POLITECNICO DI MILANO

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA

A POWER-EFFICIENT FRAMEWORK AND METHODOLOGY TO CONFIGURE RESOURCE-CONSTRAINED WIRELESS SENSOR NETWORKS

Relatore: Prof. Carlo BRANDOLESE

Tesi di Laurea di:
Luigi RUCCO
Matricola n. 707988

ANNO ACCADEMICO 2009-2010

...surely from this period of ten months this is the lesson: never give in, never give in, never, never, never, never-in nothing, great or small, large or petty.
Never give in except to convictions of honor and good sense.
Never yield to force.
Never yield to the apparently overwhelming might of the enemy.
W.S.Churchill

Dear Mom,

by the time this Thesis will be presented at the Final Dissertation exactly ten months are going to elapse since our hardest challenge has raised.
The words in Winston Churchill's speech seem to be the most appropriate synopsis for what has happened to You, to me, to our family in this period and for the resistance we have offered against it.
Cancer is certainly a dreadful beast. Eleven years have still to fly away from the day it stole from this event the presence of Dad and, having not placate its appetite of sorrows, ten months ago it retried in the dark purpose of depriving our family of its remaining pillar.
In another famous speech Winston Churchill was addressing *"'... whatever the cost may be, we shall fight on the beaches, we shall fight on the landing grounds, we shall fight in the fields and in the streets, we shall fight in the hills; we shall never surrender"'* and this is what we have done. Never the misfortune has had a chance to take a foothold and we have fought in the hospitals, we have fought on the streets that lead to these hospitals, we have never surrendered. Yes, it's true, we have fought! And the battle you have asked me to fight has been consumed on the academic books, on the IEEE publications, among the lines of source code, between the forces which led us down and the desire to not give in, not this time! Standing on brink of the disaster, we have had just two choices: to get it over with our dreams or to resist. And today, we can surely say, we resisted, having bent over backwards to accomplish with renewed commitment our academic duties and our everyday life.
I know that the Victory over Cancer is still a far-reaching target for medicine and it may will take years or decades before to call it a day, but it is already sure that

we have indeed obtained a great victory as well.
Because Cancer destroys a balance rather than a body and this balance is a melting pot of physical conditions, soul aptitudes, human relationships and family bounds: in blocking this vicious effect, we have perhaps deprived our enemy of its worst relapses and this is what mainly matters.
So, getting back to the origin, from this period of ten months this is the lesson: never give in, never give in, never, never, never!
Thank You Mom, Yours sincerely,

Luigi

Acknowledgments

My first and foremost thanks are deserved to Professor Carlo Brandolese for his great and kind commitment in addressing me and my work during this research. I would especially demonstrate my gratitude for the personal growth that has tagged along the professional enrichment brought about by this outstanding experience. The exceptional support by Professor Brandolese has strongly enhanced my passion for scientific and technological research and has also helped me in better tackling not only the academic topics, but also some personal troubles I have experienced during the last year.

I would like to thank Simone Corbetta for the kind and professional helpfulness he offered me in the first phase of the present work.

Contents

List of Figures

List of Tables

Abstract

Abstract - Italiano

La nostra ricerca è stata orientata alla definizione di una metodologia e di un modello, sia teorico che applicativo, per la riprogrammazione e la configurazione di wireless sensor networks, caratterizzate da quantità molto piccole di memoria disponibile da un lato e dalla necessità di una consistente autonomia energetica dall'altro. Per via dei requisiti di lunga durata temporale, l'intera impalcatura metodologica è stata basata sulla minimizzazione del consumo di potenza, preservando al contempo la completezza e l'efficacia funzionale.

Studiando una soluzione sperimentale per la riprogrammazione dinamica di WSN, all'interno del progetto europeo WASP [2], abbiamo definito un approccio molto efficiente dal punto di vista energetico, concepito come un ibrido tra il paradigma del "‘dynamic linker"’ e quello dei moduli "‘pre-linkati"’, immediatamente caricabili sul nodo. Questa soluzione sfrutta i vantaggi di basso consumo di potenza e di contenimento della memoria caratteristici del paradigma basato su moduli "‘pre-linkati"’, arginandone al contempo gli svantaggi in termini di scalabilità e disallineamento per mezzo di alcune soluzioni teoriche mutuate dall'approccio basato su "‘dynamic linker"’. Pur non essendo un lavoro pienamente maturo, i risultati sperimentali hanno dimostrato una prospettiva decisamente interessante per la futura adozione di questa tecnica nel progetto di reti di sensori wireless ad alta efficienza energetica ed efficacia funzionale.

Il secondo grande passo della nostra ricerca ha riguardato la creazione di un'astrazione matematica del dominio funzionale e non-funzionale di una wireless sensor network. Questo studio teorico ha portato alla creazione di un modello genetico evoluzionistico per configurare l'allocazione delle funzioni software sui vari nodi della rete. Al centro di questo modello vi è la possibilità di allocare le funzioni sia in maniera statica che dinamica, quest'ultima resa possibile dal meccanismo di riprogrammazione precedentemente illustrato.

Gli esperimenti sull'algoritmo genetico hanno dimostrato un'efficacia superiore a quella delle tecniche empiriche attualmente in uso per l'allocazione funzionale su reti di sensori wireless.

Abstract - English

Our research has been focused on defining a power-efficient framework, both at applicative and theoretical layer, to dynamically reprogramming and configuring wireless sensor networks, which provide very small amounts of available memory and require long-lasting lifetimes.

In studying an experimental solution for a dynamic reprogramming mechanism (within the *WASP European Project* [2]), we have defined a very power-efficient approach which has been conceived as an hybrid between a dynamic linker and pre-linked modules based paradigm. This solution takes the advantages in terms of low-power and memory requirements of the latter, while overcoming its lack of scalability and not-synchronized behavior by means of some theoretical features characterizing the dynamic linker approach. Though not being a mature work, the experiments have shown a promising perspective for future applications of this technique.

