Modular remote reprogramming of sensor nodes

Waqaas Munawar

Karlsruhe Institute of Technology,
P.O. Box 3640, Karlsruhe 76021, Germany
Email: munawar@kit.edu

Muhammad Hamad Alizai*

Department of Computer Science,
LUMS SBASSE, 54792 Lahore, Pakistan
Email: hamad.alizai@lums.edu.pk
*Corresponding author

Olaf Landsiedel

Chalmers University of Technology,
SE-412 96 Göteborg, Sweden
Email: olafl@chalmers.se

Klaus Wehrle

RWTH Aachen University,
Aachen 52074, Germany
Email: wehrle@comsys.rwth-aachen.de

Abstract: Wireless sensor networks are envisioned to be deployed in the absence of permanent network infrastructure and in environments with limited or no human accessibility. Hence, such deployments demand mechanisms to remotely (i.e., over the air) reconfigure and update the software on the nodes. In this paper we introduce DyTOS, a TinyOS based remote reprogramming approach that enables the dynamic exchange of software components and thus incrementally update the operating system and its applications. The core idea is to preserve the modularity of TinyOS, i.e., its componentisation, which is lost during the normal compilation process, and enable runtime composition of TinyOS components on the sensor node. The proposed solution integrates seamlessly into the system architecture of TinyOS: It does not require any changes to the programming model of TinyOS and all existing components can be reused transparently. Our evaluation shows that DyTOS incurs a low performance overhead while keeping a smaller – up to one third – memory footprint than other comparable solutions.

Keywords: in situ reprogramming; software updates; modular development; incremental updates; remote reprogramming; transparent code integration; TinyOS.

Reference to this paper should be made as follows: Munawar, W., Alizai, M.H., Landsiedel, O. and Wehrle, K. (2015) 'Modular remote reprogramming of sensor nodes', *Int. J. Sensor Networks*, Vol. 19, Nos. 3/4, pp.251–265.

Biographical notes: Waqaas Munawar is a PhD student at the Institute of Process Control and Robotics at KIT, Germany. His research interests include cyber physical systems and real time operating systems. He graduated from the RWTH Aachen University and holds double Master's degree in Computer Science and Artificial Intelligence.

Muhammad Hamad Alizai is an Assistant Professor at the Department of Computer Science, LUMS. Previously, he was a Research Assistant at ComSys Group, RWTH Aachen University, Germany. He received his PhD and MSc from RWTH Aachen University in 2012 and 2007, respectively. His research interests are in mobile applications, internet of things, sensornets, and delay tolerant networking. He is particularly interested in applications, protocols, architectures, and evaluation tools for future networks.

Olaf Landsiedel is an Assistant Professor at Chalmers University of Technology in Gothenburg, Sweden. He received his PhD from the RWTH Aachen University, Germany, and was a Post-Doctoral at the KTH Royal Institute of Technology, Sweden, and the Swedish Institute of Computer Science (SICS). His work focuses on cyber physical systems, wireless sensor networks, and the internet of things.

Klaus Wehrle is a Professor of Computer Science and Head of the Chair of Communication and Distributed Systems at the RWTH Aachen University, Germany. He received his Diploma (1999) and PhD (2002) degrees from the University of Karlsruhe (now KIT), both with honours. He joined the International Computer Science Institute at University of California at Berkeley from 2002 to 2003. In 2006, he joined the RWTH Aachen University as an Associate Professor, later as Full Professor. His research activities are focused on engineering of networking protocols, (formal) methods for protocol engineering, sensor networks, peer-to-peer-networking as well as all operating system issues of networking.

This paper is a revised and expanded version of a paper entitled 'Dynamic tinyos: modular and transparent incremental code-updates for sensor networks' presented at *IEEE International Conference on Communications (ICC)*, Cape Town, South Africa, 23–27 May, 2010.

1 Introduction

The ability to remotely update (or reprogram) the software on sensor nodes is pivotal in providing incremental, application development support for wireless sensor networks. The need for incremental software update could arise for example to fix a bug, to update a set of features or parameters, or even to reprogram sensor nodes with a complete new application due to changes in the environmental requirements (Deng and Nickerson, 2013; Yick et al., 2008). However, a common understanding in the research community exists that in situ reprogramming of sensor nodes is challenged by:

- The lack of permanent network infrastructure (e.g., cables) and physical inaccessibility in most sensor network deployments (Pompili et al., 2006; Juang et al., 2002; Szewczyk et al., 2004; Werner-Allen et al., 2006), leaving the wireless medium as the only choice to access the nodes in the network.
- The embedded nature and the scale of deployment that could surpass hundreds or even thousands of sensor nodes resulting in increasingly long network downtime due to reprogramming operation.
- The limited nature of sensor nodes because of severe resource constraints in terms of communication bandwidth, memory capacity, processing power and energy.
- The complexity of reprogramming tool that often results in steep learning curves and thus the lack of interest on the part of developers in exercising such an incremental development facility.
- The need for transparent integration of the reprogramming tool in the existing system architecture to enable code reuse.

In the past few years, over-the-air (OTA) reprogramming has established itself as the only relevant solution to deal with the lack of permanent infrastructure and physical inaccessibility. Ranging from full binary image replacement in a node's memory to differential reprogramming to virtual machines, a plethora of OTA based reprogramming solutions (Jeong et al., 2003; Hui and Culler, 2004; Panta et al., 2009; Jeong, 2004; Fei and Magill, 2012; Müller et al., 2007; Bohli et al., 2009, 2011) has been proposed in the literature. However, we believe that these solutions have failed to address the aforementioned challenges in entirety. For example, full-image replacement (Jeong et al., 2003; Hui and Culler, 2004) results in a very high transmission overhead while showing low processing demands. In contrast, other approaches such as transmitting the deltas between new and old binary image (Panta et al., 2011, 2009; Jeong, 2004) can sometime reduce the transmission cost but in worst case scenario, just emulate the full image replacement. On the other hand, virtual machines (Levis and Culler, 2002; Müller et al., 2007) reduce transmission costs as well as post processing requirements but incur a major energy-cost overhead due to interpretation instead of execution. Moreover, the majority of these solutions does not transparently integrate into the existing development environments resulting in new and complex programming models rendering the application repositories developed over the years, useless.

Hence, an efficient solution for remote reprogramming is desired that collectively addresses the aforementioned challenges. This paper establishes such a solution in the form of DyTOS – a dynamic operating system extension for TinyOS to support runtime adaptation of the OS and its applications. Our choice of TinyOS platform is motivated by the fact that it is one of the most widely used (Levis, 2012) operating system and owns a very rich repository of applications and protocols. However, its remote reprogramming mechanism is still limited to full image replacement as nodes execute a statically-linked system-image generated at compilation time (Munawar et al., 2010).

Our system design preserves the component model of TinyOS: TinyOS applications and the OS itself are built by connecting so called components. Components represent functional building blocks such as communication protocols, device drivers, or data analysis modules. During the default compilation process of TinyOS, these building blocks are converted into a single, static binary. While this enables code optimisation and ensures a small memory footprint, it omits the modularity of the OS and its applications. In contrast, this work introduces dynamic extensions to TinyOS allowing the user to define TinyOS components that should be kept modular in the resulting executable. As a result, Dynamic TinyOS allows to replace these components dynamically at runtime.

We now pinpoint the following requirements for an efficient remote reprogramming solution, thereby marking the contributions of this paper by highlighting how DyTOS achieves these requirements.

