Remote Incremental Adaptation of Sensor Network Applications

Waqaas Munawar, Olaf Landsiedel, Muhammad Hamad Alizai, Klaus Wehrle
Distributed Systems Group
RWTH Aachen
firstname.lastname@rwth-aachen.de

***Abstract*—Wireless Sensor Networks (WSNs) are deployed for long periods of time, during which a need often arises to dynamically reprogram or retask them. An array of solutions has been proposed to this effect, ranging from full image replacement to virtual machines. However, the capabilities of TinyOS – the current state of the art in sensor node operating systems – are still limited to full image replacement. TinyOS based applications have a modular architecture but during compilation this modularity is lost resulting in a statically linked system image.**

In this work we extend TinyOS to allow dynamic exchange of components in WSN applications by conserving their modularity during the compilation process. This generates the possibility of incremental adaptation of sensor nodes' behavior through partial code replacement. The designed system does not require any alterations in the existing user interfaces, remaining transparent to the user. The evaluation shows that our approach imposes almost no performance overhead for loaded application while keeping a smaller memory footprint than other comparable solutions.

I. Introduction

Sensor nodes could be located far from the networked infrastructure and easy human access [1], [2], [3]. Based on the evolving analysis or the environment the software application of the sensor network often requires adaptation through introduction of new code. Manually collecting all the nodes to apply a software update is dangerous in some of the situations [2], [3] and tedious in others [1], [4]. Therefore, remote software reconfiguration – even if a rare activity as compared to the application operations – becomes a highly desirable feature.

Remote retasking of sensor nodes is mainly challenged by three constraints; limited energy, limited processing power and limited available onboard memory. kilobytes. Moreover, the major hurdle in the way of mainstream adoption of WSNs remains the steepness of the associated learning curve. Considering these constraints, an ideal solution to dynamically update a nodes functionality would be the one that optimizes the energy usage, has a reduced memory footprint, and does not require any alteration in the existing user interfaces.

Our contribution in this work is design and implementation of a solution for efficient dynamic adaptation of TinyOS based applications running on sensor nodes. The proposed system works in two phases; firstly, the existing components of an application are solitarily compiled into ELF objects. Solitary generation of the software components ensures that the component structure of the TinyOS application is preserved during the compilation process. In the second step, these components are transferred to the sensor node and integrated into the running application. To allow this, a thin node runtime is designed that includes a runtime linker and enables dynamic exchange of components. Runtime dynamic linking allows the reduction of energy-toll incurred in communication by limiting the size of required communique to that of an updated component only. The presented system is tightly coupled with TinyOS, reusing its components and interfaces hence easing the adoption process. Moreover, the designed system does not necessitate any change in the existing code repository of TinyOS hence, remaining transparent to the user of the system.

II. Related Work

The existing approaches to tackle the issue of retasking a sensor network can be classified into three main areas;

Full image replacement e.g. Xnp [5], Deluge [6] and differential updates [7], offer very fine grained control on the possible reconfigurations but are quite wasteful in terms of energy-cost of communication.

Virtual machine e.g. Maté [8], perform inversely; they optimize the energy-cost of communicating the new functionality but the control offered on the possible reconfigurations is very coarse grained, moreover the trade-offs between interpreting code and executing native binaries suggest the use of the latter for long-running systems.

Dynamic operating systems e.g. Contiki [9], SOS [10] and FiGaRo [11], provide benefits of both of the former categories however, in most cases these solutions have followed a clean slate approach which has hindered the wide scale adoption. Two notable exceptions are FlexCup [12] and TOSthreads [13] that are built on top of seasoned TinyOS repository. FlexCup offers dynamic adaptation for TinyOS based applications but lacks the support for new extensions to nesC and employs nonstandard tools. TOSthreads library and its associated linker follow a polling based approach for kernel to application calls instead of nesC's more suited, event based approach and introduces a new interface for users, rendering it difficult to adopt.

III. Design

TinyOS based applications consist of large number of wired nesC components which communicate with each other via interfaces. This component based structure results in a very modular architecture of TinyOS applications. However, once compiled, this modularity is lost. The NCC compiler when transforming nesC to C mashes up the component structure of the input to make the output conform to the semantics of C. The output is a single monolithic C source file to be compiled by the respective toolchain. We alter this process of compilation by isolating subsets of an application's constituent nesC components and compiling them into ELF files. The resulting files collectively enclose all of the constituent components of the application. These files are transferred to the sensor node which links them together and loads them in the program memory to form the executable binary image again.

The solution we present consists of two main components; *Isolater* to isolate a single or an integrated group of nesC components and compile them into an ELF object. Second, *TinyMan*, a runtime ELF linker executing at sensor node and responsible for integrating the ELF objects to form the executable binary image to be loaded in program memory. The working of both of these are detailed as follows.

