TinyInventor: A Holistic Approach to Sensor Network Application Development

Morten Tranberg Hansen
Dept. of Computer Science
Aarhus University
mth@cs.au.dk

Branislav Kusy
Autonomous System Lab.
CSIRO ICT Center
Brano.Kusy@csiro.au

ABSTRACT

Most of wireless sensor network (WSN) operating systems today provide IPv6 as their standard communication primitive. As a consequence, sensor data can be seamlessly accessed by users through their PCs or mobile devices. Growth of WSNs into this potentially large domain, however, has been limited by the high bar that currently exists in programming WSNs. In addition to different programming abstractions and language constructs, embedded systems use development environments that are distinctively different and often lag behind PC development tools.

We present TinyInventor, an integrated development environment for WSN applications that aims to resolve both of these problems. It provides drag-and-drop visual programming language Open Blocks that is easy to use for novice programmers. TinyInventor also unifies development of mote and PC code by using cross-platform programming abstractions, namely thread based execution models and IPv6 communication primitives. We demonstrate through an application example that TinyInventor is both simple to use and powerful in expressing complex applications.

1. INTRODUCTION

The main barriers in development of WSN applications have been the severe resource restrictions of the sensor devices and the implications of these restrictions on the software architecture. For example, link layer was the unifying communication abstraction for WSNs for almost a decade. Despite Internet Protocol (IP) being used by millions of computers on the Internet, it was thought to be too resource demanding for the embedded wireless sensors. However, driven by the vision of Internet of Things, the development of the 6lowpan standard [1] enabled IPv6 to become the de-facto communication abstraction for WSN. Similarly, the resource restrictions concerns that have led to highly optimized interrupt-driven language design and operating systems, have given way to design concepts that simplify application development for non-domain experts. For example, developers can decide to sacrifice code execution efficiency and use efficient preemptive multi-threaded [2] or a familiar desktop [3] environments. Despite this progress, it is still not possible for a non-embedded developer to develop a complete embedded system without some form of area expertise and knowledge about the target platform. Moreover, PC applications that access sensor data are developed separately from WSN applications and the reconciliation of source code running on the two platforms further complicates the development and maintenance of WSN applications.

To ease the task of developing WSN applications we propose TinyInventor, which brings years of research with educational programming languages, to WSNs. TinyInventor is an open-source developing environment which unifies programming abstractions, such as threads and internet communication operations, across platforms. TinyInventor builds on top of a visual programming language, called Open Blocks, that Google selected for the App Inventor for Android. Open Blocks framework abstracts language and OS features of the underlying hardware platform away by presenting them through a graphical drag-and-drop block programming model easily understood secondary school students [4]. One of the advantages of visual programming is that the distracting and annoying implementation details can be often determined implicitly by the framework from the context or by setting a simple parameter.

After reviewing related work in Section 2, we present the modular architecture of TinyInventor environment, describing both the visual programming language and compilers that generate platform specific code from the visually connected blocks. We show how the language can be extended with new features without requiring code modifications. Section 4 then demonstrates a case study of a TinyInventor application running on a sensor node and a PC. We show that even though the syntax of TinyInventor is simple, it is powerful enough to express a complex distributed application. More specifically, both the PC and the mote side programs use a number of threads to support different functionality, use two-way cross-platform IPv6 communication, and are implemented in a single environment.

2. RELATED WORK

The visual programming language that TinyInventor uses was pioneered by almost a decade of research with Scratch [5], an educational programming language that allows people of any age to experiment with programming by putting together blocks to control images, music, and sounds. Inspired by the same visual block programming approach, StarLogo