- *Flexibility* (*reprogramming granularity*): The granularity of a reprogramming facility should be a user definable parameter. This allows the users to trade-off communication overhead for processing (e.g., linking of components on sensor nodes) and vice versa, depending upon the future requirements of a particular deployment. DyTOS allows the user to control the reprogramming granularity by defining TinyOS components that should be kept modular, and hence, independently reprogrammable in the resulting executable.
- *Transparency* (*seamless integration*): An ideal in situ reprogramming solution should not necessitate any change in the existing, developed code base. It should transparently integrate with existing development platforms enabling code-reuse of large code repositories which are result of a substantial, decade long research and development effort. The operation of DyTOS does not require any changes in the existing sources. Thereby, it enables the reuse of existing application repositories by integrating seamlessly into the existing TinyOS architecture.
- *Usability* (*no new programming model*): It should be easy to use, and hence, should not introduce new and complex programming models. DyTOS is based on unmodified nesC-programming constructs and its semantics are deeply embedded in the TinyOS compilation process remaining transparent to an application developer.
- *Efficiency* (*limited resource consumption*): It should minimise energy consumption, incur minimal processing and communication overhead, and embrace reduced memory footprint. DyTOS's optimisations to the binary executables (i.e., ELF files) results in significant reduction of the communication and processing overhead, while keeping a smaller – up to one third – memory footprint than other comparable solutions.

The rest of this paper is structured as follows: The detailed architecture of DyTOS is presented in Section 2. We discuss our evaluation results in Section 3. Finally, we discuss the related work in Section 4 before concluding the discussion in Section 5.

2 DyTOS architecture

TinyOS based sensor network applications consist of a large selection of individual software components which are 'wired' together to achieve the desired functionality. In the standard TinyOS compilation process, these building blocks are mashed-up to form a single, monolithic binary-image of the application. However, as each component provides a dedicated functionality to the overall system, updates such as the deployment of a new functionality or a bug fix are commonly limited to a small number of neighbouring components or even a single component. Hence, the modularity of TinyOS forms a natural starting point for a dynamic operating system: DyTOS alters the compilation process of TinyOS to preserve user selected parts of the component structure across the compilation phase. The result is an executable consisting of multiple, replaceable objects. Hence, during deployment, updates of applications or of the OS itself can be disseminated in the network to replace existing objects on the sensor node.

Code updates in DyTOS work in three phases, see Figure 1:

- Via extensions to the NesC compiler of TinyOS we compile components, i.e., applications and system components, into multiple objects. As a result, the component based structure of the TinyOS application is preserved during the compilation process.
- Using a standard dissemination algorithm, such as the one of Deluge (Hui and Culler, 2004) or others (Stathopoulos et al., 2003; Levis et al., 2004), updates – i.e., binary objects – are transferred to the sensor node over the radio.
- A thin runtime on the node stores these updates and integrates new components into applications or the OS.

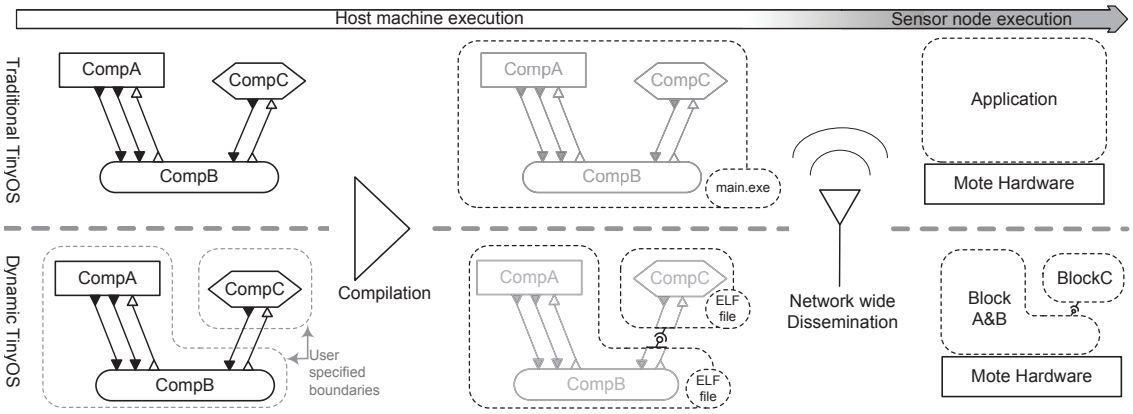
Moreover, DyTOS allows users to define the granularity of replacements in a simple configuration file for NesC compiler. Hence, a user can combine multiple TinyOS components into a single object. While this increases the size of updates, it reduces the overhead of run-time linking on the sensor node and enables compiler optimisations inside this object. As an example, Figure 1 shows the process of dividing an application. It can be one of the applications shown in Table 2 with its respective components. Here the application is divided into two blocks, one containing components A and B and the other containing component C. Any updates to component C will only require the retransmission of this single component for updates, while updates to A require the block containing A and B to be disseminated in the network. Thus, the trade-off between transmission energy and linking overhead can be adapted based on expected future application and deployment requirements.

The overall architecture of DyTOS has two main components:

- On the host, we isolate a single TinyOS component or a group of components and compile them into an ELF object.
- We provide *Tiny Manager*, a runtime system executing on the sensor node. It handles storage and integration of new components, i.e., code updates.

These new ELF objects are linked into an executable binary-image and loaded in program memory. After discussing these two core components of DyTOS, we conclude this Section by presenting optimisations for ELF objects to minimise transmission and linking overhead.

Figure 1 Overview of code updates in DyTOS in comparison with the traditional TinyOS: In the first step a user chooses boundaries among the building blocks of an application and the OS. In the second step, ELF files are generated according to the specified boundaries. These ELF file are disseminated in the network and linked on the mote to form the application



2.1 Compile time system

First, DyTOS compiles an application and the OS core into separate ELF files based on user specified boundaries for incremental updates. Compiling parts of a TinyOS application in isolation from the rest has two side effects:

- it introduces ambiguities, such as the parameterisation of NesC generics and default event handlers
- it limits compiler optimisations. To address the first issue, DyTOS provides a compiler extension, so called component isolation, which resolves these ambiguities and enables automated compilation of TinyOS components into solitary ELF objects.

Moreover, we introduce *component over-provisioning* that establishes additional functionality into the existing components on the sensor node to achieve maximum benefits from our approach, as discussed in the following sections. DyTOS addresses the second issue by allowing the user to group multiple TinyOS components into a single object limiting modularisation to user required parts. While these larger objects increase optimisation possibilities, such as code in-lining and loop un-rollments, they increase the size of updates. Hence, DyTOS allows users to balance the cost of updates and their performance penalties.

2.1.1 Component isolation

The main issues faced during isolation of TinyOS components is the non-availability of system information which is hidden in parts of the application that are not being compiled at the moment. For example, the timer dispatch component needs to be parameterised with the number of timers used in the OS and applications. Similarly, the scheduler needs to be parameterised with the number of threads in the system. *Component Isolation* of DyTOS parameterises such components by collecting information from other modules in the system and user requirements with which it configures the NesC compiler of TinyOS. This parameterisation 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 with the required knowledge of the rest of the application for 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 transparent integration allows the reuse of existing TinyOS based applications and seamless integration of DyTOS into the existing TinyOS skeleton, thereby remaining transparent to the application developer.

2.1.2 Component over-provisioning

Component over-provisioning allows to provide additional functionality for expected future updates. For example, the OS core can provide additional timers or slots for additional threads expected to be required by future deployments of new functionality. Additionally, over-provisioning can be used to configure a *base block* to provide the functionality needed by typical sensor network applications, i.e., consisting of timers, scheduler, radio and other hardware drivers. This design ensures that a currently deployed application can be changed to radically different tasks by merely communicating the user implemented part of the new application (see our evaluation in Section 3 for an example). Nonetheless, unlike TOSThreads, the proposed system is completely dynamic and supports the exchange of both user and OS related components including low-level device drivers etc. Similarly, the presented system imitates dynamic reconfiguration behaviour of Contiki. Hence, the former can be seen as a generalised case of the latter.

2.2 Node runtime: tiny manager

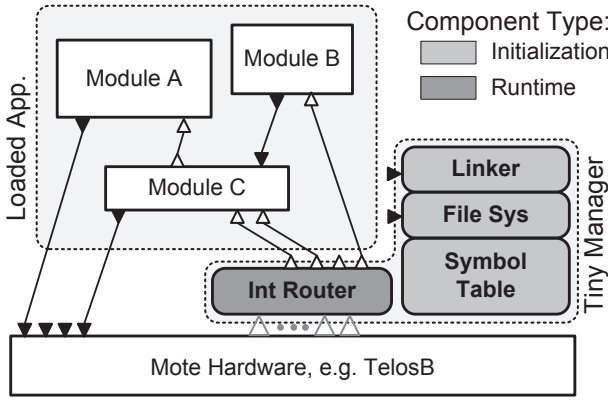
After the compilation of components, the next step consists of their dissemination and integration in the sensor application executing on the sensor node.

