {
    call Leds.init();
    return SUCCESS;
  }
  command result_t StdControl.start() {
    return call Timer.start(TIMER_REPEAT,
                            1000);
  }
  command result_t StdControl.stop() {
    return call Timer.stop();
  }
  event result_t Timer.fired()
  {
    call Leds.redToggle();
    return SUCCESS;
  }
}

// Blink.nc
configuration Blink {
}
implementation {
  components Main,BlinkM,SingleTimer,LedsC;
  Main.StdControl -> SingleTimer.StdControl;
  Main.StdControl -> BlinkM.StdControl;
  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.Leds -> LedsC;
}
```

3.4 Compilation time

In TinyOS-0.6.1, for each application, all pertinent components and application are compiled, object code are generated locally and finally all object code are linked together to generate the final executable. The problem with this approach is that the system components used by each application need to be recompiled locally regardless if those components have been compiled before for other applications.

In TinyOS-1.X.X, all pertinent components and application written in NesC are pre-processed and converted into a big C file which is then compiled using GNU C. The problem with this approach is worst its previous version. Whenever there are some changes, even just one single byte change in the application, the whole compilation process will be triggered. All the previous compilation efforts do not help any subsequent compilations.

Table 4: **Code comparison.** *The code and description file from TinyOS (left) vs the code using SNAP (right).*

```
/* BLINK.c */
#include "tos.h"
#include "BLINK.h"

TOS_FRAME_BEGIN(BLINK_frame) {
        char state;
}
TOS_FRAME_END(BLINK_frame);

char TOS_COMMAND(BLINK_INIT)(){
  TOS_CALL_COMMAND(BLINK_LEDr_off)();
  TOS_CALL_COMMAND(BLINK_LEDy_off)();
  TOS_CALL_COMMAND(BLINK_LEDg_off)();
  VAR(state)=0;
  TOS_CALL_COMMAND(BLINK_SUB_INIT)(tick1ps);
  return 1;
}

char TOS_COMMAND(BLINK_START)(){
  return 1;
}

void TOS_EVENT(BLINK_CLOCK_EVENT)(){
  char state = VAR(state);
  if (state == 0) {
    VAR(state) = 1;
    TOS_CALL_COMMAND(BLINK_LEDr_on)();
  } else {
    VAR(state) = 0;
    TOS_CALL_COMMAND(BLINK_LEDr_off)();
  }
}

/* blink.desc */
include modules{
MAIN;
BLINK;
CLOCK;
LEDS;
};

BLINK:BLINK_INIT MAIN:MAIN_SUB_INIT
BLINK:BLINK_START MAIN:MAIN_SUB_START

BLINK:BLINK_LEDy_on LEDS:YELLOW_LED_ON
BLINK:BLINK_LEDy_off LEDS:YELLOW_LED_OFF
BLINK:BLINK_LEDr_on LEDS:RED_LED_ON
BLINK:BLINK_LEDr_off LEDS:RED_LED_OFF
BLINK:BLINK_LEDg_on LEDS:GREEN_LED_ON
BLINK:BLINK_LEDg_off LEDS:GREEN_LED_OFF
BLINK:BLINK_SUB_INIT CLOCK:CLOCK_INIT
BLINK:BLINK_CLOCK_EVENT CLOCK:CLOCK_FIRE_EVENT
```

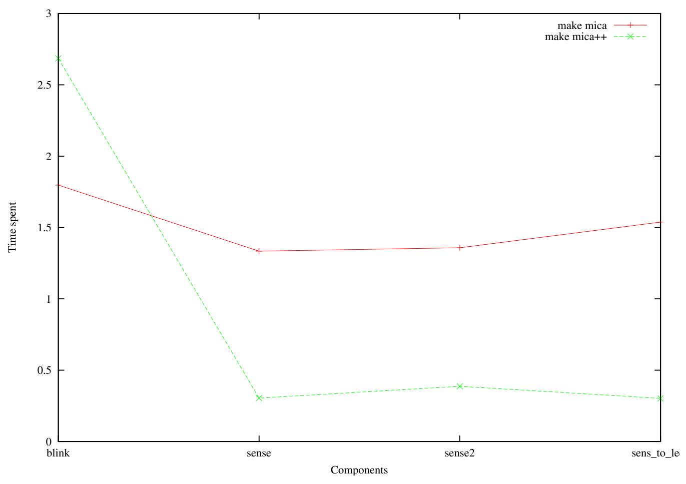
```
// blink.cc
#include "plugin.h"
#include "leds.h"
#include "clock.h"

class Blink : public Plugin,
              public Leds, public Clock {
protected:
    void fire(void) {
        redToggle();
    }

public:
    Blink(void) : Clock(tick1ps) {
    }
};