A. Isolater

The *Isolater* functions by compiling parts of a single TinyOS based application separately into ELF files. It executes on PC (host) and utilizes the binary component generation feature of the NCC compiler. This feature was introduced primarily to provide better commercialization support as binary components can be used and distributed without their corresponding source code. We utilize this feature to isolate and compile a single or a set of interconnected nesC components belonging to a TinyOS application. Compiling parts of an application solitarily causes loss of code optimization possibilities as well as introduction of ambiguities which either lead to an incorrect decision by the compiler or result in a compile time error.

The main issues faced during isolation of nesC component are compile-time operators, default events and generic components and interfaces. All of these are caused due to the non-availability of information hidden in those parts of the application that are not being compiled at the moment. *Isolater* provides this missing information using the additional input in the form of a nesC configuration. This additional input consists of two main parts for each component to be isolated; a component-wrapper and an application side place holder. The component wrapper ensures that the component being isolated is provided the required knowledge of the rest of the application for the correct compilation. Likewise, the application side place-holder ensures that the application gets the required knowledge about the component which will be linked in at runtime. During this process, the actual source code of both the application and its component is not changed. This allows complete 'recycling' of existing TinyOS based applications and seamless integration of the system into the existing TinyOS skeleton, thereby remaining totally transparent to the application developer. In the next section we discuss the issues related to dissemination and integration of the generated components.

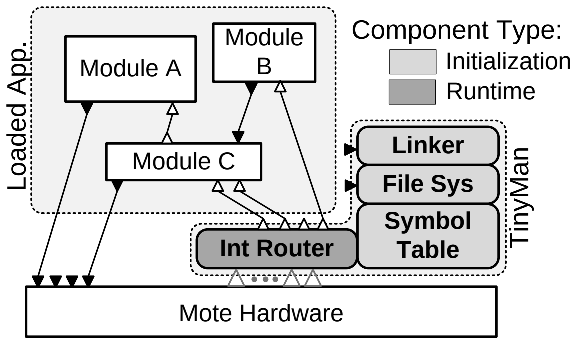

Fig. 1. Architectural elements of *TinyMan*. Only runtime components are active during the normal execution of a loaded application. Linker and File System APIs are provided by the kernel to support the application to kernel calls.

B. TinyMan

After the compilation of components, the next step consists of their dissemination and integration in the sensor application executing on the sensor node. The dissemination in the network takes place through the use of the Deluge data dissemination protocol. Other existing protocols can be employed as well and these, since treated as part of loaded application, can also be replaced remotely on runtime. This design approach makes data dissemination a concurrent process along with the normal execution of loaded application resulting in reduced downtime due to an update in progress.

After the required modules and an update command have been received the data dissemination protocol invokes the linker to integrate the received modules and place the new binary image in program memory of the sensor node. To accomplish this, the node runtime consists of the following main components:

- **File System:** provides the storage capability for large data elements such as received ELF modules.
- **Linker:** responsible for linking the the new ELF modules and placing them in code memory.
- **Global Symbol Table:** As the linking is done among dependant ELF modules, this component holds the symbols offered by one module that are needed by some other ELF module.
- **Interrupt Router:** Unlike the compile-time linker, the implemented runtime linker does not have the flexibility in placement of code segments. Therefore an interrupt router is implemented to route the interrupts to the inappropriately placed interrupt service routines.

These components in relation to a loaded application are shown in Figure 1. Apart from the interrupt router, all of the other components are inactive during the normal execution of application. This helps in minimizing the runtime performance impact due to *TinyMan*. The linker and the file system's APIs

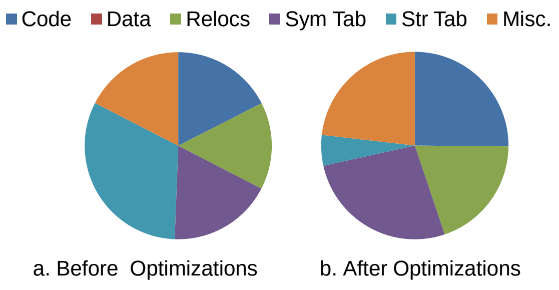

Fig. 2. Results of optimizations shown as proportionate change in sizes of different segments of the resulting ELF file.

are available to the loaded application through the global symbol table, which can be used for storing new or updated ELF modules and then integrating them into the existing application.

