CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

Title	THAWS: automated design and deployment of heterogeneous wireless sensor networks
Authors	Harte, Seán;Popovici, Emanuel M.;O'Flynn, Brendan;Ó Mathúna, S. Cian
Publication date	2008-09-01
Original Citation	Harte, S., Popovici, E.M., O'Flynn, B., Ó Mathúna, S.C., 2008. THAWS: automated design and deployment of heterogeneous wireless sensor networks. WSEAS Transactions on Circuits and Systems, 7(9), pp. 829-838.
Type of publication	Article (non peer-reviewed)
Link to publisher's version	http://www.wseas.us/e-library/transactions/circuits/2008/27-1401.pdf
Rights	Copyright © 2008 WSEAS
Download date	2026-11-02 06:03:11
Item downloaded from	https://hdl.handle.net/10468/47



UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

THAWS: Automated Design and Deployment of Heterogeneous Wireless Sensor Networks

SEÁN HARTE[1,2], EMANUEL M. POPOVICI[1], BRENDAN O'FLYNN[2], CIAN O'MATHUNA[2]
[1] Department of Microelectronic Engineering, University College Cork,
Cork, IRELAND
[2] Microelectronics Applications Integration Group, Tyndall National Institute,
Lee Maltings, Cork, IRELAND
sean.harte@tyndall.ie http://www.tyndall.ie/mai/wsn.htm

Abstract: - This research focuses on the design and implementation of a tool to speed-up the development and deployment of heterogeneous wireless sensor networks. The THAWS (Tyndall Heterogeneous Automated Wireless Sensors) tool can be used to quickly create and configure application-specific sensor networks, based on a list of application requirements and constraints. THAWS presents the user with a choice of options, in order to gain this information on the functionality of the network. With this information, THAWS uses code generation techniques to create the necessary code from pre-written templates and well-tested, optimized software modules from a library, which includes an implementation of novel plug-and-play sensor interface. These library modules can also be modified at the code generation stage. The application code and necessary library modules are then automatically compiled to form binary instruction files for each node in the network. The binary instruction files then wirelessly propagate through the network, and reprogram the nodes. This completes the task of targeting the wireless network towards a specific sensing application. THAWS is an adaptable tool that works with both homogeneous and heterogeneous networks built from wireless sensor nodes that have been developed in the Tyndall National Institute. Its advantage over traditional methods of WSN development is simplification of development.

Key-Words: - Wireless sensor networks, Automated application development, Code generation

1 Introduction

A Wireless Sensor Network (WSN) is made from a potentially large number of sensor nodes that are capable of communicating wirelessly. The sensor nodes must be inexpensive to enable a wide deployment that can record sensor data with a high spatial and temporal resolution. The nodes must also have a small physical size and have a long lifetime to allow them to be used in a large number of applications. This paper focuses on networks that take sensor readings from many nodes and transmit them back through the network to a gateway node that can be connected to a PC. The sensor data can then be analysed. Such a network can perform many tasks, such as water quality monitoring [1], or ensuring efficient and safe manufacturing plants [2], or medical applications [3].

Each node in a WSN can be viewed as being made from a number of hardware components, as shown in Fig. 1. The node is built by combining these components or a subset of these components.
In selecting the components, the target application must be considered, to ensure the desired functionality, and performance of the system.

To create the optimum network for a particular application, it may be beneficial to have many different types of nodes with different functions that together create a single heterogeneous network. One reason for this is that nodes can have different functions depending on what type of sensors they are connected to. A second reason is that, to save cost, each node should only have the minimum hardware required to perform its task. If a node only has to take a reading every 10 seconds and then transmit it, a very low-powered processor is sufficient.

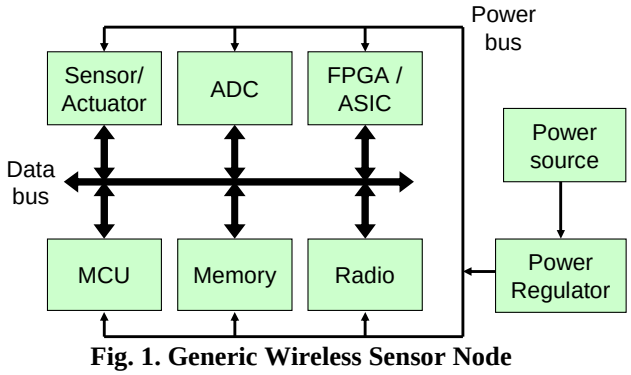


Fig. 1. Generic Wireless Sensor Node

On the other hand, a more powerful processor is required for more advanced tasks, such as routing in large networks, encryption, data compression, and error correction. These tasks are not possible to implement on a very low-powered processor.

1.1 WSN Application Development

The hardware developments in miniaturising sensors (e.g. MEMS sensors) and improving the energy efficiency of electronic components have made possible a lot of research on software suitable for WSNs. New communication algorithms have been designed for large-scale wireless networks, where energy consumption is an important factor and low-power radios create unreliable connectivity [4]. Other research is focusing on operating systems that can run on very limited processors, and still provide support for applications, such as TinyOS [5].