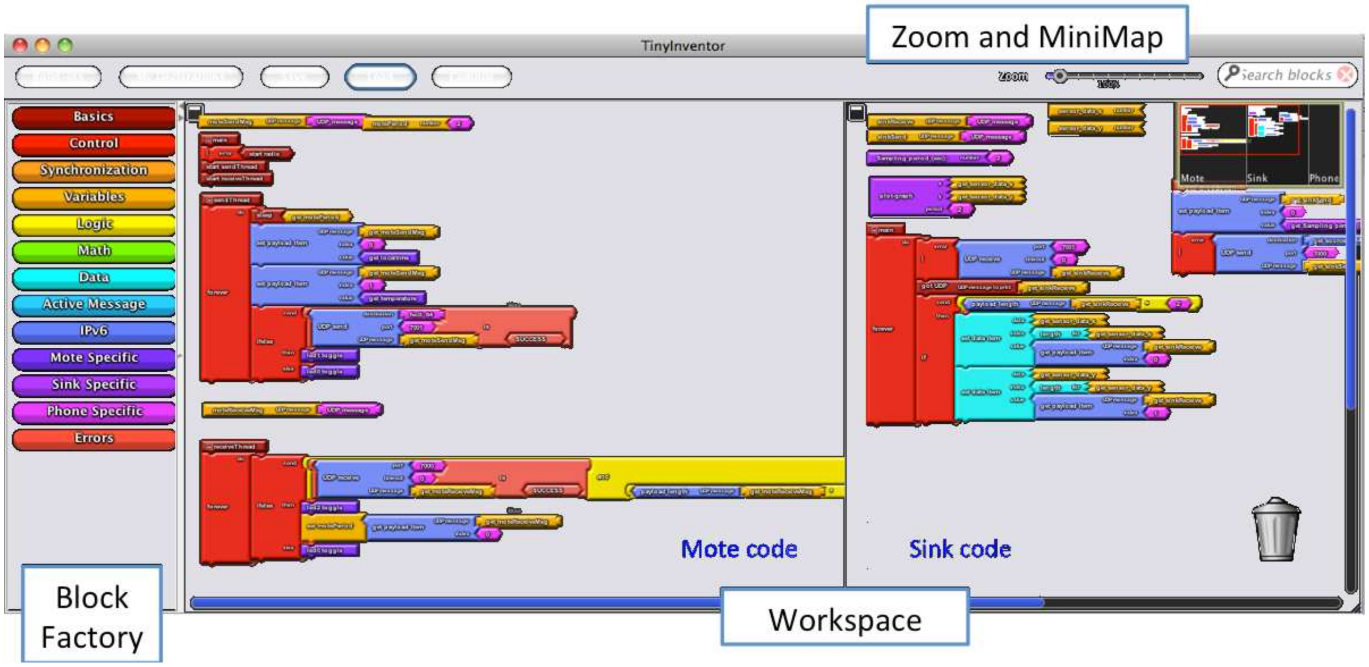


Figure 1: Graphical user interface of TinyInventor. The main components are the block factory and the workspace. Functional blocks are dragged from the block factory to the workspace to form applications. Minimap helps to manage complex applications.

TNG [4] provides an agent-based simulation language designed for students to model the behavior of decentralized systems. These visual programming environments were designed to make programming more accessible to novices and hence were designed with narrow intentions for their users and applications.

The Open Blocks framework [6] has taken a step forward and enabled developers to design their own visual block programming environment. Google found the visual programming concepts along with the extensibility of the framework attractive and used Open Blocks as a basis for their recently released Google App Inventor for Android devices.

Numerous approaches have been proposed to simplify the task of programming sensor networks and to bridge the gap between WSN and more traditional development environments. These efforts include pre-emptive thread libraries [2], declarative programming languages [7], generic tasks [8], and macroprogramming where a developer can write a single program for an entire WSN [9]. More recently, Java virtual machines have been implemented on the sensor nodes [3]. The problem with these approaches, is that they still require the developer to write code in multiple programming languages and development environments for different platforms.

Alternatively, visual tools have been developed to aid in the development of embedded applications. These include TinyOS developing environments [10, 11, 12] which on top of providing the developer with syntax validation, code navigation and code completion, also provide an visual wiring overview. Other development environments [13, 14] take an extra step and let the developer visually wire existing TinyOS components together. However, these approaches only give the developer a graphical overview of the existing applications, or allow them to wire together existing WSN components, rather than allowing them to visually program the entire application.

In contrast, TinyInventor exposes the existing source code components at a higher level, abstracting out the interfaces of the underlying operating system. Most of TinyInventor blocks are OS independent and can be visually combined with each other through standard programming language features such as control statements, conditions, or operators, to form a customizable sensor network application.

3. TINYINVENTOR

TinyInventor is an integrated development environment for WSN applications that supports writing programs for heterogeneous platforms, such as sensor nodes, personal computers, or mobile computing devices (see Figure 1). Traditionally, native programs running on the different platforms require different styles of programming and thus are typically written and maintained separately.