We currently use Deluge data dissemination protocol. However, other well established and widely used protocols (Hui and Culler, 2004; Stathopoulos et al., 2003; Levis et al.,

2004) would work as well. Since the dissemination protocol is also treated as a part of the loaded application on the sensor node, it 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. It is important to point out that the efficient distribution of code is completely orthogonal to the challenges address in this paper, which aims at reducing the size of such updates and adding the flexibility needed for adaptation. Therefore, a comprehensive exploration of all the existing dissemination techniques is beyond the scope of this paper.

Once the updates have been received on the sensor node, the dissemination protocol invokes the linker to integrate the received modules and place the resulting new binary image in the program memory of the sensor node. To accomplish this, DyTOS provides a runtime on the sensor node, called Tiny Manager (see Figure 2). It consists of the following four main components.

Figure 2 Architectural elements of *Tiny Manager*. Only the interrupt router is active during the normal execution of a loaded application. Linker, file system and symbol table handle storage and integration of newly received code



2.2.1 File system

It is used to store and retrieve the new building blocks (ELF modules) which were received by the dissemination protocol. It is implemented by the Tiny Manager and the API is available to the loaded application. Our file systems is based on the Coffee file system (Tsiftes et al., 2009) for managing large chunks of data. Coffee is shipped with Contiki operating system and utilises the external flash ROM for storage. It provides a POSIX style API for read/write access to the files.

Our file system uses a combination of micro log files and extents. When a new file is opened, it starts with an extent structure that consists of consecutive allocation pages. When the file is modified, a log file is created and linked with the initial extent structure. The log file stores modifications to the original file as log records. When the log file fills up, a fresh extent file is created with the most recent data merged from old log and extent files. The old files are then deleted.

Resource usage of our file system is quite modest: It consumes approximately 6 KB of flash memory. The maximum RAM footprint depends upon the configuration of the file system. In its default setup, our system uses 173 bytes of static memory and a maximum stack size of approximately 600 bytes. Memory usage can be further optimised by adapting the configuration.

2.2.2 Linker

It is responsible for linking the new ELF modules and placing the instructions in the designated code memory. It is divided into two parts; the platform specific part to cater for platform level details of linking, and the platform independent part for high level processing shared by all platforms.

After receiving all the components of the application, the linker is invoked. The role of the linker is to link the components and then load them into the program memory of the sensor node. Considering the role of the linker, it needs to fulfil two distinct sets of requirements; firstly, intra-object requirements or the *micro requirements*, and secondly, inter-object requirements or *macro requirements*. The micro requirements deal with the linkers operation on per object basis. This includes the operations it needs to perform in order to successfully load and execute a single object. The macro requirements, on the other hand, deals with the operations it needs to perform in order to make all objects work with each other successfully. The mechanism chosen for linking has notable effect on system properties like performance overhead, latency and memory requirements.

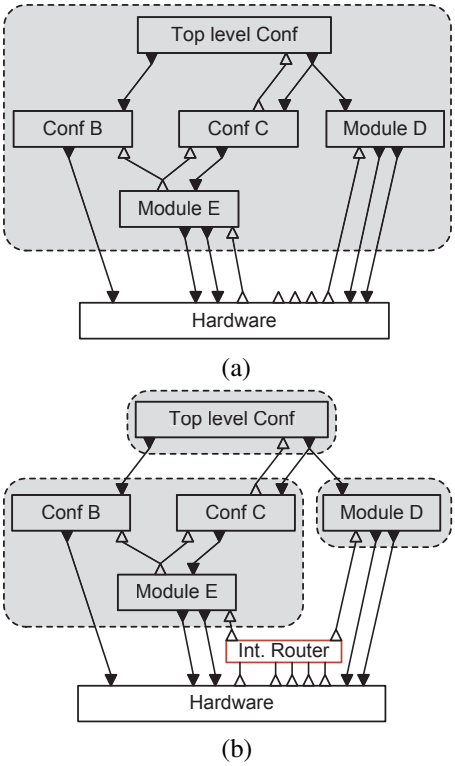
Micro requirements: To integrate an object into the existing executable core, the linker performs four steps on each ELF object. Sometimes, these steps are performed multiple times to resolve the addresses of undefined symbols. The decision to repeat the procedure is made by the linker on basis of an algorithm to resolve the missing dependencies. These steps include;

- *Memory allocation*: Depending upon the size of the different sections mentioned in the ELF object being processed, the memory is allocated for them both in RAM and internal flash.
- *Relocation*: On the basis of the addresses obtained in the previous step and the contents of the relocation segments, the relocations are calculated and applied to the code and the data segments in the ELF object.
- *Linking*: The relocation and linking phases are interleaved. During relocation, when the linker finds an undefined symbol, it resolves its address in the global symbol table and links it to a symbol having the same name in another ELF object or in the existing executable core.
- *Placement*: Lastly, the appropriate segments of an ELF file are placed in the program memory. All the available symbols from the file are added to the global symbol table.

The linker also needs to handle interrupt routines. This involves use of ‘interrupt router’ and it is done in the last phase before placement (cf. Section 2.2.4).

Macro requirements: An application in TinyOS consists of small components, which communicate with each other via interfaces. Figure 3 shows internal structure of a TinyOS application. The flow of control is bidirectional, i.e., it moves 'down' via commands and 'up' via events. To recreate this using binary components, the linker needs to fulfil two requirements; Firstly, it should be able to recreate the links as in the original application. Secondly, the system should provide a mechanism to route the interrupts to the appropriate component in the application. In the proposed system these requirements are fulfilled with the help of a global symbol table and an interrupt router.

Figure 3 Internal structure of a TinyOS application: (a) TinyOS and (b) DyTOS (see online version for colours)



2.2.3 Global symbol table

The ELF modules of an application are mutually dependent on each other for resolution of all the required symbols. These dependencies can be data objects or function implementations. These are transitive and can even be cyclic. However, all of these dependencies can be fully resolved among the components belonging to a single application. A naive solution to resolve them could be to start a search among all the components whenever a missing dependency is to be resolved. Although this method is efficient in terms of the required memory, but the time taken to resolve all the dependencies is significantly long: The space complexity is constant but the time complexity is exponential. The other method is to populate a global symbol table with the available symbols in all the components and then load these components. It imposes a time and space complexity of $O(n)$. In this method, all the files are parsed twice i.e., $2n$ operations for n files.

We employ this latter approach for resolving object dependencies in DyTOS. We parse all the files twice, except the last file that is parsed only once since the global symbol table already contains all the symbols that the last file needs at the time when it is loaded. This procedure uses a total of $2n - 1$ parse-operations for n files. The symbols provided by the executable core are added to the global symbol table during the process of compilation. These symbols include the file system's API and the linker's API. This is to ensure that the application can parse and store new ELF files that it might receive for next application update, for example. After resolving dependencies, the components are loaded into program memory. At this point the application is completely linked and ready to run, except interrupts (bottom layer of upward pointing triangles in Figure 3) which need to be routed to the component providing the appropriate interrupt service routing (ISR).

2.2.4 Interrupt router

TinyOS applications are reactive and allow a bidirectional flow of control. Most of the events are generated by the hardware and are driven 'up' towards the software components that provide the corresponding service routines. When hardware generates an event (interrupt), it saves the context of current execution and jumps to a predefined, hardwired address in the program memory and starts executing the instructions from that address onwards. This address is different for all the interrupts that can be generated. The software routines that are meant to handle the interrupts are placed at these addresses. As the placement of the different routines in the memory is done by the offline linker (i.e., during compilation at a host machine), the appropriate routines in the code are marked by a specified keyword to communicate this fact to the linker. However, since in DyTOS, the modules are loaded dynamically, placing different routines at different addresses at run time could generate memory conflicts among interrupt service routines. We use an interrupt router to avoid such conflicts.