There are also difficulties creating applications for WSNs. Currently, many applications are developed using low-level programming languages such as C or nesC [6]. To develop a new application requires someone with experience in programming with these languages. It also requires being familiar with the various libraries available. This makes fast development and deployment of networks difficult.

There also can be other difficulties in developing applications for WSNs. To save energy, code is often event-driven. For example, the node can be woken up by a timer, take a reading, and then go back into a sleep mode. The event-driven approach is implemented through the use of interrupts which creates the opportunity for corrupted data due to race-conditions if the programmer is not careful [7]. Access to variables that are used by an interrupt must disable interrupts while accessing the variable. For example, if you want to read a variable called `state`, and check if it is equal to 'a'. If it is equal to 'a' then you want to set it to 'b' and do some processing.

```
if (state == 'a') {
    state == 'b';
    doProcessing();
}
```

The code above can introduce errors if `state` is also part of a read-modify-write cycle within an interrupt. After the code above has read the value of `state`, but before it has changed the value, an interrupt could also read the value and change it to 'c'. However when the code above continues executing, `state` will be set to 'b', so the change to 'c' will have been lost. The code should be written as follows to ensure correctness:

```
disableInterrupts();
old_state = state
if (state == 'a') {
    state == 'b';
}
enableInterrupts();
if (old_state == 'a') {
    doProcessing();
}
```

Such details are difficult to remember when accessing a variable and can lead to bugs that are difficult to find a debug.

The event-driven approach also makes simple tasks complicated due to the use of *split-phase* function calls. For example, consider a `i2c_send()` function. If this is a *blocking* function it will wait until all the data is sent before returning. The disadvantage of this is that it would waste a lot of computation cycles as it waits for the relatively slow I^2C interface (commonly 100 kHz) to finish sending the data. This is computation time that could be spent doing more useful work. If the function returns immediately and keeps sending data in the background (using interrupts to send a new byte when needed) there is a risk that the communication could be corrupted, for example by going to sleep mode, or by another part of the application modifying the data before it is sent. With the use of the split-phase technique, `i2c_send()` will return to the caller quickly, and continue sending data in the background. However it will signal that the transmission is finished by calling a *callback* function in the application. The application must make sure that the transmission won't be interrupted until this callback function is called.

Although this method is reliable and allows energy-aware application, it increases the complexity of application development. Some attempts have been made to address this issue such as a pre-compiler that makes all calls blocking, but then introduces a syntax that allows a number of code sections to be run concurrently [8].

1.2 Heterogeneous Networks

A more fundamental difficulty is created by the distributed nature of WSNs. Some applications can be simplified by assuming that all nodes are identical. However, as described above, real networks may be heterogeneous so as to minimize cost while retaining functionality where required. Developing an application for such a network means

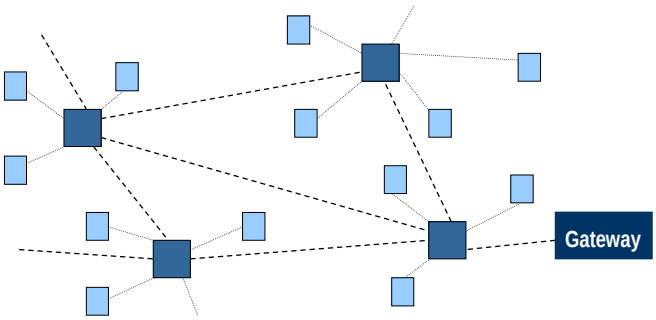


Fig. 2. Two-tiered heterogeneous network

developing different code to execute on each node. This is time-consuming and the application logic becomes separated into many different files, making debugging and future development difficult. A simple example of this is that if you change the format of packets, you will have to redevelop the code that is running in both the transmitter and the receiver.

This paper introduces a system called THAWS that simplifies application development for WSNs. It allows a developer to avoid all the above difficulties while creating applications that can run on heterogeneous networks, as well as homogeneous. The user can specify what they want the network to do, and its associated constraints, without worrying about how this will be implemented. The THAWS tool creates the necessary code to meet the user's specification. The code is then compiled, and the compiled binary files can be sent to the network, which then reprograms itself.

This paper first briefly discusses the hardware nodes that were used when developing THAWS. Then the design, development and use of the tool to help rapidly develop WSN applications is presented. The tool is analysed and compared with other similar systems. Finally, future work and conclusions are discussed.

