Online Code Compression in Wireless Sensor Networks

Igor Talzi and Christian Tschudin
Computer Science Department
University of Basel
CH-4056, Basel, Switzerland
firstname.lastname@unibas.ch

ABSTRACT

A problem in Wireless Sensor Networks (WSN), an extremely resource-limited system, is that re-tasking by replacing complete or partial code images is both disruptive and energy intensive. In this paper we report on a dynamic code compression scheme for mobile code that we implemented for a WSN. We use a fine-grained code mobility scheme based on capsules, ChameleonVM, where network functionality can be deployed on demand and for several tasks in parallel. Within a task's virtual network segment we let our ChameleonVM optimize the assignment of instruction bits: Mobile code instructions are re-coded on a regional basis and depending on the actual code sequences used, incrementally leading to smaller capsules. We demonstrate the operation of our online compression scheme with a time-sync protocol and discuss its performance.

KEYWORDS: Wireless Sensor Networks, Code Compression, Run-Time Optimization, Resource-Aware Protocols

1. INTRODUCTION

As an embedded platform, WSN has several limitations: little memory (RAM, Flash), weak CPU, battery-operated and short radio communication range. This is shown in Table 1 for some widely used modern WSN platforms. These limitations are caused by the need to keep the form-factor of WSN nodes small and nodes themselves cheap.

The lack of available resources has an effect on the software architectures used by those platforms. Many properties of PC-oriented operating systems (OS) are not required and therefore are excluded (e.g. virtual memory). Real-time support is normally not needed either. The main design requirements therefore are small memory footprint and energy awareness. TinyOS [1] achieves this by including programs into OS image at compile-time and replacing multi-threading with an event-driven programming model. The resulting image becomes relatively small which allows it to fit in almost all available hardware platforms.

Table 1. WSN Hardware Platforms

Platform	MCU	Prog+Data +Ext Mem [KB]	Radio	Sleep/ Awake Power [mW]
MicaZ	ATmega 8-bit	128 Prog, 4 EEPROM, 512 Serial Flash	Chipcon CC2420 2.4GHz	0.05/141
BTnode rev3	ATmega 8 MHz @ 8 MIPS	128+4, 240 SRAM	ChipCon CC1000 433-915Mhz + Bluetooth	9.9/198
iMote2	XScale 13-416MHz	256kB SRAM, 32MB FLASH, 32MB SDRAM	Chipcon CC2420	3.3/192.5
TelosB (Sky)	TI MSP430 16-bit	48 Prog, 10 SRAM, 1M Serial Flash	ChipCon CC2420	0.02/69
Shockfish TinyNode 584	TI MSP430	48 Prog, 10 SRAM, 0.5M Serial Flash	Xemics XE1205 868Mhz	0.02/231
Sun SPOT	180 MHz 32 bit ARM920T	512 RAM, 4M Flash	2.4 GHz IEEE 802.15.4	0.13/252

Many WSN systems are supposed to work in harsh remote environments in an unmanned fashion for a period of several years. Based on our experience [5] programming of such systems is error-prone, reprogramming is generally needed. System functionality might have to be changed not only because of the errors but also if the system profile changes – a node should be re-tasked to perform a different job. With pure TinyOS this is possible only if one replaces the entire image [15]. In OS which support run-time linking of loadable modules (ContikiOS [2] [14], SOS [3]) it can be done via module exchange; the application should initially be designed in a "lego"-manner.

To overcome this functional constraint a considerable number of middleware for WSN have been proposed. Middleware uses high-level programming abstractions to describe application functionality. The problem with these solutions is that they offer only domain-specific functions limited by middleware design. Middleware abstractions are normally fixed at high-level and cannot be changed by the application.

Another approach which tries to solve the issue of re-programming WSN uses execution environments, tiny virtual machines (VM) which reside on a node. Similar to the middleware approach these solutions introduce high-level programming abstractions which allow description of network interactions using fewer instructions (e.g. using single "send" instruction instead of making a system call with multiple parameters). This is a more general approach compared to middleware and provides more flexibility for an application. Depending on the design some VM use platform-specific code (e.g. Scylla [12]), others their own interpretive byte-code (e.g. Maté [7]), and hybrid solutions exist where native code is exposed as VM instructions (e.g. DVM [11] in SOS). Program updates are distributed using either OS-level protocols or a specific dissemination layer (e.g. Trickle [16] in Maté).

The availability of VM led to applying the concept of mobile agents to WSN domain. Agilla [10] extends Maté functionality allowing code to make decisions on where it would like to migrate to. Also, the code can now carry its execution context (state) along.

Mobile code compared to its static counterpart significantly increases the amount of traffic which might cause collisions with data flow unless a backbone network like [17] is used. Moreover, in WSN the biggest amount of energy is spent on radio transmissions. Transmitting one bit might be equivalent to thousands of CPU cycles producing that bit [21]. In this paper we focus on developing a method of changing mobile code representation depending on the task being performed. One of the biggest challenges is to make it fit resource-constrained WSN nodes. At the same time (de)compression must allow the code to meet timing requirements, i.e. provide an on-the-fly operation.

The rest of the paper is structured as follows: in Section 2 we refer to some existing work in the field; Section 3 discusses re-programming and re-tasking mechanisms used to modify network behavior; in Section 4 we present the main functional part of our re-tasking framework – ChameleonVM; Section 5 is dedicated to the compression algorithm and Section 6 to its evaluation; we conclude our work with discussing further directions in Section 7.