TinyInventor defines a higher level graphical programming language that enforces a unified set of programming abstractions, such as threads and IPv6 network connectivity, across all supported platforms. Even though such requirement may seem overly constraining, enforcing the unified abstractions need not necessarily come at the expense of efficiency. TinyInventor provides compilers that generate native source code for each supported platform and thus can optimize the generated code for performance. The advantage of using a single cross-platform environment is that shared portions of the code, such as message structure definitions or constants, are reused naturally across the platforms.

TinyInventor programs are defined as a collection of functional blocks where some blocks are running on sensors and some on personal computers. The functional blocks are ma-

```
<BlockGenus name="leds-set" kind="command"
    initlabel="set leds" color="128 30 255">
  <description>
  <text>
     Set the led's to a certain value.
  </text>
  <arg-description n="1" name="number">
     The number value to set the led's to.
  </arg-description>
  </description>
  <BlockConnectors>
  <BlockConnector label="number" connector-
     kind="socket" connector-type="number"/>
  </BlockConnectors>
</BlockGenus>
```

Figure 2: Example of a block definition.

nipulated by users through clicks, drags, and drops and are connected to each other in a specific type-constrained way. The ease and speed with which WSN programs can be created supports rapid prototyping of WSN applications as well as makes programming accessible to larger audiences of non-domain experts.

We first describe the main components of TinyInventor development environment, then provide more details on the types of function blocks that our visual programming language defines, and finally demonstrate how the native compilers generate platform specific code from functional blocks.

3.1 TinyInventor Workspace

TinyInventor is an enclosing application for Open Blocks framework and consists of the following main components: workspace, block factory, and minimap (see Figure. 1).

Block factory is the space where customized Open Block functional blocks are accessed. The blocks are defined by the Language Definition XML file (see Sec. 3.2 for block examples).

Blocks are dragged and dropped by application developers to the *Workspace*. Once in the workspace, blocks can be connected to each other to form a TinyInventor application. The TinyInventor workspace is a split workspace, with an area reserved to each of the supported platforms. We currently support mote and PC application code. Thus, TinyInventor is a holistic cross-platform environment where the entire system is viewed as a single interconnected application instead of multiple separate entities. The placement of blocks in the workspace specifies the platform they are compiled for.

Minimap and zoom bar are useful in managing complexity of distributed applications. Additionally, TinyInventor provides buttons to load and save TinyInventor applications, as well as a button to compile application to the native source code for each of the platforms.

3.2 Open Blocks

The first step in building a programming environment is to design its programming language. In our case, the programming language consists of visual blocks that connect to each other in a specific way. The selection of visual blocks is influenced by the target group that will use the tool to develop applications. We determined our target group to be people familiar with imperative programming languages that have limited experience in development of WSN applications.

Blocks are defined by specifying the Language Definition file in an XML format. Block definitions primarily consist

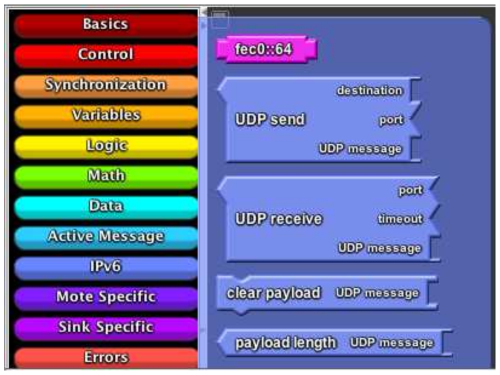


Figure 3: OpenBlock blocks for IPv6 communication. Due to the space reasons, we omitted blocks to get and set payload items and a UDP source.

of a type, a number of connectors, a label, and a color (see Figure 2). The Open Blocks framework defines four types of blocks: data, procedure, command, and function blocks. Data blocks define variables and constants, procedure blocks defines callable procedures, command blocks define statements that can be executed in order, and function blocks define functions with a return value.

Blocks can be connected with each other through typed connections. Two types of connectors exists: plugs and sockets. A plug defines "value-of" relation between blocks, while a socket defines "parameters-of" a block relation. A block can only have one plug but it can have multiple sockets.

The Open Blocks type system supports an unlimited number of connector types, by only a limited number of connector shapes which visually differentiate connectors of different types. By default, the type system makes sure that only plugs and sockets of the same type can be connected. The type system, however, can be extended through link rules to override the default behavior and allow/disallow blocks to be connected based on their context.