The interrupt router employs an additional indirection layer of abstraction by using a dummy ISR for each hardware interrupt. Later when linking a component, if one of its routines is an ISR, the interrupt router stores its address in the memory. When an interrupt occurs, the interrupt router is invoked which starts the execution of the instructions from the stored address. This extra indirection causes a nominal delay in the service of an interrupt but simplifies memory management.

Apart from the interrupt router, all components are inactive during the normal execution of application. This minimises the runtime overhead of the *Tiny Manager*.

2.3 ELF optimisations

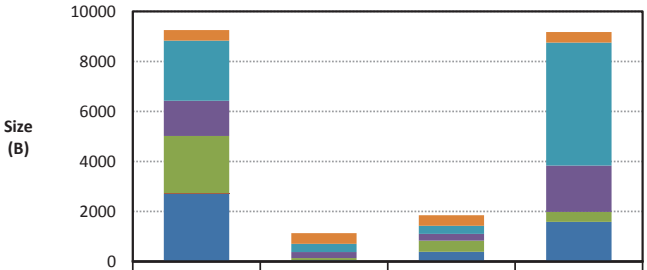
After discussing the design of DyTOS in detail, we discuss optimisations to ELF files to reduce their size. These optimisations reduce transmission overhead and storage requirements which are both scarce resources on sensor nodes. The two main hindrances curtailing the performance of DyTOS are

- large size of ELF modules
- processor intensive linking process.

2.3.1 Reducing the size of ELF modules

The ELF format, though a widely used standard, is not optimised for the low-power processors commonly found on sensor network platforms. For example, in the ELF modules, the addressing information is stored in 32 bit format whereas majority of sensor platforms offer 8 or 16 bit addressing space. Though this does not cause any reduction in performance but results in inflation in the size of the modules. Similarly, the major contribution in the size of ELF modules 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 – up to 80 characters for each symbol. The contributions of different segments of different ELF files belonging to an exemplary sensor network application are shown in Figure 4.

Figure 4 Breakdown of contribution from different sections of ELF file toward its total size. The string table included in these files is the main contributor toward their size and hence is primary candidate for optimisation. These ELF objects belong to the trademark *Blink* application[a] from TinyOS repository (see online version for colours)



	BlinkAppC	BlinkC	LedsC	Msp430 TimerC
Misc.	422	425	423	424
String Table	2404	329	323	4914
Symbol Table	1408	240	272	1856
Relocations	2292	84	444	396
Data	2	0	0	0
Code	2724	54	390	1586

[a]Blink is the most basic sensor network application in TinyOS

Our optimisations deal with reducing the size of the symbol tables because

- it offers a bigger reduction in the size of the ELF module thereby reducing the energy cost of transmission
- it reduces processing required for all the string comparison operations performed during the linking which saves on the energy spent in that phase.

We decrease the size of the symbol names down to three 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:

- significant reduction in the size of ELF file
- reduction in the size of global symbol table which is constructed during linking
- reduction in number of string comparison operations.

For example, this technique reduces the size of the string tables of all components of Blink application from 7970 bytes to 888 bytes and results in the 33% reduction in cumulative size of all ELF files belonging to this application.

2.3.2 Optimising the linking process

The second set of optimisations is directed towards reducing the amount of processing required during linking through reducing the size of global symbol table. 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 the ROM in a sorted order allowing binary search among the symbols. This results in a fast linking process. Secondly, instead of accumulating all the offered symbols in the runtime portion of global symbol table, we accumulate only the required symbols. As the generated ELF modules typically offer more symbols than they require, this causes significant reduction in the size of runtime portion of the symbol table. Thereby, further reducing string comparison operations and expediting the linking process.

These two optimisations enhance the processing speed, resulting in energy savings of up to 66% when compared to the original ELF file without needing to change the overall structure of the file itself. Both these optimisations are performed with simple scripts without using any customised tools for this purpose. This allows the use of standard tool chains while avoiding any maintenance and porting effort.

3 Experimental evaluation

System requirements, such as flexibility, transparency and usability, are inherently incorporated in the design and architecture of DyTOS discussed in the previous section. However, the efficiency (i.e., resource consumption) of DyTOS can only be demonstrated with the help of experimental evaluation. Thus, we now compare the efficiency of DyTOS with existing approaches for code updates in TinyOS, such as Deluge and, where possible, Zephyr and Maté. Zephyr is not (yet) open-source. Hence, – when available – our comparison relies on the same benchmarks as used by Zephyr to establish a base for a fair comparison. The evaluation focuses on key factors such as the energy consumption, size of updates, memory footprint, and processing overhead. We implemented DyTOS for the TelosB platform (Polastre et al., 2005) and the recent 2.1 release of TinyOS (Levis, 2012).

Our comparative evaluation covers a broad range of standard applications from the TinyOS repository, underlining the feasibility of our approach. These applications, for example, include simple applications, such as Blink and

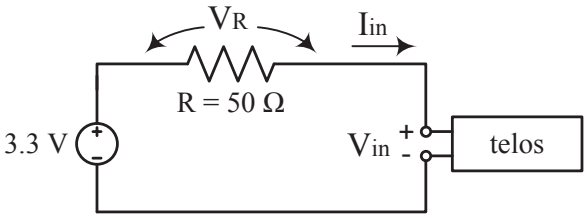
BlinkTask – which toggle the LEDs of a sensor platform – and complex applications, such as MultihopOscilloscope – which uses the standard collect tree protocol (Gnawali et al., 2009) in TinyOS. However, for our detailed analysis of energy and processing overhead, we use a simple BlinkTask application. The reason is that the BlinkTask application contains most of the programming constructs of nesC (i.e., tasks, commands, events, generic components, scheduler etc.) and can be used as a representative to calculate the detailed communication, processing and linking overhead. We show that this simple application can provide us with useful hints about the significance of our approach when compared to other existing approaches.

Deriving energy consumption of a remote reprogramming solution involves complex operations such as multi-hop communication and code integration. We therefore need a simple energy model to efficiently compare DyTOS with other state-of-the-art solutions. Hence, before presenting detailed evaluation results, we thoroughly present our experimental setup and the energy model.

3.1 Experimental setup

The circuit employed for the empirical measurements of different values of interest such as input current, energy consumption and delays is shown in Figure 5.

Figure 5 Current measurement setup



The input current (I_{in}) is significantly important since it can be used for measuring energy consumption and the timing information of different events. To facilitate the measurement of I_{in}, we add a resistor in series with the TelosB and the fixed voltage supply of 3.3 V. The operating voltage for TelosB board ranges from 1.8 V to a maximum of 3.6 V. However, when the input voltage drops below 2.4 V, the internal and external flash memories cannot be reprogrammed. Hence, with a 50 Ω resistor in series, the maximum allowed current for reliable operation is 18 mA. If the TelosB sinks more than 18 mA, it's input voltage decreases below 2.4 V.

During our evaluation, we observed that TelosB requires 12 mA to execute a TinyOS application that does not use radio hardware. However, during the operation of the CC2420 radio, the input current reaches a value of 22 mA causing a drop of 1.1 V on the series resistor. This results in the decrease of V_{in} to 2.2 V. However, as applications typically do not re-configure flash memories during the operation of the radio chip, the circuit shown in Figure 5 fulfils our requirements. With this setup, the instantaneous value of power consumption (P_t) of TelosB board can be derived as:

$$P_t = V_{\text{in}} I_{\text{in}} \Rightarrow \frac{V_{\text{in}} V_R}{R} \tag{1}$$

Energy consumption (E) during any phase of the operation of any application can be calculated by integrating the instantaneous power curve over that period (T).

$$E = \int_T P_t \mathrm{d}t \Rightarrow \frac{1}{R} \int_T (V_{\text{in}} V_R) \mathrm{d}t \tag{2}$$

V_R and V_{in} are logged using a storage oscilloscope and stored for further analysis. We collected 2500 sample points for each reading. Timing information is also retrieved from the I_{in} logs. Distinct peaks are generated in the current being sinked by blinking the three on board LEDs simultaneously. This blinking process is used as markers for measuring the elapsed time between different events.