2. RELATED WORK

Traditionally, code compression is seen as compacting pre-compiled code (binary or any form of byte-code). We consider it as a problem of changing code representation at run-time by modifying (optimizing) the instruction set. We apply this technique to our own byte-code used within ChameleonVM, a WSN re-tasking framework. This eventually leads to code size reduction. Three main approaches exist:

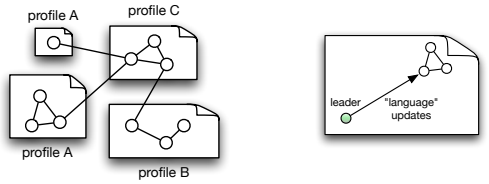
Hand-tuned ISA which is most commonly used in CISC and DSP worlds. This method reduces instruction size by designing a compact ISA based on operation frequencies. It makes the ISA more complex and the decoding stage more expensive. It also makes the ISA domain specific and eventually hampers compiler optimizations. Another disadvantage is that ISA becomes inflexible for any future extensions.

Ad-hoc ISA. This approach typically specifies two instruction modes: compressed and uncompressed (e.g. MIPS16, ARM-Thumb(2)). The compressed mode consists of alternative shorter (16 instead of 32) versions of the most commonly used instructions. Switching between modes is done through a special instruction. The main advantage is that decoding is simple and fast since instructions stay compressed in cache. However, decompression is on the critical path and compression rates are low. Mostly used in RISC embedded processors. The compression factor can be up to 20-30% and highly depends on the code, with a performance drop of 15%.

Fully customized ISA introduces a task-specific instruction set (e.g. in hardware: Xtensa, ARCompact; in software: ASVM [8] [9] based on Maté). The ISA is modified offline and tailored to perform a particular task. Any further changes are not possible. This approach gives the best performance gain.

Software dictionary-based code compression schemes exist (e.g. PowerPC CodePack [20], various adaptations for VLIW [19] architectures). These methods focus on compacting a program image before uploading it to a target where decoding is normally implemented in hardware.

Attempts to apply data compression methods (AC, GZIP, LZARI, LZO1X, SBZIP, VCDIFF, etc) directly to the binary executables [13] result in compression rates of only $\sim$50%. This is due to a different nature of code streams compared to data sets. Moreover, these methods require a lot of memory (most of them do not fit into TmoteSky); they are not adaptable and the decompression phase is usually



(a) Virtual Segmentation and Overlapping Profiles

(b) Single Profile Scenario

Figure 1. Virtual Segmentation and Network Profiles

long (1-15 seconds).

Our approach was originally developed for code streams and can be classified as creating a task-specific ISA at runtime without interrupting system operation.

3. RE-PROGRAMMING WSN

WSN re-programming is performed in order to change the behavior of a deployed system. Remote re-programming is especially important for environment monitoring WSN which can be difficult to reach. The following methods are used: 1) entire image replacement (Deluge [15] in TinyOS), 2) updates distribution via module exchange (ContikiOS, SOS), 3) middleware, 4) virtual machines and 5) mobile agents. The first two methods disseminate native pre-compiled code. The latter three approaches use a specific language in a form of byte-code or scripts, and rather belong to the class of re-tasking methods.

In our work we extend the re-tasking methodology by allowing the system to find an optimal code representation for a particular task. Tasks are grouped in a set of *profiles*. The system can switch between profiles at run-time. For each profile the system finds a separate optimized (compressed) form. Multiple profiles can co-exist. Overlapping of profiles is allowed as shown in the Figure 1a. We refer to the situation when the network has multiple, simultaneous profiles on different segments as *virtual segmentation* – the network is divided into virtual (not necessarily physically connected nodes) segments each of which implements a particular profile.

By introducing virtual segments we can optimize each profile independently. In this paper we focus on how each profile can be brought to an optimal representation: mobile code running in a profile is re-encoded at run-time in order to reduce its size and eventually the amount of transmitted code. To keep things simple we consider a single-profile scenario and apply our code compression algorithm to it (see Figure 1b). We use a centralized approach to control the process; a single leader is elected to run the compression and updates are sent around the network. Eventually, the system should learn and start using a better representation for the code moving across the network; nodes agree on a new, compressed "language" on-the-fly. As an example in Section 6 we show a simple time synchronization protocol which represents a profile. However, in other applications profiles can be compound and consist of multiple, independent tasks.

4. CHAMELEONVM

Before describing our code compression method we introduce the WSN re-tasking environment it is used in. At the heart of our WSN re-tasking solution lies a custom byte-code interpreter called ChameleonVM. Some design principles are similar to Maté [7] although a number of distinctive features have been introduced. In general, ChameleonVM was designed for resource-constraint devices (WSN nodes, wireless routers) to provide an execution environment for applications using the mobile code concept. The design focus was to provide an interaction with existing network stacks and code manipulation.

ChameleonVM is a single stack-based VM – operands are taken from the stack and results are pushed back on the same stack. Programs for ChameleonVM are distributed in the form of *capsules*. Capsules are small ($\sim$40 bytes) self-sufficient fragments of code which flood the network. Capsules can carry small portions of data as well. Programs can be composed of multiple capsules if the size limitation does not allow it to fit into a single capsule. ChameleonVM does not provide an underlying code propagation layer – programs should either rely on existing code deployment functionality or use the in-built features to propagate themselves. Upon delivery capsules are installed on a node, scheduled and executed locally by the pre-installed ChameleonVM instance as shown in Figure 2.