PLUGIN(Blink);
```

Figure 1: **Time taken to compile the components**



Unlike both versions of TinyOS, SNAP takes advantage of library. All system components are compiled and archived to be libraries. Only the very first application compilation will trigger the initial libraries construction. Subsequent application compilations involve only compilation on the application and linkage with the libraries, which is very fast. Another advantage of using library is that only pertinent components will be linked into the final executables. The goal of minimum size of executables almost comes for free.

Figure 1 shows that the compilation time for both TinyOS-0.6.1 and SNAP-based applications. As shown by the red line, TinyOS-0.6.1 spends almost constant time to compile different components. SNAP, however, since it is using libraries, in addition to compile the application, it also needs to build all the libraries for the first time. As shown by the green line, the compilation time starts to drop significantly, even much shorter than TinyOS-0.6.1, for subsequent compilations.

4 Related Work

In the project MiLAN [9], it has plugin in their architecture, but, that is for supporting transport layer from various platforms. Unlike MiLAN, SNAP provides a collection of C++ classes to serve as a middle layer between the applications and services provided by the hardware, for example, sensing the light or temperature, inside TinyOS. Also, SNAP is based on TinyOS and built on top of it.

The Embedded C++ [2], a project backed by major Japanese semiconductor manufactures such as Fujitsu, Hitachi, Panasonic, NEC, Toshiba and etc back in 1995, focused on developing a C++ specification that were used for their embedded devices. While the goal of the Embedded C++ is to provide embedded systems programmers a subset of ISO/ANSI C++ language and the Standard C++ runtime library, SNAP does not imply those constraints. Rather, SNAP utilizes C++ features as much as possible wherever it sees fit to ease the development process, yet, not to introduce too much overhead in term of speed and code size. Developers using SNAP have to use their own judgment to leverage certain C++ features and the overhead imposed by those features on the embedded devices.

Unlike the Embedded C++, SNAP does not treat multiple inheritance as evil. Rather, it is a very neat feature to "wire" different components nicely without introducing extra description file. Also, virtual base classes are used to impose the applications to override the proper functions required by the system components.

5 Conclusion

In this paper, I present SNAP, Sensor Network Application Plugin. It provides C++ classes for TinyOS system components that can easily be "wired" together by multiple inheritances from an application. This avoids extra description or configuration files that could become inconsistency with the .c files over time. The application development for the sensor networks becomes easier: just like an ordinary C++ programming which does not involve much of learning curves and there is no macro needed to wrap around variables being used. Also, debugging becomes easier without constructing mangled variable names on-the-fly. All the nice features above increase the code size for only about 0.8K in average which is about 9.3% increased on a platform with 8KB of program memory and 512 bytes of data memory.

One important feature of SNAP is the forward compatibility. By using macro `PLUGIN` to instantiate an application, it provides the application the forward compatibility to become a plugin automatically once recompiled on a DLL-capable platforms without any code modification.

As the technology evolves, the vision that sensor nodes are equipped with more powerful CPU and more memory would become true in the near future. Together with the help of Plugin Dissemination Protocol[5] [1], can the plugins not only possible run on the DLL-capable platforms, but also can they be loaded on any sensor nodes that have already been deployed at will. There are a number advantages of doing so: 1) new task can be assigned on the sensor nodes after being deployed based on their physical location. For example, it is more suitable for sensor nodes that closes to the source of light to sense light than sound. This information would not be available until the nodes are deployed; 2) the ability to re-assign a new task to existing sensor networks. There are life cycles for any sensor networks. When their mission has been accomplished, except the fact that they run out of power source, each nodes in the existing networks can be re-programming on-the-fly and appointed a new task. In other words, the life cycle of the sensor networks is extended and prolonged with new missions; 3) the ability to learn the mission and states from an old or dying node. This is extremely useful for mission critical sensor networks to continue their missions by replacing some old or dying nodes without affecting the functionalities of the rest of the nodes.

So, the future work is to continue developing a reliable and efficient Plugin Dissemination Protocol that make the nice things described above happen in the near future.

Last but not least, SNAP will be extended to cover more system components as well as to support more platforms.

[5]This is a protocol that is currently being developed to selectively disseminate plugins from based/sink node to any other sensor nodes

References

[1] Plugin Dissemination Protocol. `http://www.cs.ucr.edu/~weesan/snap/pdp.txt`.

[2] The Embedded C++. `http://www.caravan.net/ec2plus/`.

[3] Zinf Audio Player. `http://www.zinf.org/`.

[4] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Programming Language Design and Implementation (PLDI)*, June 2003.

[5] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. *ASPLOS*, 2000.

[6] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, Massachusetts, Mar. 1994.

[7] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 3rd edition, 1997.

[8] The Embedded C++ Technical Committee. A New Work Item Proposal: C++ for Embedded Systems (Embedded C++). `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1998/n1151.asc`.

[9] H. S. C. Wendi B. Heinzelman, Amy L. Murphy and M. A. Perillo. Middleware to support sensor network applications. *IEEE Network Magazine Special Issue*, 2004.