2 Implementation details

The tool is implemented to work on a two-tiered network with two different classes of nodes, as shown in Fig. 2. The first node is small in size, inexpensive, and has very low-power energy consumption. The second node is bigger in size, and also more expensive. However it has more processing capability. They are used to build a heterogeneous network where the larger, more powerful node can provide the backbone of the network, and do any heavy information processing that is required. In the THAWS system, each larger node supports a cluster of smaller of the smaller,

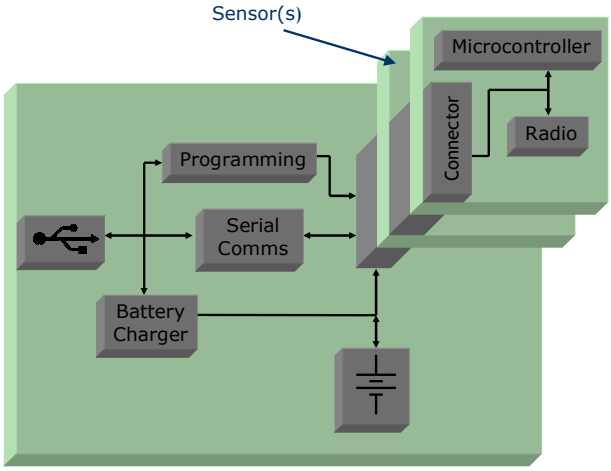


Fig. 3. Modular wireless sensor node design

cheaper nodes that can be used for sensor interfacing and more simple tasks. The smaller nodes do not have to worry about routing; they always just transmit their own information to the larger parent node. The larger node is suited to higher-powered long-distance communication between clusters as they can have a large battery. The two nodes are described in the next section.

2.1 Tyndall Wireless Sensor Nodes

In the Tyndall National Institute a number of different nodes have been developed. Along with various application specific nodes, two modular nodes have been designed with a size of 10 mm by 10 mm [9], and 25 mm by 25 mm [10]. These are referred to as the 10mm and 25mm nodes. Both these nodes are made up of a number of different layers as shown in Fig. 3. Each node has a processing and transceiver layer. Sensor layers can then be connected with application specific sensors, for example temperature sensors, humidity sensors, accelerometers, gyroscopes, etc. In addition to sensors, a battery or energy harvesting device can be

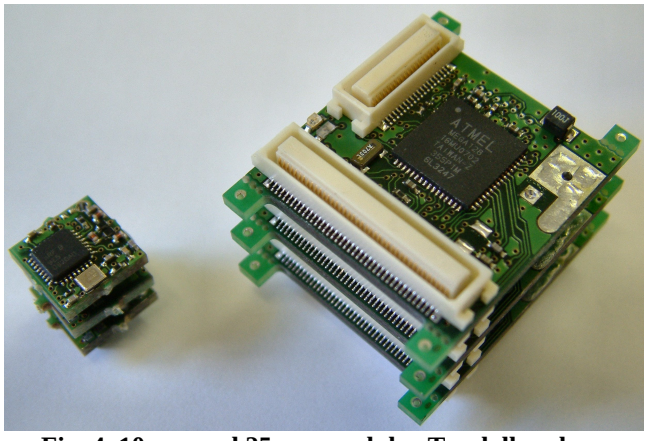


Fig. 4. 10mm and 25mm modular Tyndall nodes

Table 1. Comparison of 10mm and 25mm Tyndall nodes

	10mm	**25mm**
Size	10 mm x 10 mm	25 mm x 25 mm
Energy used in sleep mode	8.9 µW	30 µW
Range	< 100 m	< 3.8 km
Connectivity	30 pins	120 pins
Processing Capability	Microcontroller with 4 kB code memory	Microcontroller with 128 kB code memory + FGPA

connected to provide a power supply. This modular approach allows the nodes to be used to build sensor networks for many applications e.g. environmental monitoring [1], and an inertial measurement system [11]. A photo of the 10mm and 25mm nodes is shown in Fig. 4.

The 25mm node has more powerful processing capabilities than the 10mm node. This is provided by a layer with an Atmel ATmega128 microcontroller with 128 kB of program memory. There is also an FPGA layer that can be used for intensive processing, such as forward error correction [12], cryptography, or image processing. The 25mm has a number of different layers for RF communications. In the 2.45 GHz frequency band there is a layer using a Nordic nRF2401 transceiver and another layer using an Ember EM2420 ZigBee compatible transceiver. There is also a 433/868/915 MHz layer using a Nordic nRF905 transceiver, which allows a longer range, of up to 3.8 km in line-of-sight conditions, compared to the 2.45 GHz options, which has a maximum range of about 200 m. The drawback is that bandwidth is limited to 50 kbps, compared to 1000 kbps for the Nordic nRF2401 [13]. Data in sensor networks is often only a few bytes, so the difference in bandwidth is not necessarily significant. However it will require that the radio is transmitting for a longer time and therefore using more energy.

For the 10mm node, there is currently a single transceiver layer. This uses a Nordic nRF9E5 chip. This chip has a radio that is compatible with the Nordic nRF905 so this allows heterogeneous networks to be built [14]. This chip also has an integrated 8051-compatible microcontroller with a limited 4kB program memory. The small size of the 10mm nodes allows a greater range of applications, for example it can be more easily embedded into clothing, or it can be used in medical applications. The 10mm node is cheaper due to reduced PCB size, the lower component count, and lower assembly cost (due to fewer components). The range of the 10mm node is less as the antenna (a quarter-wavelength monopole) does not perform will with such a small ground reference, and also less than optimal design of the balun circuitry (that matches the differential output of the chip with a single ended antenna) in order to fit it into such a small area.

A summarizing comparison of both nodes is given in Table 1. Using the 10mm nodes together with the more powerful 25mm nodes allows a lot of flexibility in building WSNs suitable for a wide range of applications.