One of the central parts of the design is the code (de)compression engine discussed in detail in Section 5. The compression scheme works on our custom form of byte-code which essentially represents the ChameleonVM instruction set architecture (ISA). Our ISA includes the following classes of instructions: 1) stack (e.g. `push`, `pop`), 2) arithmetic (e.g. `add`, `div`), 3) binary/logic (e.g. `and`, `xor`), 4) control (e.g. `jmpeq`), 5) communication (e.g. `send`), 6) memory access (read/write), 7) capsule manipulation (merge/split/clone) and 8) aliases (dictionary records). Classes 1-7 are basic instructions available on each ChameleonVM installation. Class 8 (aliases) are instructions newly created by the system at run-time and

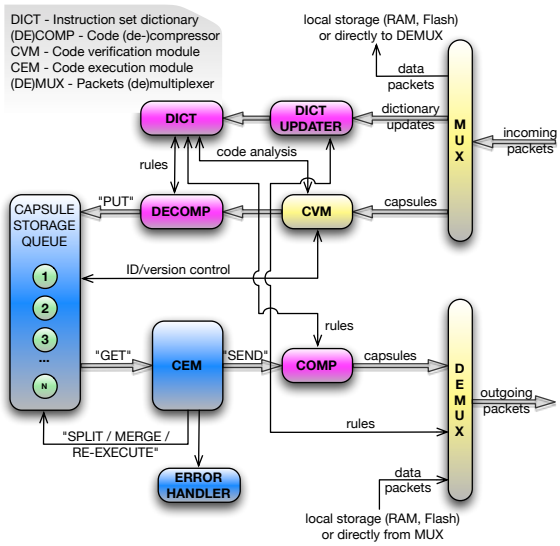


Figure 2. ChameleonVM's System Architecture

stored in the dictionary. Aliases can refer to a sequence of basic instructions, other aliases or an external OS-level function (a system call or a library function).

ChameleonVM features several memory regions: an in-capsule data segment (denoted as `BUFC`; can be accessed only by the capsule's code; is carried along with the capsule), heap (denoted as `BUFS`; resides on a node; can be accessed only by the capsule's code), shared memory (denoted as `SHMEM`; resides on a node; can be shared between capsules) and capsule storage (used to store capsules; not accessible from an application).

Communication properties of ChameleonVM include sending and receiving data packets and capsules as well as extracting network status information from the underlying protocols and execution environment (e.g. node id, number of successfully received packets, etc). Therefore, if the necessary system information is available, communication can be done in a high-level manner, e.g. "go one level up in the spanning tree".

Code manipulation is done using `merge`, `split` and `clone` operations. These allow the breaking of capsules into pieces at specified locations within the code, re-assembling pieces together to create capsules with new functionality and generating multiple copies of a single capsule.

Each capsule features an id, a version number and various system flags used to manage and distribute code inside the network. Capsules have limited lifetime: when it expires the capsule is destroyed. This allows the creation of dynamically renewed applications.

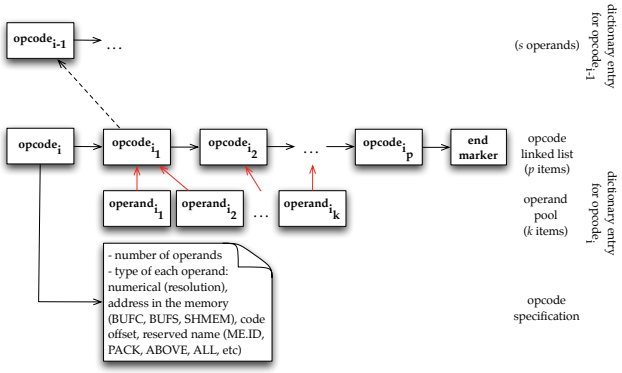


Figure 3. ChameleonVM's Dictionary Structure

Support for external protocols is provided through information exchange via memory (external protocols and capsules can read/write the same memory regions). ChameleonVM provides a set of memory access calls.

Capsule execution follows an event-based model. A capsule can contain the following code segments which are executed independently in response to the corresponding event: 1) initialization (executed only once after the capsule has been installed on a node), 2) data packet received, 3) another capsule received, 4) timer fired or 5) custom event.

All instructions (basic ones and aliases) are accessed through a dictionary. The dictionary has a similar structure as shown in Figure 3. A copy of the dictionary is held on each node and is updated via commands. For this paper we use a centralized version of the method: one node is elected as a leader which runs compression and sends updates out to other nodes. A distributed version exists where decision making is made on a peer-to-peer basis.

The following parameters are used to define each instruction: opcode, linked instructions (a sequence, linked list of pointers, of instructions which must be executed when this opcode is to be executed), number of arguments, argument type, various system flags (e.g. on/off protection from a change). Dictionary updates are sent in the form of commands affecting the local copy of the dictionary on each node. The following commands are supported: "add" (40 bits), "remove" (9 bits), "update" an instruction (40 bits), switch on/off protection (10 bits). Each time only a change (delta) is distributed, not the entire dictionary. Periodical updates from guided nodes are collected to ensure integrity of their local dictionary copies.

ChameleonVM can be used to exchange network protocols

on-the-fly, as well as to re-task the network at the application level. In Section 6 we show how to dynamically introduce an additional network service (time synchronization) to a deployed and running system. After doing so, we show how the system can self-optimize by applying the code compression technique from Section 5.