The second great step of the work has then concerned the creation of an accurate mathematical abstraction of both the functional and non-functional domains of a wireless sensor network. This theoretical inquiry has brought to the creation of an evolutionary genetic model for configuring WSNs' functional allocations. In doing that, we have considered the opportunity of allocating functions on a node both in a static and a dynamic way, the latter enabled by the dynamic reprogramming mechanism.

Experiments on the genetic model we have defined on this mathematical abstraction have demonstrated very good results and have pointed out a superior effectiveness of the algorithm with respect to configuration capabilities, compared to that provided by actual empirical techniques.

The Thesis is structured as follows: Chapter 1 provides an introduction to WSN and the motivations underneath our work. Chapter 2 reports a review on the state of art in WSN reprogramming. In Chapter 3 we present our dynamic linker/loader approach. Chapter 4 presents the Genetic Model we have defined. Chapter 5 summarizes the achievements and the considerations raised from the work.

Struttura ed Organizzazione della Tesi

La struttura della tesi segue il naturale flusso della ricerca, che partendo da uno studio generale sulla composizione e le caratteristiche delle reti di sensori wireless si è poi concentrato su un'attenta analisi della letteratura riguardante la riprogrammazione dinamica delle WSN come punto di partenza per la definizione di un meccanismo che potesse superare alcune delle limitazioni presenti negli approcci ad oggi disponibili. Successivamente, viene presentato il meccanismo di riprogrammazione dinamica da noi definito come una variante dell'approccio basato sul "'dynamic linker"' e di quello basato su moduli "'pre-linkati"'. In seguito viene presentato il modello genetico per l'ottimizzazione dell'allocazione funzionale in funzione della "'lifetime"' della rete, sia nei suoi aspetti teorici che implementativi. Infine vengono formulate le dovute conclusioni sulla ricerca e vengono discusse le linee di evoluzione e i futuri sviluppi del presente lavoro.

L'organizzazione dei capitoli segue pertanto questa struttura ed è articolata come segue: il Capitolo 1 fornisce un'introduzione al mondo delle wireless sensor networks e le motivazioni sottostanti al nostro lavoro di ricerca. Il Capitolo 2 riporta un'analisi dettagliata della letteratura e dello stato dell'arte rigu-radante la riprogrammazione dinamica delle reti di sensori wireless. Nel Capitolo 3 presentiamo il nostro approccio basato sul "'dynamic lynker/loader"' nelle sue caratteristiche teoriche ed implementative, unitamente ad una valutazione sperimentale della sua efficienza energetica. Il Capitolo 4 è interamente dedicato all'introduzione ed alla discussione del Modello Genetico da noi definito, proponendo una approfondita valutazione sperimentale delle sue prestazioni e delle sue funzionalità. Il Capitolo 5, infine, riassume le considerazioni e i risulati venuti alla luce a seguito del lavoro, unitamente ad una digressione sulle linee di ricerca future e i possibili sviluppi.

Estratto dei Capitoli

Di seguito proponiamo l'estratto di ogni singolo capitolo, riportandone i concetti principali e presentandone la struttura. Questa breve introduzione dei capitoli mira a fornire una panoramica esaustiva in lingua italiana sui contenuti generali della trattazione, rimandando ai capitoli corrispondenti per il dettaglio degli specifici argomenti.

Estratto del Capitolo 1

Nel primo capitolo della tesi viene inzialmente introdotta la storia delle prime Sensor Networks nate a ridosso della seconda guerra mondiale e sviluppatesi durante la guerra fredda, illustrando poi le varie evoluzioni che hanno portato alla nascita delle moderne Wireless Sensor Networks. Dopo questa digressione iniziale, viene discussa la generica struttura di una rete, individuandone i componenti costitutivi, i meccanismi di funzionamento ed i campi di applicazione.

Viene poi introdotta la struttura hardware di un tipico nodo sensore, inquadrando dunque il problema dei vincoli sulle risorse di sistema come fattore centrale per lo studio e la ricerca in questo ambito. Vengono inoltre schematizzati i principali nodi-sensori attualmente in commercio.

Successivamente, viene analizzato il problema della riprogrammazione dinamica dei sensori a livello software, come requisito fondamentale per l'efficienza e la completezza applicativa della rete. Vengono dunque individuati i fattori critici di successo proposti in letteratura per la progettazione e lo sviluppo di applicazioni finalizzate all'aggiornamento ed al caricamento di software nei nodi.

Infine, vengono proposte le motivazioni alla base del nostro lavoro di ricerca, analizzandole in maniera critica rispetto al percorso che ci ha portati a studiare prima una soluzione sperimentale per la riprogrammazione dei nodi e successivamente un modello genetico per l'ottimizzazione funzionale e la massimizzazione della "'lifetime"'.

Estratto del Capitolo 2

Il secondo capitolo è dedicato allo studio della letteratura sullo stato dell'arte nella definizione di meccanismi di riprogrammazione dinamica della rete. Con riprogrammazione dinamica si vuole indicare ogni meccanismo in grado di modificare, aggiornare o semplicemente cambiare le funzionalità software dei nodi.

Inizialmente vengono presentati gli approcci basati sulla sostituzione della immagine dell'intero sistema operativo. In questo ambito vengono discusse le caratteristiche dei protocoli di "'image replacement"' più usati come XNP, MNP e MOAP per la sostituzione integrale e Deluge che è invece un meccanismo per l'aggiornamento incrementale delle immagini sui nodi. Vengono infine accennati altri meccanismi di sostituzione dell'immagine nativamente supportati da molti dei sistemi operativi per WSN.

La seconda famiglia di meccanismi di riprogrammazione riportata è quella relativa all'aggiornamento differenziale dell'immagine di un nodo, principalmante per mezzo di patch binarie. Questo tipo di approcci sono di fatto rimasti confinati ad una dimensione puramente accademica e in questa sezione vengono presentate le proposte di Koshy-Pandey e Reijers-Langendoen.