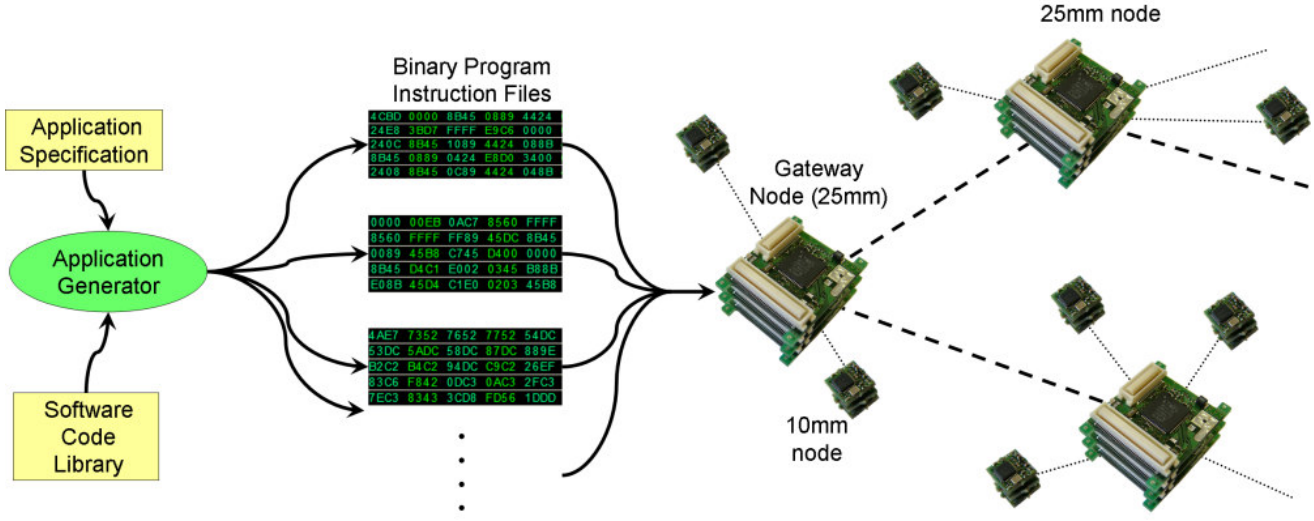


Fig. 5. Application development tool

2.2 THAWS Overview

The core of the THAWS system is an application generating tool. This is introduced in Fig. 5.

The tool has two inputs. The first of these is a software library containing modules of code that act as primitives in building up a WSN application. Some of the modules are in the form of templates that are customised for varying application requirements. The second input into the tool is a description of the desired application. This defines the functionality of the network, and also constraints of the network. For example the type of sensors, number of nodes etc. and network topology are declared.

Using these two inputs, the tool then outputs binary program images for each node in the network. This is done by first producing C files and then compiling these using the appropriate compiler. The use of wireless in-network programming then allows the network to be programmed or reprogrammed/reconfigured to have the desired functionality.

2.3 Software Code Library

The performance and efficiency of the final developed application will depend greatly on the performance and efficiency of the software library. Both energy efficiency and the memory (RAM and ROM) footprints were considered when creating the library. The limited ROM of the 10mm node especially requires efficient code. The code must also be reliable to minimise maintenance costs of the network.

2.3.1 Hardware abstraction layers

Some common modules are required in each WSN application. Modules are needed to interface to radios, and interface to sensors. This code for interfacing to hardware follows HAL (Hardware Abstraction Layer) principles [15]. A common interface is defined for hardware that has similar functionality. For example each of the 4 radios used by Tyndall nodes share a common low-level interface as defined here:

```
rf_init(channel, power, addr, netId);
rf_disable();
rf_send(address, msg, length);
rf_receiveEnable(receiveBuffer);
rf_receiveDisable();
rf_setChannel(channel);
rf_setPower(power);
rf_callback(msg, msgInfo);
```

Currently there is a very simple MAC layer on top of this low level layer that supports addressing, collision avoidance, and star and tree networks with a predefined topology. This MAC layer is independent of the radio and microcontroller, due to the use of a HAL.

For interfacing with sensors, modules have been developed that use I2C, SPI, and UART protocols to interfacing with digital sensors. Using integrated ADCs, analogue sensors can also be interfaced with. These sensor interfaces have also been developed, so that the higher level application logic does not need to be changed if it is running on a 25mm node or a 10mm node.

2.3.2 Plug and play sensor interface

The modules mentioned above can be used to interface with many different commercial sensors. However, the THAWS library also contains code for interfacing with plug-and-play sensors that have been developed in Tyndall. This uses ideas from the IEEE1451.3 [16] standard, but is modified, as it must run on low-power hardware.

In the IEEE1451.3 standard, there a number of TBIMs (Transducer Bus Interface Modules) connected to an NCAP (Network Capable Application Processor). Each TBIM is used as an interface between one or more transducers and the NCAP. In our system, the NCAP is implemented by a Tyndall node. The TBIM is a Cypress PSoC (Programmable System-on-Chip) [17] that can be connected to the node and a number of sensors. This is shown in Fig. 6. Communication between the NCAP and the TBIM is using the I^2C protocol.