5. ONLINE CODE COMPRESSION

(De)compression is used by ChameleonVM in order to reduce code size which subsequently has a positive effect on energy saving, a critical aspect of most WSN systems. Our compression scheme continuously works at run-time while the system is performing its main task (e.g. sensing and collecting data). The dictionary is growing, moving semantics from the mobile code to the on-board storage. Graphically the process of how the system evolves over time is shown in Figure 4.

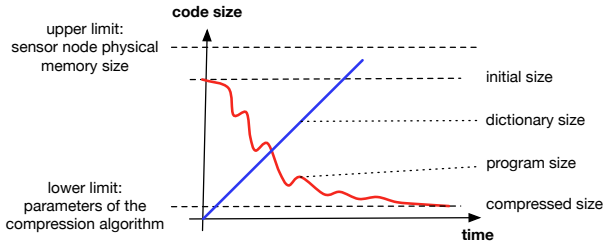


Figure 4. Compression Process

Each node holds its own copy of the dictionary containing the definitions of the current instruction set. By committing changes to the dictionary, code is constantly being rewritten, gradually reducing code size. When the configuration of the software changes (software pieces are added, removed or exchanged) the system is able to reflect these changes in encoding.

The (de)compression engine is an integral part of ChameleonVM design as shown in Figure 2. Incoming code must be decompressed before it can be executed. Only outgoing capsules are compressed, code which is executed locally stays untouched. (De)compression is done according to the rules stored in the on-board dictionary. This mechanism allows the system to modify encoding internally: no pre-deployed dictionary is required, nor full dictionary exchange.

In order to keep the memory footprint small a very compact dictionary structure is used (see Figure 3). Dictionary updates are also compact (max 40 bits). The decision on which rewriting rule to introduce next is made by the system itself based on the observation of the current code set. A selection algorithm detects the next rule (possible opcode assignment) and is based on the idea of a pair-wise sliding window (see Figure 5), a simple weighting of instruction opcodes and their pairs in the code and on the current state of the dictionary.

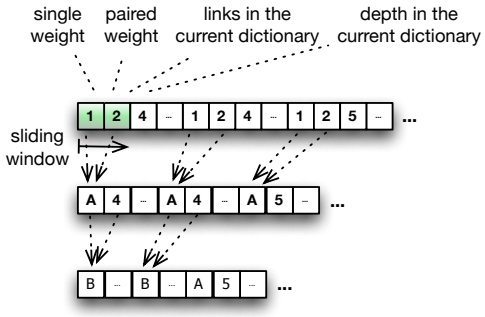


Figure 5. Sliding Window

The algorithm collects statistical information on the appearance frequency of instruction symbols and pairs of instruction symbols (e.g. probabilities) and allocates short codes to most frequent sequences. Such a scheme eliminates the need to have a complicated and computationally-abundant (de)compression run each time. Our design allows the system to gradually converge to a near-optimal code representation without carrying complex computations. It also allows most target applications to meet timing requirements since no long delays are introduced to the decoding phase (decoding is transparent to the execution engine).

For the supplied code stream we first calculate weights of each *single* instruction, then for each *pair* of consecutive instructions. Doing this we also keep track of how many links (appearance rate) each instruction already has in the dictionary as well as the level of nesting (depth). Based on this information one pair of instructions is picked. If multiple pairs have equal weights than a random selection is made. The selected pair is encoded as a new single instruction. Following this the original code stream is re-written (the selected pair is replaced with the new opcode) and the process repeats (see Figure 6).

6. EVALUATION

6.1. Impact of Varying the Compression Parameters

In order to estimate the efficiency of our algorithm we consider the influence of the following parameters:

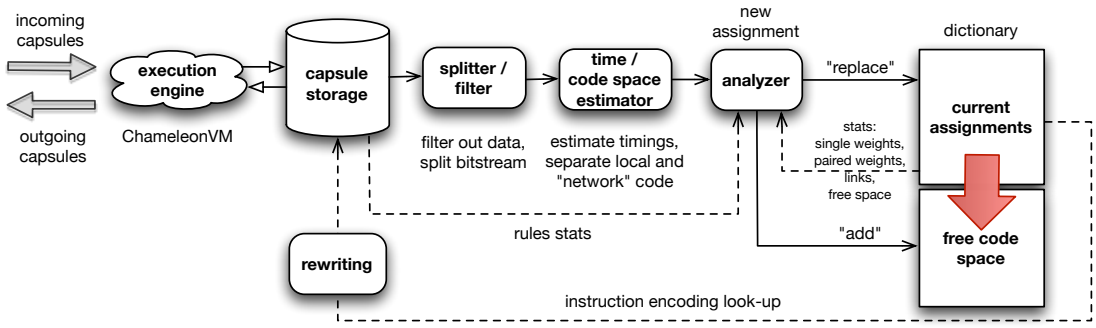


Figure 6. Compression Engine's Architecture

1) Program size (the size of code stream being compressed). A continuous code stream is defined by one single parameter (**N**) while a fragmented stream by two: number of fragments (**N**) and fragments length (**L**). Stream fragmentation is caused by having `split` instructions in the code.

2) Alphabet size (the number of unique symbols (instructions) in the original code stream (**A**).

3) Free code space (**F**). This parameter defines the dictionary capacity. By tuning it we can regulate up to the size we allow the dictionary to grow. Larger free code space gives more opportunities for the compression process to find a better representation for the code.

In theory, our approach should provide continuous and more fine grained model than batch-oriented schemes like LZW [18] which we use for comparison. In our experiments we vary one of the above parameters while keeping the rest fixed. Additionally, we vary the amount of duplicates (instruction appearance frequency). We show the results for 0% (fully random stream) and 50% of duplicates in Figure 7[1].