TinyInventor blocks are organized in two groups: generic and specialized blocks. The generic blocks consist of general language, thread and communications primitives that are non-platform specific and hence can be compiled into binaries for any supported platform. More specifically, this includes variables, control statements, logic and number operators, thread control and definitions, and UDP send, receive, and data access functions. The specialized blocks correspond to functions specific to each supported platform and include sensors, actuators, and user interfaces. Examples of IPv6 communication blocks are shown in Figure 3.

3.3 Abstractions

Even though the TinyInventor is independent of the underlying operating system, it does require the underlying system to support a few core programming primitives, to ensure the consistency of programs across different hardware platforms. In addition to generic imperative programming concepts (such as control loops), the main two primitives that we require are support for preemptive threads and IPv6 communication.

We chose the preemptive threads to be our basic programming abstraction because they provide an intuitive way of programming and are supported in most modern day programming languages and operating systems. Even though event driven execution model has been traditionally favored over threads in WSN community for performance reasons, most modern WSN operating systems support threads in some form [2, 15].

We also decided to enforce IPv6 communication in all supported platforms to support seamless cross-platform communication within the deployed WSN applications. Similarly to threads, IPv6 is widely supported in modern programming languages, and through the 6lowpan standard [1], IPv6 has become a de-facto standard communication primitive in WSNs today.

3.4 Compilers

TinyInventor implements a block compiler for each supported platform. The compiler works in two stages: first, a platform specific code is generated from TinyInventor program, and second, the platform code is compiled to a platform specific binary using the appropriate platform toolchain. Here we focus on the first stage compiler and provide details on how the code is generated using information provided by the blocks.

Similarly to the Language Definition XML file for Open Blocks, TinyInventor compilers use XML files to define the source code generation process from each of the functional blocks. Any new features, in terms of blocks, can then be added to TinyInventor and its applications without code modifications of TinyInventor itself.

The compilers are built around components which are groups of connected blocks. The generated code is in general parametrized by any input parameters that the functional blocks define, for example, *UDP send* shown in Figure 3 is parametrized by a destination, a port, and a UDP message to be transmitted. Components are comprised of hierarchically connected blocks, each component having a single top level block. Blocks are sequentially connected to the top-level block. Section 4 has numerous examples of blocks and their connections. A component can produce platform specific static code which is to be directly part of the final platform code, or nested code which can become part of its parent component's static code.

TinyInventor first stage compiler also works in two phases. In the first phase, the compiler first runs through the block hierarchy of the workspace in a depth-first manner and creates a representative component for each block. Components, representing child blocks in the hierarchy, return their nested code to their parents upon creation, which means that at the end of the run, all code is stored as a static code in some component. Once this run is completed, the compiler commences the second phase. It fetches the static code from all the created components and concatenates them in the resulting platform program, as dictated by block connections.

TinyInventor currently supports two compilers: a nesC compiler that generates TinyOS code for sensor nodes, and a Python compiler that generates application code for PCs.

3.4.1 nesC Compiler

The nesC compiler compiles TinyInventor blocks into a TinyOS program. Three files are generated: a Makefile, a configuration file, and a module file. The nesC Language Definition file defines nesC code for each TinyInventor block. The final program is then produced by stitching together the static code from all blocks. The code generated by the nesC compiler can be deployed on all supported TinyOS hardware platforms.

Due to the peculiarities of TinyOS and its nesC language, we had to define five different static code types: wiring, interface, declaration, initialization, and module code. The wiring code consist of code that needs to be written into the nesC configuration file. The interface code defines the interfaces used by the module file. The declaration code contains declarations of variables and functions that needs to precede the module implementation. The initialization code contains variable initializations or any other code that needs to be executed during TinyOS's software boot process. Finally, the module code contains functions, thread definitions, commands, and events that need to be defined in the module.

3.4.2 Python Compiler

The Python compiler is conceptually simpler than the nesC compiler as it requires fewer types of static code. We only define function code, declaration code, and main code. The function code contains python code representing functionality of the blocks, the declaration code contains declarations of variables, classes, or threads that are used by the function code, and the main code contains initializations required at the program startup.

In addition to basic blocks, we provide a graphical framework for python applications built on top of Tk GUI toolkit which is the standard python GUI. The most basic python application opens a window that logs printf messages over the runtime of the application. Printf messages can be printed from the application using a printf block. The basic GUI further contains space for adding buttons, generated by dragging a button block to the workspace. Buttons work similarly to threads—their code gets executed when a button is clicked. GUI also supports text fields, generated by dragging a text-entry block to the workspace. Text field contents are accessed similarly to the integer variables or strings. See Figure. 7 for an example of *"Change period"* button and *"Sampling period (sec):"* text-entry.