Il passo successivo consiste nella descrizione di tecniche basate su linguaggi interpretati e pertanto vengono discusse le caratteristiche di un Interprete Python per WSN così come SensorWare, che è un modello basato sul linguaggio TCL. Sempre a proposito di linguaggi interpretati, ma con riferimento specifico alle "'virtual machines"', vengono successivamente presentati due maeccanismi molto efficienti, ovvero SwissQM e Maté, entrambi basati su macchine virtuali opportunamente ottimizzate per WSN.

I database distribuiti per WSN sono stati a loro volta oggetto di studio per via della grande flessibilitá che permettono nel riconfigurare le funzionalità della rete, per mezzo di semplici query. Viene approfondita in questa sezione la struttura del TinyDB, uno dei meccanismi in assoluto più usati e per meglio capirne la logica, che risulta di fatto essere molto articolata e complessa, l'intero paradigma è stato analizzato e comparato rispetto alla teoria dei Database canonica.

Una breve descrizione è riservata alle reti di supporto per la riprogrammazione, dopodiché vengono subito introdotti i meccanismi basati su agenti software come Agilla o una famosa proposta sviluppata dal alcuni ricercatori dell'Università della California Davis.

Le ultime due sezioni trattano infine le caratteristiche dei meccanismi basati rispettivamente su moduli pre-linkati e su dynamic linker, i cui punti di forza e di debolezza sono stati attentamente studiati per definire la logica ala base del nostro approccio, che verrà presentato nel successivo capitolo.

Infine viene proposta una comparazione critica dei vari modelli.

Estratto del Capitolo 3

L'intero capitolo è dedicato alla descrizione del progetto sperimentale di un meccanismo per la riprogrammazione dinamica di WSN, da noi intrapreso nell'ambito del progetto europeo WASP.

Inizialmente viene introdotto lo *scope* del progetto, fornendo una descrizione esaustiva della natura di WASP, della piattaforma hardware utilizzata –ovvero il TelosB di Crossbow – e sul sistema operativo Mantis. Queste premesse sono di fatto necessarie per comprendere le caratteristiche del sistema e le scelte progettuali adottate.

Viene poi introdotto il formato ELF (Executable and Linking Format), con riferimento alle specifiche sezioni considerate nella defінzione del linker dinamico da noi ideato.

La maggior parte del capitolo è poi dedicata alla descrizione del nostro approccio, proponendone nel dettaglio il modello teorico e le scelte implementative.

Viene poi proposta una valutazione sperimentale delle performance del dynamic linker/loader e vengono infine formulate le opportune coclusioni e considerazioni.

Estratto del Capitolo 4

Questo capitolo tratta l'altra grande fase della presente ricerca, che ha riguardato la definizione di un modello matematico, poi mappato su un algoritmo genetico, per l'ottimizzazione funzionale e la massimizzazione della "'lifetime"' della rete. Con ottimizzazione funzionale intendiamo la possibilità di scegliere la migliore allocazione possibile delle funzioni sui nodi, tale che garantisca la completezza del task applicativo e riduca al contempo il consumo complessivo di potenza massimizzando la durata della rete.

Grazie all'introduzionde del dynamic linker/loader, descritto nel capitolo precedente, ogni funzione è stata considerata come dinamicamente o staticamente allocabile su un nodo e lo scopo dell'ottimizzazione è proprio quello di trovare la combinazione di funzioni staticamente e dinamicamente allocate più efficiente dal punto di vista energetico.

Il modello viene prima presentato nella sua astrazione e formalizzazione matematica, discutendone il dominio di definizione, i vincoli e la funzione obiettivo nel dettaglio. Vengono poi riportate le scelte implementative adottate ed in particolare la trasposizione del modello matematico sull'aloritmo genetico e la sua implementazione per mezzo delle librerie GAUL, disponibili su ambiente Linux.

Viene poi proposta un'ampia e dettagliata valutazione sperimentale del modello, formulandone infine le opportune conclusioni e considerazioni.

Estratto del Capitolo 5

L'ultimo capitolo è dedicato ad alcune considerazioni conclusive sul lavoro di ricerca e sui risultati ottenuti. Vengono infine presentate le direzioni di ricerca future e i possibili sviluppi previsti per il presente lavoro.

Chapter 1

Introduction

The omnium-gatherum of electronic computing devices, in their more disparate sizes and applications, are day after day invading our ordinary life, putting into concrete form the theoretical paradigm of *pervasive computing*. The interconnection of all these devices through Internet interfaces could become hypothetically world-covering, if the ipv6 protocol will actually creep in.

In proceeding on the above-quoted process, we are slowly laying the foundation for a gigantic Worldwide Brain, whose memory is composed by large databases and web repositories, whose neurons are intelligent and independent human-computer interactions and whose synapses are nothing else but the network connections themselves. In this perspective, Wireless Sensor Networks are equivalent to nerve-endings, used by the Brain to retrieve information from Real World. Following we propose a brief introduction to the WSNs architectures and platforms, along with the relative issues.

1.1 History, evolution and perspectives of Sensor Networks

At the beginning of Wireless Sensor Networks' civil diffusion, in 1999, a world-class journal as *Business Week*, classified this family of systems as one of the 21 technologies which would change the course of 21th century [3] [4].

WSNs are descendants of more general *Sensor Networks*, the Research on which has begun during *Cold War* for military purposes. By the time, computing machines were far to reach the current technological evolution and Internet (ARPANET, to be precise) had not already been developed. Notwithstanding the primitive nature of earlier sensor nodes, a lot of funds were alloted from U.S. Department Of Defense to enforce investigations on this kind of features, since many advantages were foreseen for controlling battlefields, monitoring air-marine-ground traffic, spying enemies' environments and improving accurate ballistic targeting.

The development of Sensor Networks' technology drew on three main areas: sensing, communication and computing research [3]. The very first example of Sensor Networks were the Sound Surveillance System (SOSUS), used during *Cold War* for detecting soviet submarines, and the U.S.A.-Canada air defense system, then AWACS (Airborne Warning And Control System). Clearly, these systems were quite incomparable in size, technology and application with modern Wireless Sensor Networks, but have surely represented the starting-point for all their theoretical and technical principles.