As can be seen in the Figure 7 the larger dictionary (**A**) gives a better compression factor (CF) as we have more freedom. The same is true for the free code space (**F**) which is the factor that gives most benefits on the compression side. Similarly, the longer stream (**N***L**) has a positive effect as we collect more statistical information. Fragmentation spoils the picture as with every new fragment the compression process is reset. In the worst case CF tends to 80%. In the best case 10% can be reached. In all cases we outperform LZW as our solution is better suited for short sequences.

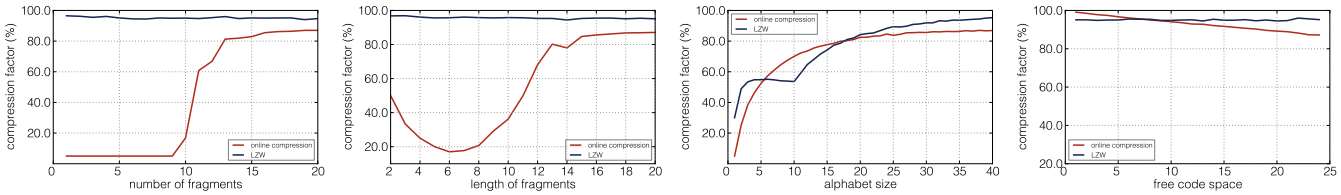
[1]We define the compression factor (ratio) as a lowering multiplier on the interval [0:1]: $compression\ \ factor\ \ (CF) = \frac{compressed\ code\ size}{original\ code\ size} * 100\%$ (the lower the ratio the better the compression).

6.2. Example of Compressing a Time-Sync Protocol Implementation

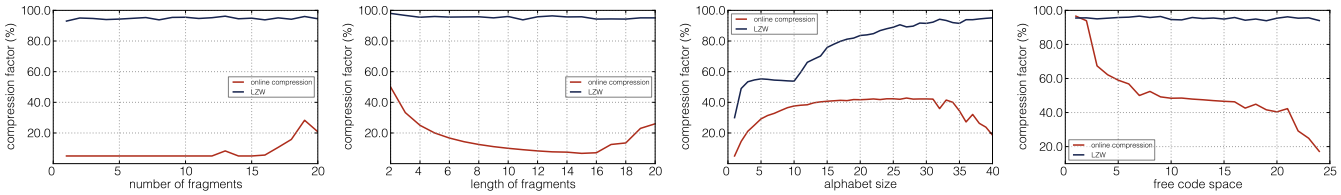
ChameleonVM was developed for TmoteSky under ContikiOS because of it's run-time module exchange feature. In order to evaluate our compression scheme we took, as an example, a light-weight time synchronization protocol [6] which we had developed for a real-world project [4] (at that time running on TinyOS-1.x). The pseudo-code for the protocol is shown in Appendix A.

The *Skew Balance Time Synchronization Protocol* achieves a collaborative clock drift compensation. Clock calibration is done post-hoc: This unloads complex computations of drift estimates from a sensor node and permits more detailed failure analysis, ultimately resulting in a more accurately calibrated time stamp assigned to each data packet. Local clocks are not adjusted. Instead, clock differences are recorded as events in the data stream (clock skew events) and used to reconstruct global time at the database side (a PC connected to the sink node). Clock *drift* has nevertheless to be corrected at run-time in order to let nodes periodically wake up at the same time to deliver data and exchange system information. Additionally, the protocol can handle extended periods of network partitioning that naturally occur in environmental monitoring. To this end, the protocol features a recovery strategy which is applied to fully desynchronized or newly joining nodes. Besides it's time sync capabilities, the protocol can be used as a simple TDMA-like MAC-layer.

The protocol is based on a common sync time window where each node has a time slot when it has to send a beacon message. This beacon message is broadcast and serves for: 1) letting other nodes measure how much skew their local clock has when compared to the sender, and 2) letting the node announce how much skew on average it measures. The beacon exchange process is shown in Figure 8a. Figure 8b illustrates how two nodes mutually measure their skew.

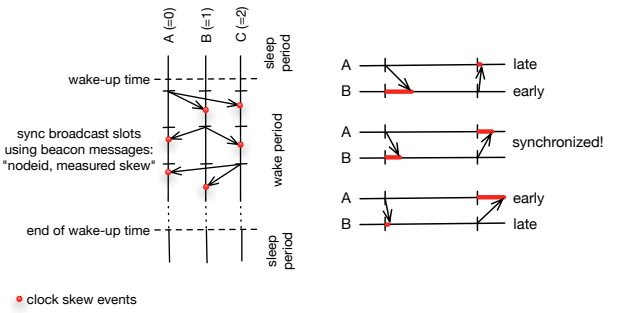


(a) 0% duplicates



(b) 50% duplicates

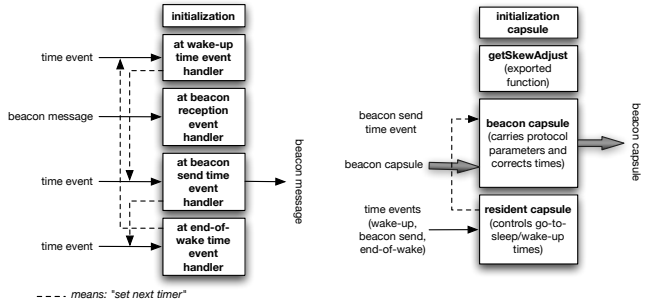
Figure 7. Variable Parameter vs Compression Factor



(a) The Sleep/Wake Cycle

(b) Two Nodes Mutually Measuring Their Skew

Figure 8. Skew Balance Time Synchronization Protocol: Working Principles



(a) Message-Based Implementation

(b) Mobile Code and ChameleonVM-Based Implementation

Figure 9. Skew Balance Time Synchronization Protocol: Implementation

The original implementation was based on the traditional message exchange mechanism (see Figure 9a). We have re-designed the protocol using the concept of mobile code and the programming tools provided by ChameleonVM presented in Section 4. If we compare these two versions we will see that the implementation based on code exchange (see Figure 9b) is more straight-forward and has a clearer structure; functionally the two versions are identical.