Another useful python GUI extension that TinyInventor provides is a plot-graph block for easy plotting of received sensor data. We use the matplotlib python library to allow python applications to plot arbitrary graphs. We set matplotlib in the interactive mode and refresh the graph periodically, to obtain graphs that dynamically update when new sensor data is received. See Figure 8 for a demonstration of the python GUI.

4. CASE STUDY

Our case study application shows a simple TinyInventor WSN application where a number of motes periodically sends UDP packets with temperature data to a PC. Note that due to the use of IPv6 communication the PC does not have to be co-located with the motes. The temperature data from the motes is timestamped with the nodes localtime, so upon receiving sensor data from a mote the PC plots the received data over time graph, using the nodes localtime. We also show how TinyInventor enables communication in the other direction by letting the PC application set the future sensing period of the node it received the last data packet from.

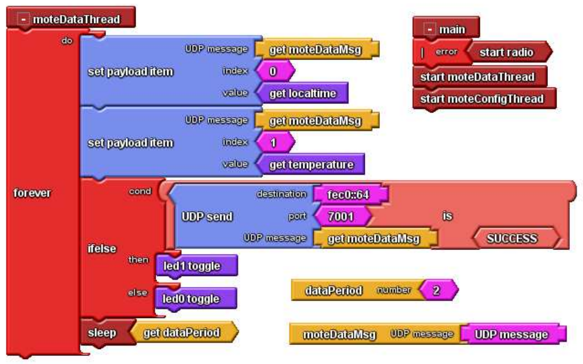


Figure 4: TinyInventor example application, mote code part 1. Contains the main thread and the data thread which periodically samples the motes temperature sensor and sends it to a PC.

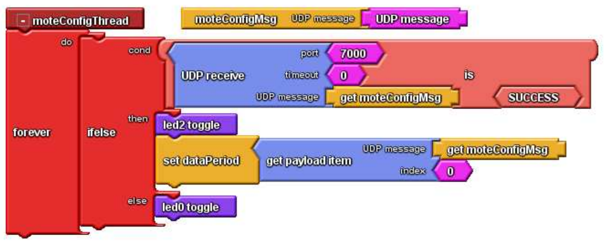


Figure 5: TinyInventor example application mote code part 2. Contains a config thread which listens for configuration message which can change the sampling period.

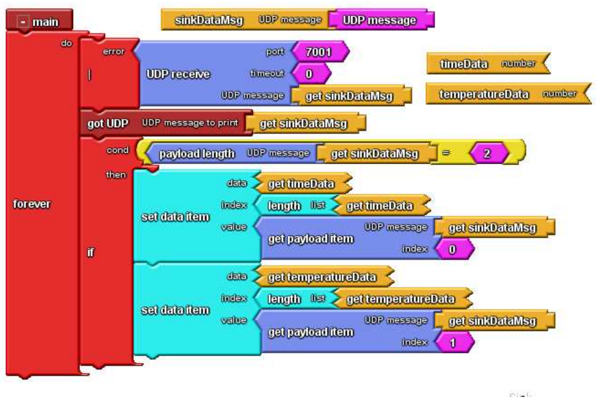


Figure 6: TinyInventor example application sink code part 1. Contains a data thread at the PC that listens for incoming messages containing sensed data which is then stored locally.

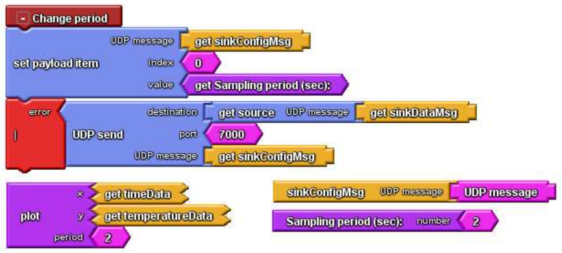


Figure 7: TinyInventor example application PC code part 2. Contains a button handler, a plot, and a text-field used to set the motes sampling period.