The actual research on sensor networks started at the end of '70s, prompted by U.S. Defense Advanced Research Project Agency (DARPA) and put in concrete by the Distributed Sensor Networks (DSN) program. In 1980 Arpanet had been operational for some years and R.Kahn (the co-inventor of TCP/IP) wanted to investigate the possibility of extending Arpanet's communication approach to sensor networks [3]. In 1978, during a workshop at Carnegie Mellon University [5], were defined the requirements for advancement in sensor networks research, holding true still today:

- the need for low-cost, small, spreadable nodes;
- the need for dynamic modifications;
- the need for heterogeneous node support;
- the need to integrate new software versions into a running system;

- the need for reliable, secure and power-efficient communication protocols;
- the need for eventual integration of Artificial Intelligence on networks' reconfigurations (considering the current DARPA's effort in enhancing AI).

As the previous points show, the problem of updating or loading new software on nodes at run-time has been pointed out as central one since the first sensor networks' symposium and this is exactly the target of our work and our dynamic linker in particular.

During the decade 1980-1990, the research was mainly concerned in developing military solutions (*network-centric warfare*) and, though the idea of small sensors was ever in mind of scientists and engineers,the technology was quite not ready. In that phase a key role was played by the great American universities, particularly Massachusetts Institute of Technology, University of California Berkeley and Carnegie-Mellon University, whose works had flowed into some important architectures: the U.S. Navy's Cooperative Engagement Capability (CEC), the Fixed Distributed System (FDS) and the Advanced Deployable Systems (ADS) for overseas warfare; the Remote Butterfield Sensor System (REMBASS) and the Tactical Remote Sensor System (TRSS) for ground warfare as Unattended Ground Sensors(UGS).

Finally, the 21st Century has revealed microelectromechanical systems (MEMS) [6], wireless networking and inexpensive low power processors getting a foothold. The conclusion of DARPA's Sensor Information Technology (SensIT) program have brought to a close important breakthroughs in networking techniques and data processing. Wireless technology has dramatically grown and new standards as the IEEE 802.11 and the sensor-oriented IEEE 802.15 protocols have been defined. Many advancements have also touched the research on microelectronics and sensing technologies, leading to the creation of WSN-development companies such as Ember, Crossbow, Sensoria and Dust Inc., many of them born as spin-offs from the University of California Berkeley, all along linked to U.S. advanced Military Research.

As scientists had still predicted in late '60s and more thoroughly in early '80s, nowadays is officially begun the era of Wireless Sensor Networks as the natural evolution of Sensor Networks in general. In next section we present an overview on Wireless Sensor Networks, exploiting the various facets related to their characteristics, potentialities and challenges.

1.2 An overview on Wireless Sensor Networks

In the space-age vision of a self-winding condition for pervasive computing, something of very real are now happening for what concerns the Research on widely-spreadable and easily-reprogrammable Wireless Sensor Networks.

These networks are composed by a certain number of small computing devices, called nodes, whose production is becoming more and more cheaper and whose battery-powered lifespan increases hand in hand with memory capacity and computing power. Those devices are commonly utilized in monitoring environmental phenomena [7] [8] [9] [10], patients' vital parameters [11], traffic conditions [12], production processes [13], plant and equipment [14], saying nothing of espionage purposes [15] and many others. All the retrieved information could be either locally processed or delivered to an Host base-station, through a wireless connection.

Each node is provided with a microprocessor, a radio-transmission unit, a small RAM, a primary FLASH, an external EPROM and some sensors relieving temperature, pressure, volume, images and everything else users are interested in. In dealing with WSNs' design, many criticalities must be taken into account, from the resource- constrained programming environment to the need for a robust routing protocol and an efficient reprogramming mechanism, all in the continuous commitment of reducing power-consumption to increase nodes' lifetime. A typical WSN structure is shown in Figure 1.1.

The nodes in a network are generally of two different types: sensors, which are used to sense for some external parameters, are typically smaller than routing nodes, which act as ties between the Host base-station (a PC in the figure) and the peripheral nodes. Routing nodes are commonly placed on top of networks' sub-trees and convey radio-traffic between theirs and others' sub-trees, as well

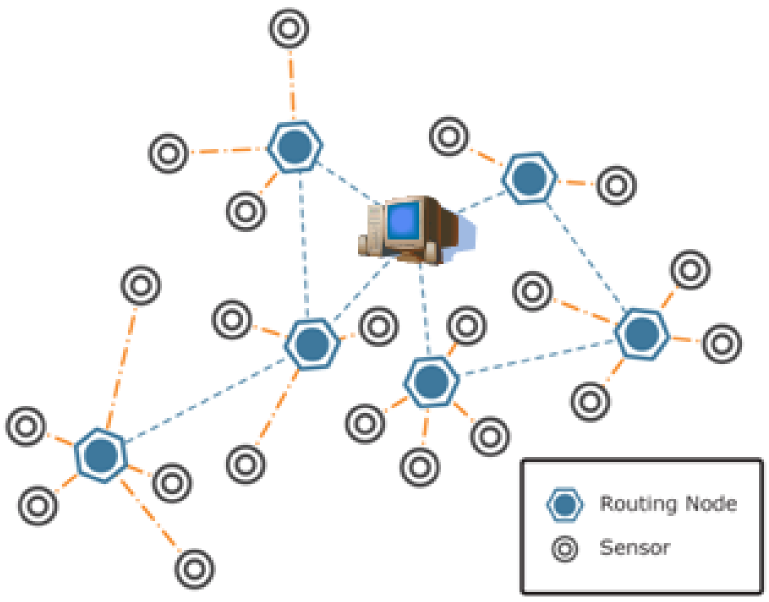


Figure 1.1: A typical WSN structure

as the information flux coming from the Host; they could alternatively be larger nodes with higher computing power and battery-life, out-and-out PCs placed in proximity of a certain set of nodes or small devices wired to a continuous energy source.