In the mobile code version, the protocol is functionally encapsulated into three capsules and an export function. The initialization capsule (see Listing 1 in Appendix B) sets up necessary variables and constants. A resident capsule resides on each node and switches the system between sleep and awake modes (Listing 3). The beacon capsule moves across the network and allows nodes to measure and correct their mutual skews (Listing 2). The export function `getSkewAdjust` implements a linear regression algorithm which is used to correct skews; it is called from the resident capsule.[2]

The init and resident capsules are disseminated only once. Moreover, the init capsules needs to be executed only once; after that it can be removed from a node. Capsule dissemination order matters: 1) resident capsules must be installed before init and 2) beacon capsules must be installed after init and resident. Both limitations can be overcome by switching the `AUTOEXEC` flag off for all the capsules and then triggering execution of the init capsule manually. Alternatively, capsule's presence can be checked with mobile code but this would increase the capsules' size. To simplify things we distribute a linear regression algorithm as a ContikiOS loadable module. It is then called from ChameleonVM capsules

[2]In the program listings, in order to save space we use a pseudo C-like code which can be easily translated into ChameleonVM notation (assembler-like stack-oriented language). `SHMEM[X]`, `BUFS[X]`, `BUFC[X]` denote the memory region where the corresponding variable or constant `X` resides.

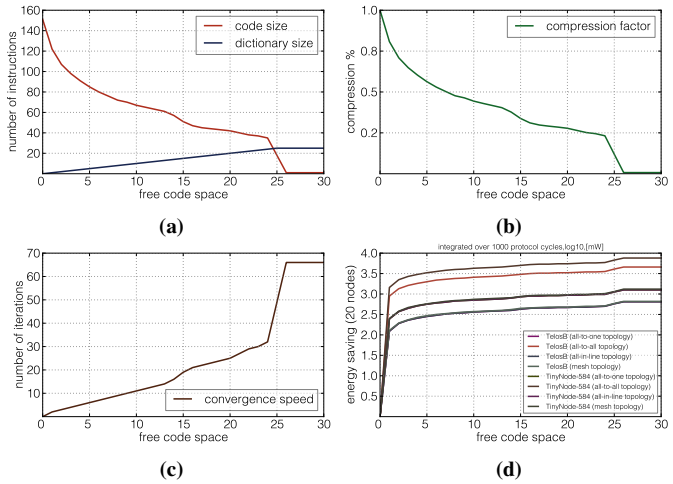


Figure 10. Code Compression Process: Skew Balance Time Synchronization Protocol

as export function through an instruction assignment.

The beacon capsule is moving across the network all the time in order to keep the nodes synchronized. The protocol uses local node IDs as a reference to the slot within the synchronization window (see Figure 8a: node *A* (id=1) transmits in the first slot, node *B* (id=2) in the second, *C* (id=3) in the third, and so on). Each node transmits exactly 1 and receives $N-1$ beacon capsules in every round. Previous capsules are replaced by newer ones, the last (out of $N-1$) arrived capsule is used by the node itself to participate in the synchronization round. As it can be seen from the code in Listing 2 the init segment of the beacon capsule plays the role of *message.received()* handler on the recipient side.

In the mobile code version, packet size increases significantly since now we transmit an actual code and not show data fields only. This indeed will have a negative impact on the energy spending in a long term. On the other hand, by applying compression to the traveling code we manage to almost half the packet size (see Table 2: 152 vs 72 bytes). Obviously, the static version still uses much smaller packet size (see Table 2: 4 vs 72 bytes) but if service flexibility is needed our solution is preferable. Since init and resident capsules are supposed to be disseminated only once they are not included in the compression process. Thus, we consider only the beacon capsule here. The performance results of its compression are shown in Figure 10[3].

The free code space has potentially the biggest impact on the compression factor. With 25 extra opcodes we manage to decrease code size by 95% (see Figures 10a–10b). This

[3]In this case the only thing the topology type defines is the number of packet transmissions within one round.

Table 2. Skew Balance Time Synchronization Protocol: Comparison of Static and Mobile Versions

	Static code (exchange of data packets)	**Mobile code (exchange of compressed ChameleonVM capsules)**
resident code size	2kB	108 bytes + 0.5kB (export function)
uncompressed mobile code size	—	152 bytes
packet payload size	4 bytes	72 bytes (fully compressed code + data)
dictionary size	—	25 entries

is also proportional to the number of iterations needed, i.e. convergence speed (see Figure 10c). In order to estimate the gain in energy saving we model the protocol behavior for 1000 cycles, 20 nodes, 2 modern widely used platforms (Tmote Sky and TinyNode) and various topologies. The simulation result is shown in Figure 10d. This takes into account packet transmissions only, the processor operation is not included.