A TEDS (Transducer Electronic Data Sheet) provides the ability for TBIMS and individual sensors to identify themselves to the NCAP. The IEEE1451 family of standards define many different TEDS for different purposes. However, they have many fields, which would require a lot of memory to store and handle. This makes them unsuitable for implementing in microcontrollers. Our system uses therefore a minimalist TEDS. There are two types of

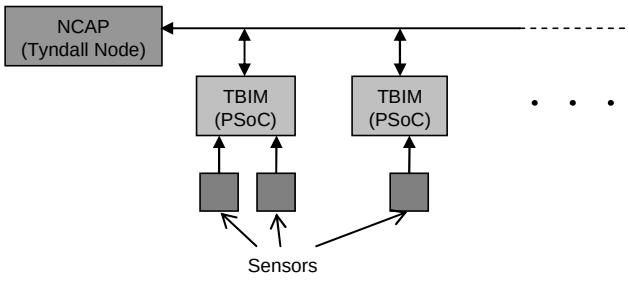


Fig. 6. Plug and play sensor architecture

TEDS: Meta-TEDS, and Sensor-TEDS. Each TBIM has one Meta-TEDS. It says how many sensors are connected and identifies the TBIM board. Each sensor on a TBIM has its own individual Sensor-TEDS, which describes the sensor and its capabilities. Details of the TEDS are shown in Table 2 and Table 3.

Table 2. Meta-TEDS

Field	Bytes	Description
tedsLength	1	Number of bytes in TEDS
tedsVersion	1	Version of TEDS
uuid	2	Universal Unique Identifier
numSensors	1	Number of sensors connected
tedsChk	2	Checksum

Table 3. Sensor-TEDS

Field	Bytes	Description
tedsLength	1	Number of bytes in TEDS
sensorType	1	Type of sensor.
dataType	1	Type of Data.
dataLength	1	Number of bytes in data
warmUpTime	2	Time before 1st reading is valid (Unit = 1 ms)
samplingPeriod	2	Min. time between sampling (Unit = 1 ms)
tedsChk	2	Checksum

For the node to access the sensors there are a number of commands that can be sent to the NCAP:

```
INIT_TBIM
WAKEUP_TBIM
SLEEP_TBIM
READ_METATEDS
READ_SENSORTEDS_n
INIT_SENSOR_n
START_SAMPLING_n
READ_SENSORDATA_n
SLEEP_SENSOR_n
```

For the READ commands, the NCAP will respond with the requested data. The timing of the commands for the sensors should be sent according to the specification in the Sensor-TEDS.

2.3.3 Additional Libraries

In addition to communications and sensing, code has also been developed for timers and buffers, which are common building blocks that make up a WSN application Timers are used to enable low-power sleep modes. The timer can run in this mode, and wakeup the node when required, for example when a sensor reading needs to be taken. Buffers can be used for temporarily storing sensor readings either in RAM for volatile storage or in EEPROM/Flash for persistent storage, where the data will not be lost if the node needs to reset (for example because of a watchdog timeout).

Higher level software modules tie together the hardware interfacing code to produce an application. These are in the form of application templates which can be automatically modified to produce a specific application. For example setting the address of each node in the network, or including the appropriate files for the attached hardware.

The software modules which are core to the THAWS tool have been tested in a real-world deployment. The SmartCoast project [1] has been monitoring water quality (pH, conductivity, turbidity, depth, temperature), using plug-and-play sensors, in the River Lee in Ireland for almost 12 months. In this time the only maintenance required was to periodically clean the sensors, and recharge the battery.

2.4 Application Generation

The part of the THAWS system that is most visible to the user is a wizard tool. This is currently implemented as a console application that asks the user a number of questions about the network, as shown in Fig. 7. This information is then used in the task of code generation.

THAWS has knowledge of which software modules are needed for each node depending on what options the user picks. It also knows how to modify application templates to have the needed functionality. This is done by substituting marked text in the application template with code to create valid C files. THAWS searches through the source file until it finds a variable marked with the prefix "THAWS_VAR_". For example to support tree routing each node is given the required addressing information, e.g. THAWS_VAR_PARENT_ADDR.

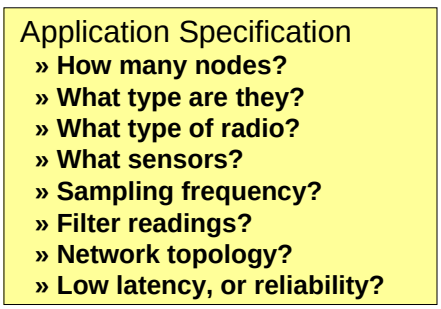


Fig. 7. Questions to generate network specification

The variables can have default values, so that the code can be compiled without using the THAWS tool for testing and debugging purposes.

The use of HALs for interfacing with the radio and sensors simplifies the code generation. Different modules that present the same interface can be linked to without changing any other code. For example the code for any radio can be linked to without any other modifications because they each have the same functions.