Sensor nodes, on the other hand, are small devices, whose battery-life must be maximized while ensuring the accomplishment of their tasks. They could be either of homogeneous or heterogeneous families, and either provided with homogeneous or heterogeneous operating systems and applications. These nodes are often limited in their ability of gathering and storing power, they're commonly scattered in tough environmental conditions and exposed to frequent position changes and critical degradations in network signal.

On this account, the whole network needs to be accurately managed, both for maximizing nodes' lifetime and to master possible changes in network topology or nodes' breakdowns or communication faults.

1.3 An outlook on Sensors' hardware

As mentioned in the previous section, in most cases sensors have an extreme resource-constrained hardware and this small asset of resource must be carefully handled in order to minimize power consumption. In despite of Moore's law, from their first appearance in 1998 to nowadays, sensor nodes (or motes, as they often are referred to) have not been increased in computing power and memory, because of the need for low-power microcontrollers to maximize autonomy. This stagnation in hardware advancement seems not to be inverting even today, and if some modifications are improved, they never change substantially the overall performance of these devices, with the exception of useful memory enhancements.

A typical architecture of a sensor (Figure 1.2) is composed by a Power source, a Microcontroller, an external memory and some sensors.

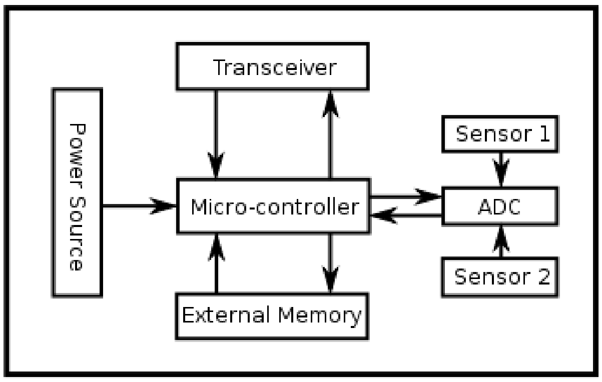


Figure 1.2: A typical mote architecture

The power source is generally a battery, which could be supported by some renewable sources like solar, temperature or vibration rechargers.

The transceiver could be based on Radio frequency, Laser communication or Infrared wireless transmissions: the first is preferred thanks to its suitability for different atmospheric conditions and relative positions; frequencies vary between the 433 MHz and 2,4 GHz and the functioning switches among four possible states (Transmit, Receive, Idle and Sleep) by means of an internal finite-state controller.

The band belongs to the ISM (industrial, scientific and medical) family, allow-

TelosB: The TelosB is a 2.4 GHz, IEEE/ZigBee 802.15.4, board used for low-power,wireless, sensor networks. It has USB programming capability, Chipcons CC2420 IEEE 802.15.4 standard-compliant radio transceiver for communication with integrated antenna, a low-power microcontroller TI-MSP430 from Texas Instruments with 8KB RAM, 32KB Flash (now expanded to 1 MB) and several onboard sensors.	
MICAz: The MICAz is a 2.4 GHz, IEEE/ZigBee 802.15.4, board used for low-power, wireless, sensor networks. It uses an Atmel ATmega1281 8-bit micro-controller with 8KB of RAM and 128KB of ROM along with Chipcons CC2420 IEEE 802.15.4 standard-compliant radio transceiver for communication. The maximum packet size supported by 802.15.4 is 128 bytes and the maximum raw data rate is 250Kbps. It also has several sensor boards providing light, temperature, audio, among other sensors.	
FireFly: The FireFly Sensor Networking Platform is a low-cost low-power hardware platform. In order to better support real-time applications, the system is built around maintaining global time synchronization. The main Firefly board uses an Atmel ATmega1281 8-bit micro-controller with 8KB of RAM and 128KB of ROM along with Chipcons CC2420 IEEE 802.15.4 standard-compliant radio transceiver for communication.	
IrisMote: AtMega1281 Processor,RAM 8K, Program Flash Mem 128K, Serial Flash 512K, Config EEPROM 4K,Radio RF230 2.4 GHz IEEE 802.15.4 output 3dBm with MMCX ant connector for higher gain antenna connection. UART, 10 bit built-in ADC, 51 pin connector	
Sun SPOT: 180 MHz 32 bit ARM920T core, 512K RAM, 4M Flash, 2.4 GHz IEEE 802.15.4 radio with integrated antenna, AT91 timer chip,USB interface	

Table 1.1: Specs of principal sensor nodes

ing free radio, huge spectrum allocation and global availability.

External memory consists of a small FLAH device, usually of just few KB and hardly ever greater than 1 MB , whose reads and writes turns out very energy-expensive for the motes.

There are many types of microcontrollers (the most diffused are: ATmega128, for larger nodes, and TI MSP430, for the smaller ones), showing 8-bit or at last 16-bit architectures and capable of self-idling to optimize work-cycles and power absorption.

Sensors could be of many different types, but classifiable in three macrocategories, the first of which is the most commonly used:

1. Passive, Omni Directional Sensors: self-powered (requiring energy only to amplify their analog signal) sensors used to retrieve data from the environment without modifying it.

2. Passive, narrow-beam sensors: differing from the previous since there is a precise direction in the measurement (e.g. camera).

3. Active Sensors: they probe environment in an active way, e.g. using a sonar or radar, which generate shock waves by small explosions.

Following we present a brief summary of the most common nodes being used today (Table 1.1).

1.4 The dynamic reprogramming Issue

The problem of reprogramming Wireless Sensors Networks is a very tough one, insofar as it is essential to enhance the network efficiency and scalability. The low-level features provided by the sensors' operating systems, justified by the necessity of minimizing power consumption during all node lifecycle, makes really hard the development of an effective mechanism to change or add functionalities in a dynamic way, i.e. without withdrawing the nodes from their position to reprogram them locally as well as without refreshing the entire status of a node. Many mechanisms have been proposed to overcome these limitation but the way to converge in a well standardized paradigm is far to be delineated.