7. CONCLUSIONS

In this paper a new method of dynamic code compression has been proposed. Our method is based on the use of specific byte-code whose representation is optimized at run-time. The mobile code paradigm brings a new way of building task-specific network configurations. It gives more flexibility but comes with an overhead of transmitted code. Our method tries to compensate this overhead. By extracting semantics from the code moving across nodes and putting it in the on-board dictionary, we manage to reduce the amount of information bits used to encode a program, e.g. the transmitted code size. This eventually has a positive effect on the energy consumption of a system. The method works ”online”, code is added to the compression process during system operation. Our approach requires a pre-installed VM on each node. The compression engine is an integral part of the VM. The selected design allows for integration of this VM into resource-constrained devices like WSN. The method shows better results if the dictionary size is big which means more memory usage. For that reason a very compact dictionary representation was designed. Our method assumes the entropy of the encoded stream to be low (which is true for code). Streams of fully random nature (collected data) should still use specific data compression methods (LZW, Huffman, etc).

Our work is still ongoing. In the near future, we would like to create a statistical energy model of the method in order to understand the influence of different code streams (different programs) on the performance of the system.

REFERENCES

[1] Philip Levis et al, "TinyOS: An Operating System for Sensor Networks", Ambient Intelligence. W. Weber, J. Rabaey, and E. Aarts (Eds.), Springer-Verlag, 2004.

[2] Adam Dunkels, Björn Grönvall and Thiemo Voigt, "Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors", Swedish Institute of Computer Science, 2004.

[3] Chih-Chieh Han, Ram Kumar Rengaswamy, Roy Shea, Eddie Kohler and Mani Srivastava, "SOS: A Dynamic Operating System for Sensor Networks", 2005.

[4] Igor Talzi, Andreas Hasler, Stephan Gruber and Christian Tschudin, "PermaSense: Investigating Permafrost with a WSN in the Swiss Alps", 2007.

[5] J. Beutel, S. Gruber, A. Hasler, R. Lim, A. Meier, C. Plessl, I. Talzi, L. Thiele, C. Tschudin, M. Wöhrle and M. Yücel, "PermaDAQ: A Scientific Instrument for Precision Sensing and Data Recovery under Extreme Conditions", ETH Zürich, 2009.

[6] Igor Talzi, "Technical Report CS-2008-002: Time Synchronization with Post-Hoc Calibration in Small-Scaled Intermittently Connected Wireless Sensor Networks". University of Basel, 2008.

[7] Philip Levis and David Culler, "Máte: A Tiny Virtual Machine for Sensor Networks", University of California, Berkeley, 2002.

[8] Philip Levis, David Gay and David Culler, "Active Sensor Networks", University of California, Berkeley, 2005.

[9] Philip Levis, David Gay and David Culler, "Bridging the Gap: Programming Sensor Networks with Application Specific Virtual Machines", University of California, Berkeley, 2004.

[10] C.-L. Fok, G.-C. Roman and C. Lu, "Agilla: A Mobile Agent Middleware for Self-Adaptive Wireless Sensor Networks", Department of Computer Science, Washington University in Saint Louis, 2009.

[11] Rahul Balani, Chih-Chieh Han, Ram Kumar Rengaswamy, Ilias Tsigkogiannis and Mani Srivastava, "Multi-level Software Reconfiguration for Sensor Networks", University of California, Berkeley, 2006.

[12] Stanley P. Marbell and L. Iftode, "Scylla: A Smart Virtual Machine for Mobile Embedded Systems", 2000.

[13] Nicolas Tsiftes, Adam Dunkels and Thiemo Voigt, "Efficient Sensor Network Reprogramming through Compression of Executable Modules", Swedish Institute of Computer Science, 2008.

[14] Adam Dunkels, Niclas Finne, Joakim Eriksson and Thiemo Voigt, "Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks", Swedish Institute of Computer Science, 2006.

[15] Jonathan W. Hui and David Culler, "The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale", University of California, Berkeley, 2004.

[16] Philip Levis, Neil Patel, David Culler and Scott Shenker, "Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks", University of California, Berkeley, 2004.

[17] Matthias Dyer, Jan Beutel, Thomas Kalt, Patrice Oehen, Lothar Thiele, Kevin Martin and Philipp Blum, "Deployment Support Network: A Toolkit for the Development of WSNs", ETH Zurich, 2007.

[18] Jacob Ziv and Abraham Lempel, "Compression of Individual Sequences Via Variable-Rate Coding", IEEE Transactions on Information Theory, 1978.

[19] Montserrat Ros and Peter Sutton, "Code Compression Based on Operand-Factorization for VLIW Processors", The University of Queensland, 2004.

[20] Charles Lefurgy, Eva Piccininni and Trevor Mudge, "Evaluation of a High Performance Code Compression Method", University of Michigan.

[21] Joseph Polastre, Robert Szewczyk and David E. Culler, "Telos: Enabling Ultra-Low Power Wireless Research", University of California, Berkeley, 2005.