3.2 The energy model

To evaluate the per-node energy consumption caused due to a network wide reconfiguration, we devise an energy model and calibrate it using the readings taken empirically. The two main factors that contribute to the overall energy overhead of any remote reprogramming solution are:

- dissemination of component updates
- linking of newly received components with the application on the sensor node.

We model the energy cost as:

$$E_T = E_C + E_P \tag{3}$$

where

E_T is the total energy consumed

E_C is the energy cost of multihop communication

E_P is the energy consumed during processing.

The total energy consumed during the complete process is the sum of energy consumed during the transfer of the updated component and its integration into the existing core. Assuming each node receives the update and then propagates it, the energy consumed during multi-hop communication can further be divided into transmission cost and the reception cost. For the sake of simplicity, we currently ignore the fact that the bordering nodes that do not need to propagate the update further.

Since TelosB's radio hardware, when tuned to the maximum output power, consumes nearly the same amount of energy (cf. Table 1) for both transmission and reception operations, we treat the energy cost of transmission and reception as equal;

$$E_C = E_{Tx} + E_{Rx} = 2E_{Tx} \tag{4}$$

The energy consumed to transmit a component is proportional to the size of the component, and a constant factor due to the overhead introduced by the protocol employed for reliably disseminating the updated component across multiple hops.

$$E_{Tx} = K_F S_C K_{BT} \tag{5}$$

$$\Rightarrow E_C = 2K_F S_C K_{BT} \tag{6}$$

where

K_F is the average overhead of the protocol used

S_C is the size of the component to be transferred in bytes

K_{BT} is the energy consumption required for the transmission of each byte.

Table 1 Energy consumption for communicating 3000 bytes on TelosB. Measurement methodology is outlined in Section 3.3

	Energy (mJ)	*Per byte* (mJ)
Transmission	31.56	0.0105
Reception	31.80	0.0106

The second contribution in the total consumption of energy comes from the processing overhead associated with the integration of the received component into the main executable core at the sensor node. The different phases of this operation include storage of received component, linking and relocation of all the received components and loading the components into program memory. Hence, the corresponding energy cost of these operations, i.e., the cost of storage, the cost of linking/relocating and the cost of loading the component into program memory is formulated as:

$$E_P = E_S + E_{LR} + E_{LP} \tag{7}$$

where

E_S is the energy required to store the component

E_{LR} is the energy consumed in linking and relocation operation

E_{LP} is the energy consumed in loading the component.

These entities also depend on the size of the component being processed. Since, all these operations take place on the same individual sensor node, the energy consumed during these operations can be empirically measured. The results of our E_P measurements are presented in Section 3.5.

3.3 Energy consumption during code dissemination

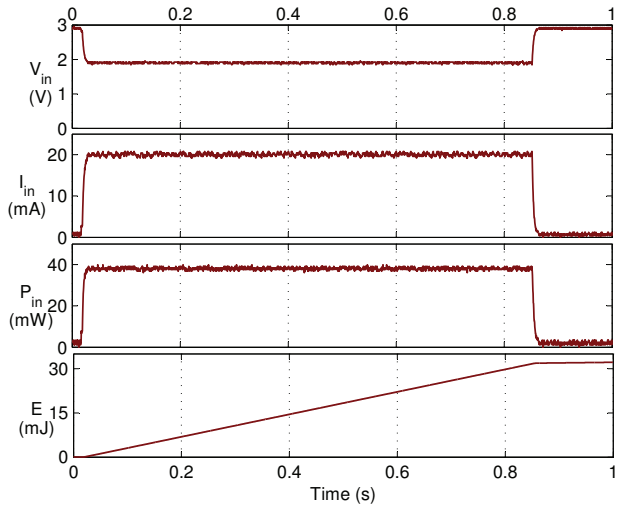
Our comparison in this Section compares DyTOS with the original Deluge full image replacement mechanism (Hui and Culler, 2004) – the widely used in-field code replacement tool.

The energy consumed during the dissemination process include

- the energy required for the transmission of an updated component
- the energy consumed during the reception of that component.

Telos B employs the CC2420 radio that does not provide a bit level interface; the received data and transmitted data is handled through 128 byte I/O buffers. Hence, it is not possible to measure the duration of transmission of a single byte. To resolve this, we programmed a Telos B board to transmit 100 data packets, each with a payload of 17 bytes. The messaging component of TinyOS adds an 11 byte header to each packet resulting in transmission of 30 bytes every time. These packets are sent back-to-back without any delay. After transmission of 100 packets the transmitter is switched off. The input current of the transmitter is plotted in Figure 6. The measurement was repeated five times but the results remained consistent.

Figure 6 Input current, instantaneous power and energy consumption during the transmission of 3000 bytes with the telos platform (see online version for colours)



For the Deluge protocol the K_F is 3.35 (Hui and Culler, 2004) and TelosB node configured with TinyOS consumes 0.0105 mJ per byte for transmission. The second contributor to the communication toll is E_{Rx}. However, our empirical results (cf. Table 1) suggest that it is equal to the E_{Tx} for TelosB platform when its transmitter is set to the maximum output power. Summarising,

$$E_C = E_{Tx} + E_{Rx} \tag{8}$$

$$\Rightarrow 2E_{Tx}$$
$$\Rightarrow 2(K_F K_{BT} S_C)$$
$$\Rightarrow 0.0703 S_C(\text{mJ})$$

We evaluate the energy consumption during the dissemination process of DyTOS for a set of representative applications from the TinyOS repository (see Table 2). These applications utilise a broad range of TinyOS system components and protocols – MAC, timers, LEDs, radio, and sensing hardware – required to drive the sensor hardware-platform, allowing us to comprehensively validate our results. To obtain these results, we modify a certain part of an application and then update the resulting binary on the sensor node using both Deluge and DyTOS. For this comparison, DyTOS also uses Deluge's dissemination mechanism for delivering modular code updates across the network. It is fair to conclude that DyTOS consumes significantly less transmission energy – up to a factor of 40 – than Deluge. However, reducing the size of our updates introduces processing overhead at the sensor node as discussed in Section 3.5.

Table 2 Savings in transfer energy due to incremental updates in DyTOS. The results are obtained after modifying parts of application and updating the resulting binary on a sensor node

Application	*Component*	*Size (B)*	*Tx. Energy* (mJ)	*Deluge size*	*Saving factor*
Blink	OS-Comp	6616	465.1	33,726	5.1
	Blink	824	57.93		40.9
	Leds	1728	121.48		19.5
	Timer	5424	381.31		6.2
	Scheduler	1980	139.19		17
BlinkTask	OS-Comp	6640	466.79	33,726	5
	Blink	992	69.74		34
	Leds	1728	121.48		19.5
	Timer	5424	381.31		6.2
	Scheduler	2356	165.63		14.3
Radio-	OS-Comp	28,232	1984.71	33,954	1.2
CntToLeds	Radio	1352	95.05		25.1
	Leds	1728	121.48		19.6
	Timer	4772	335.47		7.1
	Scheduler	3092	217.37		10.9
Sense	OS-Comp	17,040	1197.91	34,074	2
	Sense	940	66.08		36.25
	Leds	1728	121.48		19.7
	Timer	6296	442.61		5.4
	Scheduler	2576	181.09		13.2
Oscillo-	OS-Comp	39,328	2764.75	34,504	0.87
scope	Oscilloscope	2008	141.16		17.1
	Leds	1720	120.91		20.0
	Scheduler	3728	262.07		9.25

3.4 Memory requirements

We now evaluate the size of updates in DyTOS to determine its storage requirements. Figure 7 shows the results for our approach in comparison to Deluge and Zephyr for different software update scenarios ranging from a simple timer-frequency change in the Blink application to the re-tasking of sensor nodes with a completely new application. The results show that Zephyr, as it is based on byte level comparison, performs better for very small changes in an application, such as the addition of a small function to a component. However, DyTOS shows consistent update-sizes and outperforms existing approaches for component level changes, such as addition of a new component to the existing application. This is because it preserves the high level structural knowledge of an application and its components. For example, updating from the CntToLeds to the RadioCntToLeds application only requires communicating the main application component, while radio, timer and led components can be reused.