In the following we analyze the critical success factor that a dynamic reprogramming mechanism should match, to be considered a good solution (Table 1.2).

1.5 Motivations

The introduction of a dynamic reprogramming mechanism provides many advantages in managing the complexities related to wireless sensor networks' tight con-

Critical Success Factor	**References**
the need for dynamic modifications, the need for heterogeneous node support, and the need to integrate new software versions into a running system	[5]
the need for the update to reach all the nodes, and to support fragmentation	[16]
the ability to update all the code on a node, and cope with packet loss	[17]
the need for distributed version control, bootloaders, and update builders and injectors	[18]
the ability to update the update mechanism itself, to reconfigure non-functional parameters (such as performance or dependability) without needing to update all the affected applications, and the need for utilities to provide users with standardized ways to manage online changes	[19]
the need for overall management of the software update process	[20]
unintrusiveness, low overhead, and resource awareness, and especially to minimize flash rewriting	[18]
minimize impact on sensornet lifetime, and limit the use of memory resources	[16]
minimize processing, limit communication to save energy and only interrupt the application for a short period while updating the code	[17]
possibly meeting real-time constraints	[21]
fit within the hardware constraints of different platforms	[22]
cope with asymmetric links and intermittent disconnections, have a low metadata overhead during stable periods, provide rapid propagation, and be scalable to very large, high density networks	[23]
security: integrity, authentication, privacy, and secure delegation, and intrusion detection	[24] and [25]
robustness: identifying and handling failure (or corruption) of the updated application, other applications, the operating system, and of the network , to tolerate hardware and software failures, to monitor system status, and to meet other dependability criteria (availability and reliability)	[26], [19] and [27]
ease of use and mantainance	[20]

Table 1.2: Critical success factors for an efficient and effective dynamic reprogramming mechanism

straints. As Table 1.2 shows, the need for dynamic reprogramming has been unanimously considered as a foremost priority in the evolution of WSN's technology and the literature has ever put a great emphasis on this topic since the very first workshop on wireless sensor networks [5].

Also the European Community, well understanding the potentialities of this new facilities, has endorsed a number of projects for defining and clarifying the guidelines for future research on WSNs. Among these, the *WASP* project [2], involving the most important European Universities and Research Centers, among which the Politecnico di Milano has played an important role. The first part of this work has placed under the aegis of *WASP project* and has targeted the definition of an experimental approach to dynamically reprogram wireless sensor networks, capable of running also in extremely resource-constrained nodes.

The good results obtained in this direction have pointed out that an energy-efficient dynamic linker/loader makes it possible to broaden the capabilities of a memory-constrained network in supporting a larger number of allocable functions, hence empowering the applicative usefulness of such networks.

Moreover, the superior degree of flexibility introduced by a dynamic management of a WSN, has opened the door for a better exploitation of the available energy, by searching for optimal configurations of statically and dynamically allocated functions, that maximize the lifetime of the whole network, while enforcing its capability to accomplish a given task.

It is cleat that if the overall available memory in the network would be sufficient, all the possible functions defining a given task could be statically allocated. However, this is not the case we are coping with, since the greatest part of actual wireless sensor networks present an overall available memory undersized respect to the need of functional allocations.

To guarantee the fulfillment of all the operations related to these functions, in presence of memory constraints, there is just one possible solution: to define an hybrid configuration of both statically and dynamically allocated functions, the latter executing in turn-over. A dynamically allocated function, in fact, can be downloaded from a central host and executed on node for a certain period of time, to be then unloaded and replaced by another dynamic function and so on. From this comes the need for a formal and rigorous approach to optimize the network configuration, in terms of statically and dynamically allocated functions, in order to reduce as much as possible the energy consumption and maximize the lifetime. The life span of a WSN, in fact, has often been required to last for months or years, so the need for functional completeness must be shaped on the respect for lifetime requirements.

Considering these aspects and getting back to our work, the first step has consisted of creating a mechanism for dynamic reprogramming that would be energy efficient enough to justify its introduction in extremely resource-constrained WSNs. Many proposals, in fact, have been advanced in this direction, but most of them are too expensive in terms of radio transmissions and/or processor effort. Many others suffer from harsh functional limitations, tough presenting a good level of energy-efficiency. Conjugating these two aspects has been a very hard challenge, but through a deep analysis of the literature and a long theoretical investigation, we have found a way to exploit the advantages of a power-efficient mechanism based on pre-linked modules, with the dynamic capabilities offered by another technique, the dynamic linker, which has always been considered too expensive under the energy-consumption profile.

The very low energy-costs required by our lynker/loader, have smoothed the way for the successive step, concerning the definition of a model to optimize the allocation of static and dynamic functions, targeting the lifetime maximization and guaranteeing the execution for all the functions of a certain task.

From the work on the dynamic linker/loader we have inherited the possibility of precisely estimating the costs, in terms of energy consumption, to dynamically load a given function, as well as the costs for forwarding it through the routing path. Using these information, after a rigorous domain definition, we have formalized a mathematical model that, under both resource and functional constraints, targets the optimal configuration of the network in terms of statically and dynamically allocated functions. This model has then been mapped on a genetic algorithm, a choice that has been proven as an outstanding solution for modeling evolutionary behaviors in many fields including WSN and other dynamic interdependent systems, such as neural networks.

The choice of a genetic algorithm has also been motivated by the future perspective of integrating the algorithm's logic directly into the network, to enable eventual context-sensitive reconfiguration in next-generations WSNs, which will be object of future works.

The genetic algorithm has been implemented in C-Languages, by means of the GAUL [28] libraries and some experiments have been performed to test its effectiveness and reliability, obtaining very promising results.

A future ambition is then to integrate the dynamic reprogramming mechanism, the genetic model and the non-functional manager [29] to enable a fully distributed management of a wireless sensor network, in distributing as much "'intelligence"' as possible among the nodes in order to reduce the need for human control.