APPENDIX A: Pseudo Code for Skew Balance Time Synchronization Protocol

```
VARIABLES:

  integer refTime
  integer currentSkew
  bool validSkewFlag
  circBuffer skewTable
  "skew measurement variables"
  "received skew variables"
  "constants (upper-case)"

EVENTS:

  at wake-up time (= refTime):
    reset skew measurement variables
    reset received skew variables
    start timer for sending beacon at
     "reftime + GUARD + myNodeId * SLOT"

  at beacon b=<id,validS,s> reception:
    accumulate measured skew = now() -
     ("reftime + GUARD + b.id * SLOT")
    if b.validS then
      accumulate received b.s
    fi

  at beacon send time:
    send beacon <myNodeId, validSkewFlag, currentSkew>
```

```
    start timer for end-of-wake at
     "reftime + 2*GUARD + SYNCWINDOW + DATA"

  at end-of-wake time:
    if any beacons received then
      skewTable.add(average(measuredSkews))
    else
      if missedTooManyBeacons then
        validSkewFlag = FALSE
        skewTable.reset()
      fi
    fi
    if skewTable.isFull() then
      currentSkew = skewTable.getAverage()
      validSkewFlag = TRUE
    fi
    if any valid skews received AND validSkewFlag then
      skewTable.adjust(diff/2)
      diff = currentSkew - average(receivedSkews)
      currentSkew = currentSkew + diff/2
      reftime = reftime + diff/2
    fi
    reftime = reftime +
     "GUARD + SYNCWINDOW + GUARD + DATA + SLEEP"
    sleepUntil(reftime)
```

APPENDIX B: Skew Balance Time Synchronization Protocol (mobile code version)

```
{ .code.init
  SHMEM[SY_NUMBER_OF_NODES] = 20;
  SHMEM[SY_GUARD_DURATION] = 50;        // 50 ms
  SHMEM[SY_SLEEP_DURATION] = 5*60*1024; // ~5 min
  SHMEM[SY_SLOT_DURATION] = 50;         // 50 ms
  SHMEM[SY_SYNC_DURATION] = SHMEM[SY_NUMBER_OF_NODES]
                           *SHMEM[SY_SLOT_DURATION],
  SHMEM[SY_CYCLE_DURATION]=2*SHMEM[SY_GUARD_DURATION]+
                            SHMEM[SY_SYNC_DURATION]+
                            SHMEM[SY_SENSE_DURATION]+
                            SHMEM[SY_SLEEP_DURATION];
  SHMEM[SY_BOOT_DURATION] = SHMEM[SY_CYCLE_DURATION];
  SHMEM[SKEW_MAXENTRIES] = 8;
  SHMEM[state] = STATE_BOOTING;
  call Timer0.startOneShot(SHMEM[SY_BOOT_DURATION]);
}
```

Listing 1. Initialization Capsule

```
{ .code.init
  BUFS[msrdSkewSum] = BUFS[msrdSkewCnt] =
   BUFS[rcvdSkewCnt] = BUFS[rcvdSkewSum] = 0;
  if(SHMEM[state] == STATE_BOOTING) {
    BUFC[nodeid] &= ~SHMEM[FLAG_MASK];
    SHMEM[state] = STATE_STOPPING;
    SHMEM[reftime] = call Timer0.getNow() -
     (SHMEM[SY_GUARD_DURATION] +
     BUFC[nodeid]*SHMEM[SY_SLOT_DURATION]);
    call Timer0.startOneShotAt(SHMEM[reftime] +
     2*SHMEM[SY_GUARD_DURATION] +
     SHMEM[SY_SYNC_DURATION]);
  }
  else if(SHMEM[state] != STATE_SLEEPING) {
    BUFS[msrdSkewSum] += call Timer0.getNow() -
     (SHMEM[reftime] + SHMEM[SY_GUARD_DURATION] +
     (BUFC[nodeid] &
     ~SHMEM[FLAG_MASK])*SHMEM[SY_SLOT_DURATION]);
    BUFS[msrdSkewCnt]++;
    if(BUFC[nodeid] & SHMEM[FLAG_VALIDSKEW]) {
      BUFS[rcvdSkewSum] += BUFC[skew];
      BUFS[rcvdSkewCnt]++;
    }
  }

  .code.timer0
  if(SHMEM[state] == STATE_SYNCWINDOW) {
    SHMEM[state] == STATE_STOPPING;
    call Timer0.startOneShotAt(SHMEM[reftime] +
     2*SHMEM[SY_GUARD_DURATION] +
     SHMEM[SY_SYNC_DURATION]);
    if(BUFS[skewCnt] == SHMEM[SKEW_MAXENTRIES]) {
      BUFC[nodeid] = call getNodeId() |
       SHMEM[FLAG_VALIDSKEW];
    }
    else {
      BUFC[nodeid] = call getNodeId();
      BUFC[skew] = 0;
    }
    send "BEACON capsule" to ALL
}}
```

Listing 2. Beacon Capsule

```
{ .code.timer0
  switch(SHMEM[state]) {
    case STATE_SLEEPING:
      SHMEM[state] = STATE_SYNCWINDOW;
      call Timer0.startOneShotAt(SHMEM[reftime] +
       SHMEM[SY_GUARD_DURATION] +
       getNodeid()*SHMEM[SY_SLOT_DURATION]);
      break;
    case STATE_BOOTING:
      if the beacon capsule is installed {
        SHMEM[state] = STATE_SYNCWINDOW;
        SHMEM[reftime] = call Timer0.getNow() -
         (SHMEM[SY_GUARD_DURATION] +
         getNodeid()*SHMEM[SY_SLOT_DURATION]);
        call "timer0" handler of the beacon capsule
        break;
      }
      else {
        call Timer0.startOneShot(
             SHMEM[SY_BOOT_DURATION]);
        break;
      }
    case STATE_STOPPING:
      SHMEM[state] = STATE_SLEEPING;
      getSkewAdjust();
      SHMEM[reftime] += SHMEM[SY_CYCLE_DURATION] +
       BUFS[adjustment];
      call Timer0.startOneShotAt(SHMEM[reftime]);
}}
```

Listing 3. Resident Capsule