The compiling and linking processes, that select which code will be included for each node in the network are all controlled by THAWS to output the required binary files. This is done by generating makefiles [18] with the necessary rules for including the correct code modules and using the appropriate compiler. An example of a generated makefile is shown below. It can define some global values, the processor that is being used, and states which modules should be compiled and linked. A separate makefile is then included that has rules for the compiling and linking.

```
TARGET = thawsRouter
BOARD_HEADER = "nrf905_revB_433.h"

CDEFS += -DUART_BAUDRATE=115200
CDEFS += -DRF_RF_MAX_PAYLOAD_SIZE=10
CDEFS += -DDELAY_USE_CALLBACK_S

MCU = atmega128
F_CPU = 8000000

SRC = $(TARGET).c
SRC += ../../lib/25mm/avr_adc.c
SRC += ../../lib/25mm/avr_delay.c
SRC += ../../lib/25mm/avr_uart0.c
SRC += ../../lib/spi.c
SRC += ../../lib/25mm/avr_i2c_hw.c
SRC += ../../lib/25mm/rf_nrf905.c

# Include rules for compiling,
# linking and programming.
include ../25mm_makerules
```

THAWS also outputs a text file with a formal description of the network. This can be used as an input to the tool to regenerate the same network. It can also be modified to change the functionality.

In addition to the above method of compiling and linking C files separately we have also experimented with outputting all the needed code to one C file. This requires some renaming of module-level variables to avoid name conflicts. For an application that was using 3292 bytes of program memory, this was reduced to 3002 bytes using this single C file, with the same functionality. We believe this to be because the compiler has more opportunity for performing optimizations when the whole program is in a single file.

2.5 THAWS Communication Protocol

The scope of the developing the THAWS tool does not cover researching new communication protocols. We have implemented a simple but reliable protocol that supports tree networks where the topology is known at compile time. Each packet has a common format. The first two bytes are the address of the sender. The next byte is a counter that is incremented once, each time a packet is sent. The sender keeps track of 1 counter for each node that it can send to. The purpose of this is that the receiver can detect if a packet has not been received, if the counter has incremented more than once since the last message it received. The fourth byte states the type of packet (e.g. acknowledgement, sensor data), and the rest of the bytes depend on the packet type. The destination address, and a CRC checksum, are added automatically by the radio hardware. If the destination address of an incoming packet does not match with the nodes own address, the packet is ignored.

The Ember EM2420 radio also supports automatic network ID support, where a node will reject packets not from its own network, and can receive packets send to a network broadcast address. The Nordic nRF905 does not provide such support, so it is implemented manually. Each node can be in two modes: broadcast-accepting, or broadcast-rejecting. Nodes in broadcast-accepting mode will all have the same physical address, so will accept all packets sent to this address. However they still know their *real* address. When using the real address they will only receive packets sent directly to the real address. Nodes can be commanded to switch to using their real address, or the broadcast address.

2.6 Wireless Programming

After the code generation, the binary files can be programmed onto the network wirelessly. This avoids the time-consuming task of manually connecting each node to a PC and programming it. This can be especially difficult if a network has been deployed in a harsh environment, such as marine monitoring.

To support wireless programming each application has the ability to receive a new program binary image and write this to its own program memory. When a complete program has been

written to the memory, the node can restart itself and execute the new program. With the 10mm node there is an external EEPROM which provides persistent storage of the program. When the node is powered up, the microcontroller copies from the EEPROM to an internal RAM that is used solely for program code. As the EEPROM is only accessed at power-up, it can be easily rewritten. For the 25mm node, integrated Flash memory is used for the program code. A special area of this is reserved for a bootloader program. This bootloader program can receive data through the radio and overwrite the rest of the Flash.

3 Analysis

All programming systems must make a compromise between the level of control and ease of use. To have no compromises on possible applications, the application can be developed in a low-level language like assembly, C, or using the TinyOS [4] system. Assemblers and C compilers are available for most platforms, so they are a possible choice for almost all platforms. However the availability of libraries will vary from platform to platform. TinyOS provides a large set of libraries to support energy-aware WSN applications and has libraries for many common tasks. With each these options a lot of care is required to create a reliable application. The programmer must be able to create energy-efficient applications, avoiding, for example, race-conditions, as discussed in the introduction.

Much effort has been spent developing systems that support easier application development for WSNs. One such system is Maté [19], which defines a list of byte-code instructions that can be used to construct wireless sensor network applications. The byte-codes can be sent to a node and will be run by an interpreter on the node. This system has the advantage of fewer lines of code then developing in C, and thus a faster application development time. It is also easier to disseminate new programs into the network, because of the small size of the byte-codes. Limitations of Maté are that the user must be familiar with the byte-codes which look like an assembly language, and also it is assumed that every node will have the same function and design.

SNQPs (Sensor Network Query Processors) [20] are another approach that provide macro-programming of the full network of nodes, from a single declaration. With a SNQP the network can be interfaced with as if it were a database. The user can then enter SQL queries which are interpreted by the network and the desired data is returned to the user.