Chapter 2

The state of art in WSN dynamic reprogramming

In a widely distributed Wireless Sensor Network (WSN), the opportunity of dynamically reprogramming sensor nodes holds a primary role in assuring efficiency, scalability and effectiveness to whole system management. Many solutions to this topic have been proposed both by academic and industrial community, each one adapting better or worse to the specific context, depending on network dimension and resource capability of a specific sensor node.

The strong performance requirements of a WSN environment have triggered many inquiries about possible implementations. At the moment, the main solutions proposed by the International Research are: Full Image Replacement, Differential Updates, Run-time Interpreters and Script-based Approaches, Virtual Machines, Distributed Databases for WSN, Service networks, Agent-based approaches, Pre-linked loadable modules and Dynamic Linking.

Following we analyze each technique, concentrating on the leading references and releases in current literature. We expect that studying the strengths and weakness of each technique could help us to distil some useful information about the correct way of facing up many dynamics underneath the behavior of a distributed set of nodes.

2.1 Image replacement

It consists in compiling (at host-level) a new image of the operating system together with the new functionalities and then replacing it entirely into node. The only advantage of this approach lays on the non-necessity of node-level computation: a full image of the system can be loaded at the same, known, physical address. The disadvantages, on the other hand, are many and heavy: this technique is not scalable and hardly applicable to network with a large number of nodes as well as to networks with heterogeneous types of sensors. Moreover, it's very expensive in terms of power consumption, since a massive bulk of code (referring to the mean dimension of a node memory) must be loaded into the sensor, wasting a lot of power in radio transmissions.

2.1.1 XNP,MOAP and MNP for TinyOS

TinyOS is one of the earlier and certainly the most famous and diffused among the various operating system for WSN. The first mechanism proposed for reprogramming a TinyOS-based nodes is a direct full image replacement. This mechanism is called XNP [30], and it consists of an unicast protocol involving the host computer, from which the network is managed and the various nodes.

The protocol is implemented by means of a boot-loader, embedded in the operating system, and some XNP's protocol information that are provided along with the modules to be loaded.

In receiving the new image, a node suspends all its running tasks and stores the image into the external flash. Once download is completed, the host remotely invokes the boot-loader, which installs the new program in the main memory, reboots the node and sets the program counter with the first instruction of the new application.

When a new application needs to be updated, the host sends the new image, opportunely divided into chucks of the same dimension of a network packet, to a specific node or to a set of nodes. In spreading this information, the host uses a root node, connected through a serial port to the PC, provided with the TOS-

Base execution environment. This root-node node is charged with forwarding the packets to the target nodes via directed radio transmissions in two different ways:

- if the packets are broadcasted, the protocol articulates in two phases: in the first phase, the image is sent (in chunks of a network packet's byte dimension) to all nodes, carelessly as regards the correct receiving of each message. In the second phase, a query is sent by the root to all nodes, requesting a list of the missing packets: each node completes its list and sends it back to the host, which will provide for sending these packets again.

- If the packets are directly sent to a specific node, the root requests an acknowledgment message for each packet, re-sending the message if a lost occurs.

In trying to overcome the XNP's lacks of scalability and efficiency, a TinyOS multi-hop protocol has been developed and successfully ported to Mica2 architectures: its name is MOAP [16], acronym of Multi-hop Over the Air Protocol.

This protocol is very articulated and resource-expensive, the reason that why it only applies to large nodes, such the Mica2 ones. The logic surrounding this paradigm is to spread a new image through the network without the cost of a unicast transmission or the constraining broadcast alternative. Each node in this architecture acts as bridge for its siblings and takes an actual part in disseminating the new image, storing it in the local EPROM and forwarding to remaining nodes. In that way a complete reprogramming hierarchy could be established in the network and the code distribution becomes more efficient. To create this hierarchy, an election protocol has been defined by means of a subscribing-message exchanging between child-nodes and the nearest sub-parent, that has already received a new image. When a node doesn't receive any more subscribing message, it assumes to be located at a leaf-level and starts loading the new image in the main memory and re-boot the system. So, the reprogramming process floods from leaves to root, little by little no more subscribing message are received. The host operates as a publisher, sending a new image and marking it with a version number used to identify the reprogramming status of the network: only newer versions are downloaded by nodes, preventing eventual duplicates. To avoid dangerous losses in message transmission, each node maintains a sliding window of the received packets and continuously check for the sequence number of each one. If a packet misses, then a re-sending request is transmitted to the parent and a timeout is set, after which the request is newly sent. To make more power-efficient this mechanisms, acknowledgment messages are buffered in a stack and a threshold is defined to specify the message loosing tolerance. Another tiresome problem, that often afflicts distributed systems is the network partition: MOAP manage this factor by dint of frequent samplings over the radio transmission quality. All the samplings are then organized in opportune statistics, used to control the transfer rate and to enforce a linear routing tree and avoiding two or more nodes to become senders for the same set of nodes.

Another mechanism falling back in this category is the MNP protocol [24], which could be considered a super-structure of TinyOS's XNP, enhancing a routing-tree mechanism for distributed code diffusion. For this, it may be considered a low-level variant of MOAP reprogramming system and, as MOAP does, also MNP takes the advantage of a publisher-subscriber mechanism. Once a new image is created by the host, it is published so that interested node can subscribe and download it. The routing tree evolves in an efficient way, since each node sense for subscribing requests of the other participants and univocally forward the new image to nodes that have already subscribed to it and have not registered to other sources. This entails the presence of a single forwarding node for each sub-tree in the network.

Once the set of recipient-nodes has been defined, the parent sends a message of prepare and starts forwarding the image in chunks of byte, according to the network-packet dimension. In addition to the MOAP approach, MNP allows a transmission rate control, based on the FLAH write-speed of the children. The recipient, on the other side, creates a sliding window of the received messages, store them into the external flash and checks their sequence numbers to finds out eventual losses. All the missing sequence-numbers are collected in a list, stored in turn into the flash, and sent back to the parent in response to a recover message, that it spreads to the children to control download status. This recovery process is reiterated until all the packets have been correctly received from each child-node.