4.1 Design

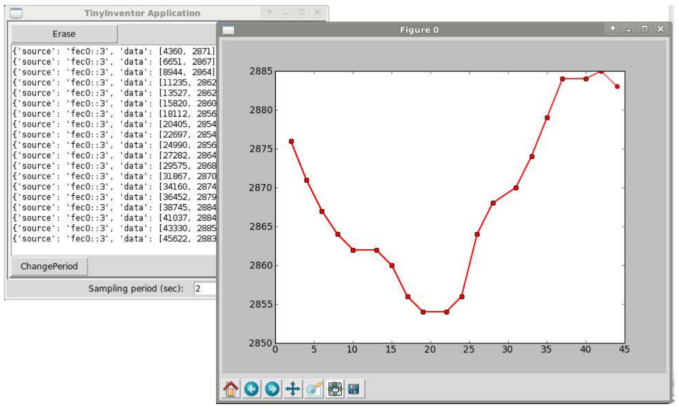
The design of the WSN application is divided into two parts: the mote side and the PC side. Figure 4 shows the main thread and the data collection thread of the mote side. First the `main` thread, which is started after a mote is booted, starts the radio and then the data collection thread `moteDataThread` and the config thread `moteConfigThread`. The data collection thread is also seen in Figure 4 which, before sending the UDP message `moteDataMsg` to the PC on port 7001, sets the payload of it to contain two integers: the nodes localtime and the temperature from the nodes temperature sensor which returns a temperature value in Celsius. The thread uses the UDP send function which sends the given message to the given IPv6 address and port. After a packet is sent, the motes leds are toggled accordingly to show whether it was successful. Note that UDP communication does not guarantee reliable delivery of the packet. When the send is complete, the data thread sleeps for a number of seconds according the the `dataPeriod` variable. The time a mote sleeps defaults to $2s$.

Figure 5 shows the configuration part of the mote code. This consist of a single thread, started from the main thread, which listens for UDP packets on port 7000 and sets the `dataPeriod` of the mote to the first integer in the payload. The mote's leds are toggled according to the outcome of the receive call. The blocks in the mote code are mostly generic and can be used across platforms, except for the leds, temperature and localtime commands, which are only available on motes.

Figure 6 shows the main part of the PC code. It listens to UDP messages on port 7001. Upon reception of a message it prints and checks whether the message contains two integers. The data history lists of times and temperatures are then updated, using the two variables `timeData` and `temperatureData`, respectively. Note that the list access cannot be out of bounds: this is handled internally by TinyInventor by increasing the length of the list before accessing the list.

Figure 7 shows the GUI part of the PC code. As opposed to the PC code shown in Figure 6 this figure contains PC specific TinyInventor blocks in the form of a plot and a text field represented by the block with the label "Sampling period (sec):". The figure does not contain any threads. Instead `Change Period` is a button event handler which is a generic component available on both motes (the Tmote Sky platform which we use has one user button [16]) and PCs. The inclusion of the handler adds a button to the PCs GUI. When the button is pressed, the code sets the payload of the `sinkConfigMsg` message to the integer current stored in the "Sampling period (sec):" text-field and sends the message to the mote.

Figure 7 also shows a plot component which represents a plot in the GUI that is updated with the data stored on the `timeData` and `temperatureData` list every two seconds.



```
{'source': 'fec0::3', 'data': [4360, 2871]
{'source': 'fec0::3', 'data': [6651, 2867]
{'source': 'fec0::3', 'data': [8944, 2864]
{'source': 'fec0::3', 'data': [11235, 2862
{'source': 'fec0::3', 'data': [13527, 2862
{'source': 'fec0::3', 'data': [15820, 2860
{'source': 'fec0::3', 'data': [18112, 2856
{'source': 'fec0::3', 'data': [20405, 2854
{'source': 'fec0::3', 'data': [22697, 2854
{'source': 'fec0::3', 'data': [24990, 2856
{'source': 'fec0::3', 'data': [27282, 2864
{'source': 'fec0::3', 'data': [29575, 2868
{'source': 'fec0::3', 'data': [31867, 2870
{'source': 'fec0::3', 'data': [34160, 2874
{'source': 'fec0::3', 'data': [36452, 2879
{'source': 'fec0::3', 'data': [38745, 2884
{'source': 'fec0::3', 'data': [41037, 2884
{'source': 'fec0::3', 'data': [43330, 2885
{'source': 'fec0::3', 'data': [45622, 2883
```

Sampling period (sec): 2

Figure 8: A screenshot of the resulting example application running. Incoming data messages are printed to the screen and plotted in a graph.

4.2 Result

The TinyInventor application compiles to TinyOS nesC code for the motes and Python code for the PC. The nesC code will have to be compiled with the TinyOS toolchain and uploaded to the desired number of motes while the Python code can be executed directly using the Python interpreter.