For example:

```
SELECT nodeid, temperature,
FROM sensor,
WHERE temperature > 20,
SAMPLEPERIOD 10s
```

This will return – from each node where the *temperature* is greater than 20 °C – the *nodeid* and *temperature* reading every 10 seconds. SQL is easier to use than byte-codes as it focuses on what the user wants and not how this should be implemented. However there is still a cost in interpreting the queries, and currently the system is not designed for heterogeneous networks.

Tenet [21] is an architecture for creating two-tiered heterogeneous networks. More powerful nodes are called masters, and less-powerful nodes are motes. With Tenet, the master nodes do most of the work, and this is where the application is programmed by the user. The motes are programmed directly by the master nodes, using tasks, which are sent by the master to the mote. Each task is made up of a string of tasklets, which are simple instructions, for example sampling an ADC channel. The mote performs each tasklet and then, to complete the task, sends a response to the master. Tenet provides some support for some motes having different functionality. A task can contain predicates that must be met before the task will be executed. However this test is done on the mote so the task still has to be sent to every node, which is inefficient. A significant difference between Tenet and our system is the class of the nodes. The mote in the Tenet system is comparable in functionality to the 25mm Tyndall node. The master nodes are a PC or based on the Intel Stargate platform, which has a 32bit, 400MHz processor, and 32MB of program memory. This node is several orders of magnitude more expensive than the 25mm node.

The concept of mobile agents is another approach to developing wireless sensor networks. One implementation is discussed in [22]. In this approach a virtual machine is running on each node. This virtual machine supports agents which can move from node to node to carry out their desired task. For example a tracking agent can follow an event of interest by sending itself to the node it believes to be closest to the event. New agents can be inserted into the network, which is ideal when it is expected that the function of a network will require many changes over its lifetime. However, the agent approach requires sending the agent from node to node, which will be wasteful of radio transmission energy when a smaller packet could be

sent, and more complicated logic on each node to understand the packet.

An approach for building heterogeneous networks is introduced in [23]. This approach involves the use of heterogeneous radios, and has special gateway hardware node that has multiple radios for converting between different physical layer communications, and implements a programming interface that works for multiple radios. It does not however provide any ability for generating heterogeneous code for running on each node as THAWS does. The code for each node must still be manually created.

The THAWS tool that has been presented in this paper allows fast and easy application configuration and rapid deployment of two-tiered heterogeneous and homogeneous sensor networks. THAWS is also easy to use for non-engineers. No knowledge or embedded systems development is necessary. Our system is expressive enough to allow the fast development of any sensor data gathering application. The use of code generation, and not an interpreter, allows for greater efficiency, which is very important on severely constrained systems that must have a long life-time, and where it is expected that the function of the network will not undergo many major changes throughout it's lifetime, The use of C code allows our system to be extended relatively easily to different platforms. It is currently working on Atmel ATmega128 and an 8051 compatible processor, which have completely different tool-chains. The Maté, SNQP, and Tenet , and mobile agent systems all use TinyOS as a base system. Although this gives access to TinyOS's libraries, it also limits their system to TinyOS compatible platforms.

4 Future Work

Our system is presently in an intermediate stage of development. Although it is possible to develop applications that are capable of being deployed, there a number of improvements that can be made. There is a lot of potential to research optimized algorithms for our system. The current library supports a simple communication protocol. However it can be improved through the use of more advanced MAC algorithms, that can enable better energy efficiency as the transceiver can be in a sleep mode for more time. One possible approach is the use of framelets [24], where each message is sent multiple times to ensure that the receiver was awake for at least one message. Such a system can also remove the need for time synchronization or carrier sensing if each node within a cluster repeats its message with a different frequency. The THAWS tool can generate this frequency value at compile time, as it knows the size of each cluster. Otherwise this has to be determined at run-time using radio communication.

Supporting in-network wireless re-programming of networks provides many difficulties. Much research has been done into solving these. Dissemination algorithms for sending the large program code to all nodes in the network without causing network congestion have been examined [25]. Complementary research has been done into only reprogramming only the parts of the program memory that have changed [26]. This reduces the amount of data that has to be sent over the radio. However, due to its heterogeneous nature, our network will provide extra difficulties.

The THAWS system will be validated by using it to develop and configure some real WSN deployments. The Tyndall National Institute has deployments of wearable, environmental monitoring, and medical sensor networks that can be used for testing. This will give valuable information on the ease-of-use and reliability of THAWS.

5 Conclusions

We presented in this paper a new method for fast development and deployment of wireless sensor networks. The sensor networks can be heterogeneous to minimize the cost of the overall network, and also to facilitate non-uniform functionality of each node.

To support this development, the THAWS tool allows macro-programming of the entire network from a single application definition. This definition is obtained from the application developer without the need for detailed knowledge of software programming or embedded systems. This, along with the greater portability is an advantage that has not been seen in other comparable systems.

THAWS has been implemented to use the modular Tyndall nodes, and uses software modules, including plug-and-play sensor support, that have been tested in real-world deployments.

6 Acknowledgements

This work was supported by the Irish Research Council for Science, Engineering, and Technology, as part of the Embark Initiative

References:

[1] B. O'Flynn et al., “SmartCoast: a wireless sensor network for water quality monitoring,” in *Proc. 32nd IEEE Conf. Local Computer Networks*, Dublin, 2007, pp. 815-816.