C. Optimizations

The ELF format, though a widely used standard, is not optimized for the low power processors. In the ELF libraries, compiled from the NCC compiler's generated code, the bulk of the contribution in size comes from the string table which holds the names of all the symbols in the ELF file. These names often tend to be quite long – about 80 characters each. We decrease the size of the symbol names down to 3 characters by replacing each symbol name with a unique string based on an alphanumeric counter. The mapping of the replaced names is stored in a database which can be used later when recompiling parts of the application. This procedure results in: (1) significant reduction in the size of ELF file, (2) reduction in the size of symbol table and (3) reduction in number of string comparison operations. The average proportionate reduction in the size of string tables for ELF files of the Blink application is shown in Figure 2.

The second set of optimizations that result in significant resource savings is applied to the symbol table which is used during the process of linking. We split the symbol table into two sub-tables; one containing static core symbols and the other filled dynamically from the symbols included within the ELF files being loaded. The static part is created at compile time and placed in ROM in a sorted order allowing binary search among the symbols. This results in a quicker hence more energy efficient linking process.

These two sets of optimizations together cause a major improvement in processing speed, resulting in energy savings of up to 66% when compared to the original ELF.

IV. Evaluation

We evaluate the proposed system along the lines of major constraints faced in WSNs i.e. energy consumption, processing requirements and memory utilization.

A. Energy

To evaluate the per-node energy consumption caused due to a network wide reconfiguration we devise a simple energy

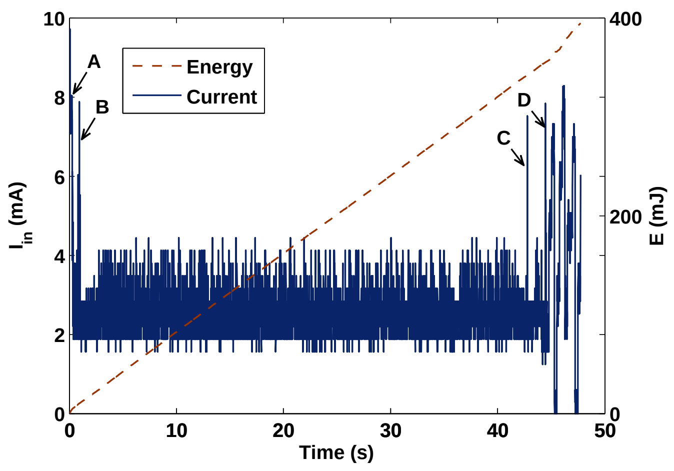

Fig. 3. Current draw and energy utilization during processing and loading of Blink application at telos platform. The peaks are generated by turning the onboard LEDs on simultaneously. A to B: saving the modules in external flash, B to C: linking the modules, C to D loading the modules in program memory, from D onwards: executing the application.

model and calibrate it using the readings taken empirically. The final results are compared against Deluge [6] – the widely used in-field code replacement tool and protocol.

We model the energy cost of reconfiguration as;

$$E \quad = \quad E_{Tx} + E_{Rx} + E_P$$

Where,

E_{Tx}	is the energy consumed in transmitting an update
E_{Rx}	is the energy cost of reception, and
E_P	is the energy consumed during related processing.

We assume that each node receives the update, propagates it and then processes it to reconfigure itself. This might not be accurate for bordering nodes and those nodes which do not need to propagate further because of their close vicinity to the other nodes in the network, in which case we get an upper bound on the consumption of energy. However, the assumption adapts to reality more closely in a bigger network with a lower node density. In such networks, the transmitters are tuned to transmit at maximum output power due to larger inter node distance. Under this condition, in telos, the current consumption during transmission and reception is almost the same hence, so is the energy consumption. The transmission and reception costs also depend upon the size of the component (S_C) being communicated and the protocol used for communicating it. The overhead introduced by the protocol can be measured as a constant multiplicative factor (K_F) to the size of original data to be communicated. These factors multiplied by the transfer cost of a single bit (K_{BT}) complete the expression for transfer cost of a component. Since E_P involves processing on a single node only, it can be measured empirically as shown in Figure 3. For the Deluge protocol the K_F is 3.35 [6] and a telos node configured with TinyOS consumes 0.0105 mJ per byte for transmission.