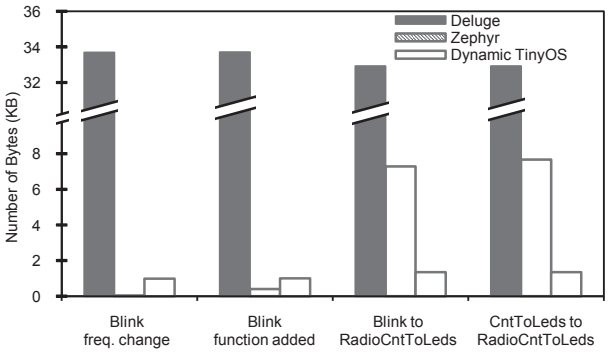
Similarly, due to a thin runtime component, i.e., Tiny Manager, the memory footprint of DyTOS is small in comparison to popular existing solutions (see Figure 8). On the TelosB platform, it consumes only 7.7% of RAM and 32% of the program memory. Furthermore, the external flash memory is completely available for the file system and 'Golden Images'. Hence, it leaves the majority of the storage resources for the OS, applications and code updates.

3.5 Processing overhead

Processing an update consists of three steps:

- storage in the external flash, i.e., file system
- linking and relocating
- loading into program memory (see Figure 9).

Figure 7 Performance in common software update scenarios. Zephyr results in smaller updates for very small changes in application. In contrast, DyTOS has a stable update-size because it operates at the component level. It outperforms Zephyr in the case of bigger changes in applications such as the addition of a new component



The energy consumption of this processing and loading of an updated component does not solely depend on the size of the component but also on the symbol dependencies and the number of relocations that need to be performed.

Deluge has a constant energy overhead because it always disseminates the complete application and OS image. In contrast, the energy overhead of DyTOS depends on the size of the components to be updated and the required processing

on the sensor node. Overall, Figure 10 shows that DyTOS outperforms Deluge in terms of the overall energy required for code dissemination and processing of updates.

Figure 8 Memory-footprint comparison for *DyTOS*. Tiny Manager only utilises 7.7% of the RAM and 32% of the internal flash ROM on TelosB platform, which is significantly less than all other comparable solutions

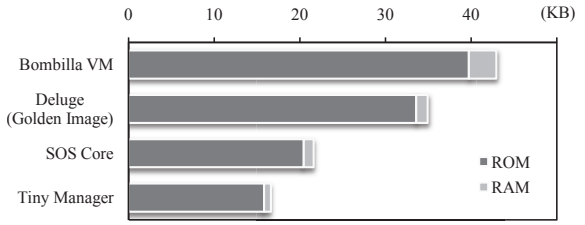


Figure 9 Current draw and energy utilisation during processing and loading of the BlinkTask application on the TelosB platform. The peaks, generated by turning on all onboard LEDs simultaneously, mark the boundaries between the different operations

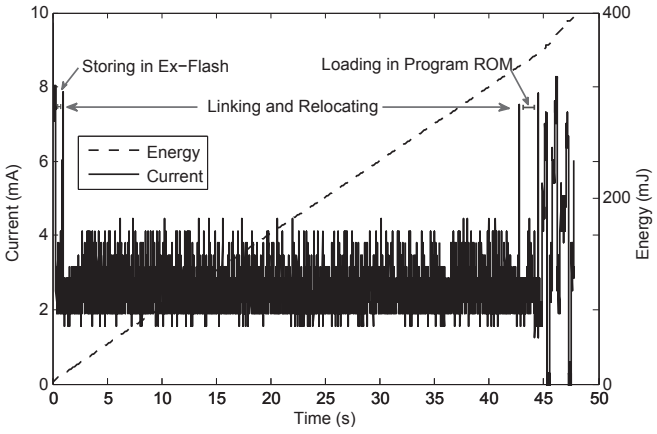
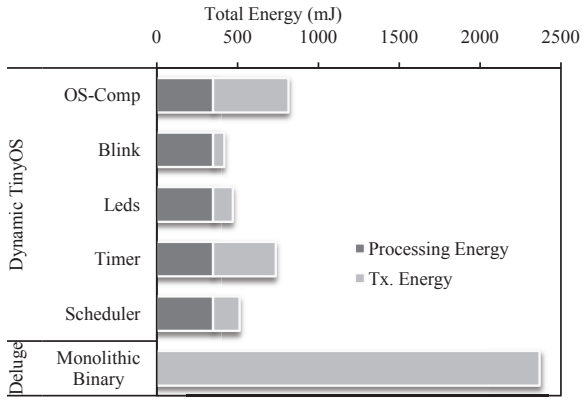


Figure 10 Energy overhead comparison of *DyTOS* with Deluge. DyTOS results in significantly less overhead than Deluge for all the update scenarios. Deluge has a constant overhead for all update scenarios



3.6 Runtime performance overhead

The changes required to enable incremental code updates in TinyOS impacts

- the compilation process
- requires the inclusion of an additional runtime layer, i.e., TinyManager.

3.6.1 Compiler optimisations

Compiling parts of an application in isolation reduces the overall code optimisation possibilities for a compiler. Moreover, setting explicit boundaries between application components can result in additional calls to those functions which otherwise might have been *inline*d. To stress-test the impact of isolated compilation of components, we use two benchmarks:

- A syntactic benchmark, which uses the TestScheduler application from the TinyOS repository, as a sanity check for our approach. To evaluate our approach from the worst-case point of view, we modified this application to make 10,000 cross component calls to the Scheduler.
- An application that calculates a 256 point FFT.

Table 3 shows that DyTOS consumes slightly more code memory when compared with the traditional TinyOS. Furthermore, the runtime performance overhead of DyTOS is very small: While showing a less than 10% worst case overhead, it takes the same execution time for a computationally intensive real-world algorithm, i.e., FFT.

Table 3 Impact of compiler optimisations on the memory size and execution time

	DyTOS		*Traditional TinyOS*	
	ROM (B)	*Time*	*ROM* (B)	*Time*
TestScheduler	1810	504 ms	1754	460 ms
FFT	14,718	5.320 s	14,232	5.320 s

3.6.2 Interrupt re-routing overhead

During normal application execution, i.e., when no updates are processed, only the interrupt routing component of *Tiny Manager* is active. It introduces a short delay in the processing of interrupts. On the TelosB platform, the worst case delay is 23 instruction cycles – equivalent to processing required for copying eight bytes in memory. No performance depreciation is caused by other components and hence code execution in DyTOS remains native.

Overall, our evaluation shows that DyTOS outperforms other approaches for code updates in TinyOS and even shows a very small overhead when compared to a static TinyOS binary.

4 Related work and comparative analysis

Generally, any viable remote reprogramming solution consists of two independent parts; the code integration mechanism and the code dissemination mechanism. The code integration mechanism comprises a node's runtime and is responsible for

reprogramming the node, or integrating the newly received code with existing functionality. The code distribution mechanism, on the other hand, deals with the propagation of the code in the network. Only the former is relevant to the scope of this work.

We now briefly revisit and comparatively analyse existing literature.

4.1 Discussion

We can broadly divide the related research efforts into the following four categories.

4.1.1 Full-image replacement

These techniques such as Xnp (Jeong et al., 2003) or Deluge (Hui and Culler, 2004) operate by disseminating a new binary image of an application and the OS in the network. Since the image is compiled and linked afresh in every iteration, these solutions offer a very fine-grained control over the possible reconfigurations. However, these approaches result in excessive bandwidth overhead as unchanged parts of an application need to be re-disseminated in the network.

4.1.2 Differential image replacement:

Zephyr (Panta et al., 2011, 2009) and others (Jeong, 2004; Reijers et al., 2003) optimise the previous approach by disseminating only the changes between the already deployed executable in the network and the newly compiled image. However, often a small change in the source code can result in a large change in the compiled binary. For example, a small increase in the size of one component can result in an offset in memory locations for all the later instructions in memory. This can be circumvented by two mechanisms: either by placing 'elastic buffer zones' at regular intervals in memory or by rerouting the function calls through an indirection table. Zephyr takes the second approach. However, both of these approaches incur a penalty: first one causes the wastage of code memory on already resource constrained sensor nodes and the second one causes an extra indirection for every function in the source code. This results in an additional performance penalty as well as a memory wastage for storage of the additional indirection table. As these approaches work on the final monolithic binary image of the application, they fail to utilise high-level knowledge of the application structure.