We deployed the application on one mote next to an IP base station and ran the PC application. Figure 8 shows the resulting PC application with two windows: the TinyInventor main window and the graph plot window. The main window consist of a text window showing printed text, the period change button, and the text-field used as input when setting the period of a mote. The graph window plots the received data over time. We held a hand around the temperature sensor on the mote to cause the temperature changes seen in the graph.

5. CONCLUSION

We presented TinyInventor, an open-source cross-platform development environment that builds on de factor WSN programming and communication primitives to enable a developer to build cross-platform complex applications. We showed how TinyInventor utilized the Open Blocks visual programming language to do this in a simple and intuitive way that even non-experts would understand.

In the future we expect to improve TinyInventor link and compile checks, add cross-platform data storage features, and another compiler for mobile hand-held Android devices.

6. ACKNOWLEDGEMENTS

Thanks to Kevin Klues for valuable discussion throughout the development of TinyInventor, which helped us turn TinyInventor into a cross-platform development environment instead of just yet another programming tool for embedded devices.

7. REFERENCES

[1] Gabriel Montenegro, Nandakishore Kushalnagar, Jonathan Hui, and David Culler. RFC 4944 – Transmission of IPv6 Packets over IEEE 802.15.4 Networks. Technical report.

[2] Kevin Klues, Chieh Jan Mike Liang, Jeongyeup Paek, E. Răzvan Musăloiu, Philip Levis, Andreas Terzis, and Ramesh Govindan. TOSThreads: thread-safe and non-invasive preemption in TinyOS. SenSys '09, pages 127–140, 2009.

[3] Niels Brouwers, Koen Langendoen, and Peter Corke. Darjeeling, a feature-rich VM for the resource poor. SenSys '09, pages 169–182, 2009.

[4] Eric Klopfer, Ricarose Roque, Wendy Huang, Daniel Wendel, and Hal Scheintaub. The Simulation Cycle: combining games, simulations, engineering and science using StarLogo TNG. *E-Learning*, 6(1):71+, 2009.

[5] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The Scratch Programming Language and Environment. *Trans. Comput. Educ.*, 10, November 2010.

[6] Ricarose V. Roque. *OpenBlocks : an extendable framework for graphical block programming systems.* PhD thesis, Massachusetts Institute of Technology, 2007.

[7] David Chu, Lucian Popa, Arsalan Tavakoli, Joseph M. Hellerstein, Philip Levis, Scott Shenker, and Ion Stoica. The design and implementation of a declarative sensor network system. In *SenSys '07*, pages 175–188, 2007.

[8] Omprakash Gnawali, Ki-Young Jang, Jeongyeup Paek, Marcos Vieira, Ramesh Govindan, Ben Greenstein, August Joki, Deborah Estrin, and Eddie Kohler. The Tenet architecture for tiered sensor networks. In *SenSys '06*, pages 153–166, 2006.

[9] Timothy W. Hnat, Tamim I. Sookoor, Pieter Hooimeijer, Westley Weimer, and Kamin Whitehouse. MacroLab: a vector-based macroprogramming framework for cyber-physical systems. In *SenSys '08*, pages 225–238, 2008.

[10] William P. McCartney and Nigamanth Sridhar. TOSDev: a rapid development environment for TinyOS. SenSys '06, pages 387–388, 2006.

[11] Nicolas Burri, Roland Flury, Silvan Nellen, Benjamin Sigg, Philipp Sommer, and Roger Wattenhofer. YETI: an Eclipse plug-in for TinyOS 2.1. SenSys '09, pages 295–296, 2009.

[12] Jun B. Lim, Beakcheol Jang, Suyoung Yoon, Mihail L. Sichitiu, and Alexander G. Dean. RaPTEX: Rapid prototyping tool for embedded communication systems. *ACM Trans. Sen. Netw.*, 7, August 2010.

[13] P. Volgyesi and A. Ledeczi. Component-based development of networked embedded applications. pages 68–73.

[14] Elaine Cheong, Edward A. Lee, and Yang Zhao. Viptos: a graphical development and simulation environment for TinyOS-based wireless sensor networks. SenSys '05, page 302, 2005.

[15] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. *Local Computer Networks, Annual IEEE Conference on*, 0:455–462, 2004.

[16] Tmote Sky Low Power Wireless Sensor Module. http://sentilla.com/files/pdf/eol/tmote-sky-datasheet.pdf.