[2] R. Fernández-Martínez, J. Ordieres, and A. Gonzalez-Marcos, “Low power wireless sensor networks in industrial environment,” *Proc. 12th WSEAS Int. Conf. on Systems*, Heraklion, Greece, 2008, pp. 643-648.

[3] F. Rahman, and N. Shabana, “Wireless sensor network based personal health monitoring system,” *WSEAS Transactions on Communications*, vol. 5, no. 5, pp. 966–972, May 2006.

[4] P. Levis et al., “The emergence of networking abstractions and techniques in TinyOS,” in *Proc. 1st USENIX/ACM Symp. Networked Systems Design and Implementation*, San Francisco, CA, 2004, pp. 2-15.

[5] TinyOS. Available: http://www.tinyos.net/

[6] D. Gay et al., “The nesC language: a holistic approach to networked embedded systems,” in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, San Diego, Ca, 2003, pp. 1-11.

[7] J. Regehr, N. Cooprider, and D. Gay, “Atomicity and visibility in tiny embedded systems,” in *Proc. 3rd Workshop on Programming Languages and Operating Systems*, San Jose, CA, 2006, pp. 4-7.

[8] M. Karpinski and V. Cahill, “High-level application development is realistic for wireless sensor networks,” in *Proc. 4th IEEE Conf. Sensor, Mesh and Ad Hoc Communications and Networks*, San Diego, CA, 2007, pp. 610-619

[9] S. Harte, B. O’Flynn, R. V. Martínez-Català, and E. M. Popovici, “Design and implementation of a miniaturised, low power wireless sensor node,” in *Proc. 18th Euro. Conf. Circuit Theory and Design*, Seville, 2007, pp. 894-897.

[10] S. J. Bellis et al., “Development of field programmable modular wireless sensor network nodes for ambient systems,” *Computer Communications*, vol. 28, no. 13, pp. 1531-1544, Aug. 2005.

[11] J. Barton, A. Gonzalez, J. Buckley, B. O’Flynn, and S. C. O’Mathuna, “Design, fabrication and testing of miniaturised wireless inertial measurement units”, in *Proc. 58th Electronic Components and Technology Conf.*, Reno, NV, 2007, pp1143-1148.

[12] J. Jeong and C. T. Ee, "Forward error correction in sensor networks," UCB Technical Report, May 2003.

[13] Nordic Semiconductor, nRF905 Datasheet. Available: http://www.nordicsemi.com/

[14] S. Harte, B. O'Flynn, R. V. Martínez-Català, J. Buckley, and E. M. Popovici, “Wireless sensor node design for heterogeneous networks,” in *Proc. XXXII Int. Microelectronics and Packaging Conf.*, Pułtusk, Poland, 2008.

[15] V. Handziski et al., “Flexible hardware abstraction for wireless sensor networks,” in *Proc. 2nd European Workshop on Wireless Sensor Networks*, Istanbul, 2005, pp. 145-157.

[16] IEEE Std. 1451.3-2003, IEEE Standard for a Smart Transducer Interface for Sensors and Actuators, 2003.

[17] http://www.cypress.com/psoc

[18] GNU Make, http://www.gnu.org/software/make

[19] P. Levis and D. Culler, “Maté: a tiny virtual machine for sensor networks,” in *Proc. 10th Int. Conf. Architectural Support For Programming Languages and Operating Systems*, San Jose, CA, 2002, pp. 85-95.

[20] J. Gehrke and S. Madden, “Query processing in sensor networks,” *IEEE Pervasive Computing*, vol. 3, no. 1, pp. 46-55, Jan.-Mar. 2004.

[21] O. Gnawali et al., “The Tenet architecture for tiered sensor networks,” in *Proc. 4th Int. Conf. Embedded Networked Sensor Systems*, Boulder, CO, 2006, pp. 152-166.

[22] D. Georgoulas and K. Blow, “Making motes intelligent: an agent-based approach to wireless sensor networks,” *WSEAS Transactions on Communications*, pp. 515-522, March 2006.

[23] F. Graziosi, L. Pomante, and D. Pacifico, “A middleware-based approach for heterogeneous wireless sensor networks” *Proc. 12th WSEAS Int. Conf. on Communications*, Heraklion, Greece, 2008, pp. 52-57.

[24] U. Roedig, A. Barroso, and C. J. Sreenan, “f-MAC: a deterministic media access control protocol without time synchronization” in *Proc. 3rd European Workshop on Wireless Sensor Networks*, Zurich, 2006, pp. 276-291.

[25] S. S. Kulkarni and L. Wang, “MNP: Multihop network reprogramming service for sensor networks,” in *Proc. 25th IEEE Int. Conf. Distributed Computing Systems*, Columbus, OH, 2005, pp. 7-16.

[26] J. Jeong and D. Culler, “Incremental network programming for wireless sensors,” in *Proc. 1st IEEE Conf. Sensor and Ad Hoc Communications and Networks*, Berkeley, CA, 2004, pp. 25-33.