4.1.3 Virtual machines (VM)

VMs such as Maté (Levis and Culler, 2002) or others (Müller et al., 2007; Brouwers et al., 2009; Fei and Magill, 2012) reduce the energy-cost of disseminating new functionality in the network as VM code is commonly more compact than the native code. VM based solutions also result in minimal post processing overhead to alter their functionality as per new supplied code. However, the performance penalty incurred due to instruction interpretation far outweighs the onetime energy saving in terms of the cost of dissemination. VMs also constrain the programmer within the boundaries of a developed language hence making fine-grained changes impossible (Fei and Magill, 2012). They also necessitate the user of a sensor network to learn a new tool to be able to program the network.

4.1.4 Dynamic operating systems

Dynamic OSs such as Contiki (Dunkels et al., 2004), SOS (Han et al., 2005) and FiGaRo (Mottola et al., 2008), provide the benefits of both image replacement and virtual machines i.e., fine grained code updates at low dissemination and run-time overhead. However, specific challenges remain: For example, SOS's design necessitates the use of position independent code, which, due to compiler limitations, is not fully supported on common sensor node platforms. Contiki allows only one-way linking for loaded modules and hence obligates more energy-intensive, polling-based service routines for interrupts. Moreover, Contiki's architecture restricts possible reconfigurations to application components only.

Commonly, dynamic OSes follow a clean slate approach which causes hindrance in their wide scale adoption. Two notable exceptions are FlexCup (Marrón et al., 2006) and TOSthreads (Klues et al., 2009), which are built on top of TinyOS. FlexCup offers dynamic adaptation for TinyOS based applications but lacks the support for new extensions to NesC, the TinyOS programming language, and employs nonstandard tools. As a result, the non-standard toolsets need to be ported to a wide range of development platforms, making maintenance and the roll-out of new features time consuming. Similar to Contiki, the TOSthreads library and its linker limit code replacement to high-level application components only. Moreover, it follows a polling based approach for kernel to application communication instead of NesC's well established, and more efficient event based approach. Additionally, it introduces a new interface for users, rendering it difficult to adopt. Concluding, both these approaches are not transparent for end users, lack the support for reuse of TinyOS code, and cause a substantial increase in the steepness of the learning curve.

In contrast to existing work on dynamic OSs, this paper shows how an existing and well established OS can be transparently transformed into a dynamic OS without following a clean slate approach or introducing new programming models.

4.2 Comparative analysis

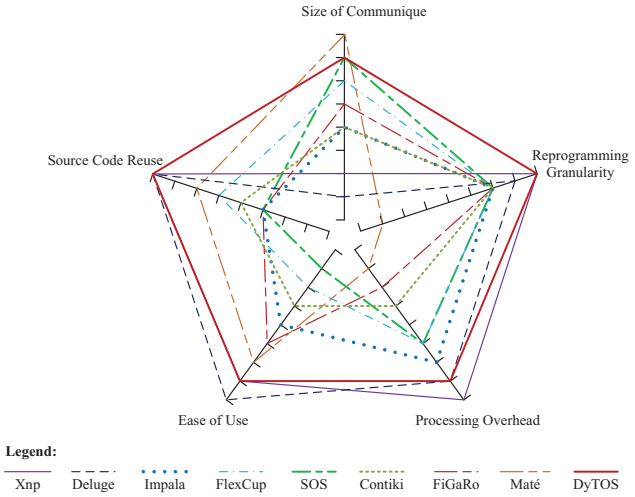
After reviewing the existing mechanisms and highlighting how DyTOS departs from these, we now comparatively analyse these mechanisms. This comparative analysis is qualitative, and is based on the set of requirements established in Section 1. The following discussion has been summarised in Figure 11 with the degree of optimisation/improvement increasing radically outwards.

4.2.1 Size of communique

Radio hardware is the most dominant energy consumer in sensor networks. Therefore, the communication overhead required for reprogramming a node can be a crucial factor

in determining the operational life of a network. Xnp and Deluge, since they are based on full image replacement, generate the maximum communication overhead. Whereas, virtual machines, such as Maté, generate the least. The dynamic operating system based solutions occupy a rather broad spectrum in this regard: Impala and Contiki need to transfer the complete application each time. FiGaRo, although does not transfer the whole application, its communique still includes the redundant parts of the ELF modules. FlexCup and SOS convert the application modules into the native hex format before transmission, and therefore generate less overhead. DyTOS also suffers from the redundancy toll but it fares better than FiGaRo. This is mainly because DyTOS further optimises the generated ELF (binary) files (cf. Section 2.3.1).

Figure 11 Comparative analysis of existing solutions with DyTOS (see online version for colours)



4.2.2 *Reprogramming granularity*

Fine grained and incremental reprogramming results in a more optimised network operation. Among the solutions discussed earlier, the image replacement mechanisms and virtual machines achieve the finest and coarsest reprogramming granularity, respectively. Dynamic operating systems occupy the middle of the spectrum; they need a constant node-runtime environment that must provide basic services to application components. DyTOS also includes a node-runtime that includes hardware drivers and basic service provision code, but it is possible to rewrite it with a newer version of itself. Due to its ability to overwrite any area of memory, DyTOS offers the same level of control on reprogramming as image replacement mechanisms.

4.2.3 *Processing overhead*

The processor's computation cycles are the second biggest drain on the limited energy reserves of a node. Image replacement, as it is based on a static binary image, results in the least processing overhead. Virtual machines, on the other hand, introduce maximum processing overhead as a result of byte-code interpretation. The dynamic operating systems result in a moderate overhead because they only process and incorporate parts of application in the form of new modules. Impala is the most optimised solution as it does not link the new application rather every application is required to provide a pre-specified, fixed offset API. DyTOS offers moderate processing overhead which basically results from rerouting of interrupts. Each ISR call in case of DyTOS suffers an additional delay of 26 clock cycles.

4.2.4 *Ease of use*

Image replacement typically does not require any significant effort on the part of the developer since no new programming constructs are introduced. Virtual machines are rather easy to use, however, one needs to learn the machine-specific scripting language. Among the dynamic operating systems, FiGaRo is the easiest to use as it provides a very intuitive 'macros' based interface to the programmers. Moreover, it has a graph traversal algorithm that automates the initialisation of the components and execution of application. Impala and Contiki both introduce new APIs for remote reprogramming. FlexCup employs NCC compiler for generating binary components, which is not trivial since the source code has not been released. SOS offers a very complex and extensive API for component generation and hence requires maximum learning effort. DyTOS in this regard does not necessitate the use of any special language constructs or any specialised tools. The user only need to specify the boundaries to partition any application. This is also done in the same language in which the application is written. We approximate the ease of use of DyTOS as similar to the image replacement solutions.

4.2.5 *Source code reuse*

Reuse of existing source code for sensor nodes is beneficial for two main reasons. First, it avoids duplicate development effort. Secondly, robustness of the system is enhanced as already tested source code is utilised. Among the presented solutions the image replacement mechanisms 'recycle' the existing sources and hence maximise its reuse. Matè and FlexCup use the existing sources with slight modifications. Contiki, although introduces a clean slate approach, has accumulated a seasoned code-base over the past few years that is compatible with its reprogramming mechanism. SOS and FiGaRo introduce new API and hence the resources from the existing solutions cannot be utilised without manual modifications. Impala is the least graded in this category because it is a clean slate approach and has not been tested on any of the mainstream sensor node platforms. DyTOS integrates seamlessly with the code repositories of TinyOS and can use all the existing source code without modifications.