Component	Size (B)	Transfer Energy (mJ)	Savings Factor
BlinkC	836	58.77	46.6
BlinkAppC	7156	506.92	5.4
LedsC	1600	113.34	24.2
Msp430TimerC	4644	328.97	8.33
Blink w. Deluge	39024	2743.38	–

TABLE I
SAVINGS IN TRANSFER ENERGY DUE TO INCREMENTAL UPDATES

System	ROM (B)	RAM (B)
SOS Core	20464	1163
TinyOS w. Deluge	21132	597
Bombilla VM	39746	3196
TinyMan	15826	792

TABLE II
MEMORY USAGE COMPARISON FOR *TinyMan*

So,

$$\begin{aligned} E &= E_{Tx} + E_{Rx} + E_P \\ &\Rightarrow 2E_{Tx} + E_P \\ &\Rightarrow 2(K_F K_{BT} S_C) + E_P \\ &\Rightarrow 0.0703 S_C(mJ) + E_P \end{aligned}$$

We use this model along with the Blink application from the TinyOS repository to estimate the transfer costs of different constituent components of the application. The Blink application is broken down into four components; LedsC, Msp430TimerC, BlinkC and BlinkAppC. In the presented system, any of these components can be individually and remotely modified whereas in Deluge the whole application needs to be replaced. The resultant reduction in energy costs is presented in Table I.

B. Memory Usage

The memory footprint of the presented system is quite moderate in comparison with the popular existing solutions as shown in Table II. On telos rev. B it consumes only 7.7% of RAM and 32% of program memory with rest of the resources available for the loaded application. The external flash memory is completely available for the file system and 'Golden Images'. The optimized memory footprint results from the design approach of keeping the runtime support layer as thin as possible. This optimizes the usage of hardware resources available on the platform, hence leaving more memory space for the loaded application.

C. Performance Overhead

Keeping the runtime support layer thinner has a positive effect on the runtime computational requirements as well. During normal application execution – the most frequent activity for a sensor node – only the interrupt routing component of *TinyMan* is active, and introduces a short delay in the processing of interrupts. On telos platform the worst case delay is that of 23 instruction cycles – equivalent to processing required for copying eight bytes in memory. No performance depreciation is caused by other components and the code execution remains native.

V. CONCLUSION AND FUTURE WORK

In this paper, we presented a fine grained code update mechanism for sensor networks that offers the functionality and performance required for remote adaptation of sensor applications. The presented system is tightly and transparently integrated with TinyOS, resulting in ease of adoption and full utilization of the seasoned TinyOS code repository.

The preliminary evaluation, as presented earlier, provided a proof of concept. In the future we plan a more thorough evaluation with real life applications. Also, some of the steps during compilation need to be automated. Finally we would like to optimize the ELF format further and evaluate the system using other common hardware platforms as well.

REFERENCES

[1] D. Pompili, T. Melodia, and I. F. Akyildiz, "Deployment analysis in underwater acoustic wireless sensor networks," in *WUWNet '06: Proceedings of the first ACM international workshop on Under Water Networks*. Los Angeles, CA, USA: ACM, 2006, pp. 48–55.

[2] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein, "Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet," in *ASPLOS-X: In Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, ser. 37, no. 10. ACM, October 2002, pp. 96–107.

[3] G. Werner-Allen, K. Lorincz, M. Welsh, O. Marcillo, J. Johnson, M. Ruiz, and J. Lees, "Deploying a wireless sensor network on an active volcano," *IEEE Internet Computing*, vol. 10, no. 2, pp. 18–25, 2006.

[4] R. Szewczyk, E. Osterweil, J. Polastre, M. Hamilton, A. Mainwaring, and D. Estrin, "Habitat monitoring with sensor networks," *Commun. ACM*, vol. 47, no. 6, pp. 34–40, 2004.

[5] J. Jeong, S. Kim, and A. Broad, "Network reprogramming," Aug 12, 2003. [Online]. Available: http://www.tinyos.net/tinyos-1.x/doc/Xnp.pdf

[6] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. Baltimore, MD, USA: ACM, 2004, pp. 81–94.

[7] N. Reijers and K. Langendoen, "Efficient code distribution in wireless sensor networks," in *WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*. San Diego, CA, USA: ACM, 2003, pp. 60–67.

[8] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," in *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. ACM, 2002, pp. 85–95.

[9] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, Washington, DC, USA, 2004, pp. 455–462.

[10] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *MobiSys '05: Proceedigs of third international conference on Mobile Systems, Applications and Services*. Seattle, Washington: ACM, 2005, pp. 163–176.

[11] L. Mottola, G. P. Picco, and A. A. Sheikh, "Figaro: Fine-grained software reconfiguration for wireless sensor networks," in *EWSN '08: Proceedings of the fifth European Workshop on Wireless Sensor Networks*, Bologna, Italy, 2008, pp. 286–304.

[12] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel, "Flexcup: A flexible and efficient code update mechanism for sensor networks," in *EWSN '06: Proceedings of the third European Workshop on Wireless Sensor Networks*, 2006, pp. 212–227.

[13] R. Musaloiu-E., C.-J. M. Liang, and A. Terzis, "A Modular Approach for WSN Applications," CS. Dept. Johns Hopkins University, HiNRG, 2008.