4.3 *Summary*

A recent report from market research ON World (http://www.onworld.com/html/newsresearch.htm) indicates that the market for sensor networks is expected to grow tenfold and attain an absolute value of $1.3 billion. The same agency reports that 'ease of programming' is the major barrier to the

adoption of wireless sensor networks. The majority of the aforementioned, existing solutions fall short on this aspect. Therefore, a new submerged system which presents old and familiar interfaces but employs the state-of-the-art in design can be a step forward. TinyOS is almost a decade old, and being the most widely used OS for wireless sensor networks, presents the ideal foundation for such a system. We based our work in this belief and presented a solution in the form of DyTOS.

5 Conclusions

In this paper, we presented DyTOS as a flexible, transparent and efficient code update mechanism for sensor networks. It offers the functionality and performance required for fine grained remote adaptation of sensor applications. The presented system is tightly integrated with TinyOS resulting in ease of adoption and reuse of the seasoned TinyOS code repository. DyTOS achieves its objectives by enhancing TinyOS's compilation and runtime process model without introducing new programming constructs. Thus, it remains transparent to an application developer.

Our evaluation highlights the superior performance characteristics of DyTOS when compared with the state-of-the-art. The runtime functionality of DyTOS only utilises 7.7% of RAM and 32% internal ROM, which is a significantly smaller memory footprint than other comparable solutions. The size of an update in the case of DyTOS is limited to the nesC component that has been modified. DyTOS allows user desired level of reprogramming granularity with different performance characteristics, i.e., multiple components can be grouped into a single update block or a single component can further be divided into multiple update block to achieve the desired reprogramming granularity.

Reducing the communication overhead by further optimising ELF files, minimising the runtime overhead of Tiny Manager, and providing a better tool support, e.g., an XML based wiring to specify the boundaries of application components, are the next steps in the evolution of DyTOS.

References

Bohli, J-M., Hessler, A., Ugus, O. and Westhoff, D. (2009) 'Security enhanced multi-hop over the air reprogramming with fountain codes', *LCN*, pp.850–857.

Bohli, J-M., Hessler, A., Maier, K., Ugus, O. and Westhoff, D. (2011) 'Dependable over-the-air programming', *Ad Hoc and Sensor Wireless Networks (AHSWN)*, Vol. 13, pp.313–340.

Brouwers, N., Langendoen, K. and Darjeeling, P.C. (2009) 'A feature-rich vm for the resource poor', *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems, SenSys '09*, ACM, New York, NY, USA, pp.169–182.

Deng, K. and Nickerson, B.G. (2013) 'A fuzzy control framework for wireless sensor networks', *Int. J. Sen. Netw.*, Vol. 13, No. 1, March, pp.1–12.

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

Fei, X. and Magill, E. (2012) 'Reed: flexible rule based programming of wireless sensor networks at runtime', *Comput. Netw.*, Vol. 56, No. 14, September, pp.3287–3299.

Gnawali, O., Fonseca, R., Jamieson, K., Moss, D. and Levis, P. (2009) 'Collection tree protocol', *Proceedings of the 7th ACM Conference on Embedded Networked Sen-sor Systems (Sensys)*, Berkeley, California.

Han, C-C., Kumar, R., Shea, R., Kohler, E. and Srivastava, M. (2005) 'A dynamic operating system for sensor nodes', *MobiSys '05: Proceedings of Third International Conference on Mobile Systems, Applications and Services*, Seattle, Washington, pp.163–176.

Hui, J.W. and Culler, D. (2004) 'The dynamic behavior of a data dissemination protocol for network programming at scale', *SenSys '04: Proceedings of the 2nd ACM Conference on Embedded Networked SENSOR Systems*, Baltimore, MD, USA, pp.81–94.

Jeong, J. (2004) 'Incremental network programming for wireless sensors', *Proceedings of the First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks IEEE SECON*, pp.25–33.

Jeong, J., Kim, S. and Broad, A. (2003) *Network Reprogramming*, http:// www.tinyos.net/tinyos-1.x/doc/Xnp.pdf

Juang, P., Oki, H., Wang, Y., Martonosi, M., Peh, L.S. and Rubenstein, D. (2002) Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet', *ASPLOS-X: Proceedings of the 2nd ACM International Conference on Wireless Sensor Networks and Applications*, ACM, October, No. 10 in 37, pp.96–107.

Klues, K., Liang, C-J.M., Yeup Paek, J., Musaloiu-E., R., Levis, P., Terzis, A. and Govindan, R. (2009) 'TOSThreads: safe and non-invasive preemption in TinyOS', *Sensys '09: Proceedings of the 7th International Conference on Embedded Networked Sensor Systems*, ACM, Berkely, California, USA.

Levis, P. (2012) 'Experiences from a decade of tinyos development', *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12*, Berkeley, CA, USA, pp.207–220.

Levis, P. and Culler, D. (2002) 'Maté: a tiny virtual machine for sensor networks', *ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, San Jose, California, USA, pp.85–95.

Levis, P., Patel, N., Shenker, S. and Culler, D. (2004) 'Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks', *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, pp.15–28.

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

Mottola, L., Picco, G.P. and Sheikh, A.A. (2008) 'Figaro: Finegrained software reconfiguration for wireless sensor networks', *EWSN '08: Proceedings of the Fifth European Workshop on Wireless Sensor Networks*, Bologna, Italy, pp.286–304.

Müller, R., Alonso, G. and Kossmann, D. (2007) 'A virtual machine for sensor networks', *SIGOPS Oper. Syst. Rev.*, Vol. 41, No. 3, pp.145–158.

Munawar, W., Alizai, M.H., Landsiedel, O. and Wehrle, K. (2010) 'Dynamic tinyos: modular and transparent incremental code-updates for sensor networks', *ICC*, On World. ON World – In The News, pp.1–6.

Panta, R.K., Bagchi, S. and Midkiff, S. (2009) 'Zephyr: efficient incremental reprogramming of sensor nodes using function call indirections and difference computation', *USENIX '09: Annual Technical Conference*, San Diego, CA, USA.

Panta, R.K., Bagchi, S. and Midkiff, S.P. (2011) 'Efficient incremental code update for sensor networks', *ACM Trans. Sen. Netw.*, Vol. 7, No. 4, February, pp.30:1–30:32.

Polastre, J., Szewczyk, R. and Culler, D. (2005) 'Telos: enabling ultra-low power wireless research', *IPSN '05: Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, Los Angeles, California, p.48.

Pompili, D., Melodia, T. and Akyildiz, I.F. (2006) 'Deployment analysis in underwater acoustic wireless sensor networks', *WUWNet '06: Proceedings of the First ACM International Workshop on Under Water Networks*, ACM, Los Angeles, CA, USA, pp.48–55.

Reijers, N. and Langendoen, K. (2003) 'Efficient cosde distribution in wireless sensor networks', *WSNA '03: Proceedings of the 2nd ACM International Conference on Wireless Sensor Networks and Applications*, ACM, San Diego, CA, USA, pp.60–67.

Stathopoulos, T., Heidemann, J. and Estrin, D. (2003) *A Remote Code Update Mechanism for Wireless Sensor Networks*, Technical Report, UCLA, Los Angeles, CA, USA.

Szewczyk, R., Osterweil, E., Polastre, J., Hamilton, M., Mainwaring, A. and Estrin, D. (2004) 'Habitat monitoring with sensor networks', *Commun. ACM*, Vol. 47, No. 6, pp.34–40.

Tsiftes, N., Dunkels, A., He, Z. and Voigt, T. (2009) 'Enabling large-scale storage in sensor networks with the coffee file system', *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks, IPSN '09*, IEEE Computer Society, Washington, DC, USA, pp.349–360.

Werner-Allen, G., Lorincz, K., Welsh, M., Marcillo, O., Johnson, J., Ruiz, M. and Lees, J. (2006) 'Deploying a wireless sensor network on an active volcano', *IEEE Internet Computing*, Vol. 10, No. 2, pp.18–25.

Yick, J., Mukherjee, B. and Ghosal, D. (2008) 'Wireless sensor network survey', *Comput. Netw.*, Vol. 52, No. 12, pp.2292–2330.

Website

On World. *ON World – In The News, 2009*, http://www.onworld.com/html/newsresearch.htm