After the download completion, a child-node has in its flash the entire new image and can start forwarding it to its children in turn. At the end, when the image reaches a leaf node (which doesn't have any subscribing request, since it hasn't children) the image is copied from EPROM to main memory, and the usual reboot is performed.

2.1.2 Deluge

From the previous approaches a critical drawback emerges: there is no support for differential image updating in case of small changes. A tough replacement of the system image, in fact, turns out useful when a substantial modification is performed on the current configuration, but is very expensive for relatively small changes.

Deluge [31] [32] is a protocol expressly studied to propagate incremental system upgrading, especially effective in managing the transition between an old and a new version of the same image. It can be classified, however, as an image replacement mechanism, since no effort is provided for heterogeneous platforms as well as for single-function disseminations: the same image is spread throughout the network. No mechanisms are provided to avoid message loosing and does not exist an algorithm (but just an heuristic) to choose senders, so multiple senders are possible, causing useless and expensive radio-transmissions.

The most evolved versions of Deluge (often experimental prototypes) implement some advanced features for what concerns multiple version support, online updating (without tasks suspension), power-consumption control and efficient message exchanging.

As the name Deluge suggests, the updates are performed through a rain of short packets, each containing part of the system image, which is opportunely partitioned in pages of the same dimension of a RAM block. This allows a direct RAM buffering and avoids expensive EPROM storages. Moreover, incremental updates are propagated through special advertisement messages (protocol-messages), specifying which pages differ from the previous version.

A bit-map representation of the whole image is maintained by each node, so that a bit is activated only if the packet relative to the corresponding part of a page has already been received. According to a specific updating rate, every node spreads some protocol-messages, containing the bit-map representation of each page and the relative version number. Listening to these protocol-messages, a recipient node controls its last page and checks if the current versions are older than the received ones, eventually requiring the packets with newer versions. Once these packets have been received, the node broadcasts in turn a protocol-message, containing the bit vector and the version numbers. At the same time, it repeats the first phase for its next-to-last page, requiring eventual upgrades, and so on, until all the pages have been updated.

In that way, the protocol evolves in a pipelined flux, that enables a powerful spatial multiplexing, made more efficient by keeping constant the dimension of the system data.

2.1.3 Others

All the monolithic environments , such MantisOS [33], NutOS [34], etc. support mechanisms for full-image replacement. These variants are based on unicast/broadcast transmissions and a boot-loader utility; their functioning is very similar to that of XNP.

2.2 Differential updates

When an update on the applicative configuration of a node causes small changes between the old and the new version, just the binary differences between the two versions needs to be installed. Updating just the delta-part of binary code, the power consumption of data transmission is drastically reduced, especially thanks to the small amount of code exchanged between a node and the host computer. This mechanism however suffers from many problems of scalability and effectiveness: no broadcast updates are possible and the modifications must be propagated separately for each node. In that way, the host computer has to know the exact configuration of each one and has to maintain this information aligned. Furthermore, the host by which the network is managed must cope with the micro-heterogeneity surrounding even nodes of the same family whit same applications. This heterogeneity is due to the small differences in C compiler (also the same compiler in its different releases), libraries and linkers used by the different programmers who concurred to the network instantiation.

The differential updates method has not been successfully implemented in any real system since it bad performs in networks whit a non-minimal number of nodes: only academic proposals have been issued by Koshy-Pandey and Reijers-Langendoen.

2.2.1 Model by Koshy- Pandey

This is a function-based approach, underlining the central role played by the modular structure of programs obtained composing a certain set of functions [18]. It is expressly designed for designed for Mica2 platforms.

The reprogramming mechanism is enabled both for static and dynamic updates and a linker is placed on the host station in order to relieve the sensors of the overhead related to linking, relocating and compressing operations. the remote linking process, however, must maintain a map of the address space, per each node. The process, as it has been proposed, comes out quite inefficient since to replace a function with its newer version (without changes the references to this function in the program), the latter must keep the same byte-dimension of its previous version, otherwise it, or its adjacent one, must be moved.

Using a special algorithm, called Xdelta, some scripts, each of the dimension of a memory page, are produced to propagate the new version in an incremental manner. In that way, each page could be singularly updated and, exploiting the ATMega128's NVRAM capabilities, a new script can be downloaded at the same time. The updates are installed by means of a boot-loader, which interprets the script and transfers in memory the relative modifications. This mechanism shows good performances for what pertains memory usage, but suffers the bottleneck of being suitable just for small updates.

2.2.2 The proposal of Reijers and Langendoen

This is another diff script technique, but differs from the previous because here the scripts are used to progressively propagate a reduced version of the system image [17]. There is not pipelining in packet distribution, so the protocol collapses in a store-and-forward disseminating paradigm.

The script is incrementally collected into the external flash, little by little the packets arrive and until their aggregation matches the just mentioned reduced image. Unfortunately, this script language is designed to only run on the MSP430F149 architecture of EYES nodes.

The main advantage of this proposal is its compatibility with run time operations, since the node does not need to suspend the current task while downloading the script. Another advantage lies on the packet structure, which contains, along with the script, also the final address range that it will occupy on the node memory, making possible an immediate fetch from the EPROM to main memory by means of a boot-loader. This also enables out of order deliveries and an immediate control on eventual lost packets, since it only suffices to control the address range of two packets to check their continuity.

2.2.3 Others

Many other diff based proposal have been advanced, often in the context of academic researchers. The Jeong and Culler's model [35] is an interesting one and substantially consists of running a resynchronization algorithm to extrapolate the difference between the old and the new version.

2.3 Run-time Interpreters and Script-based Approaches

This reprogramming paradigm falls within the broader family of script languages dissemination. Since a generic interpreter is quite heavy, in terms of execution costs and memory usage, compared to the strict requirements of a small architecture, this method doesn't fit for the greatest part of wireless sensor networks.

The logic beneath a script reprogramming consists in spreading a short message, containing the script to be executed, toward a specific set of nodes (to the limit: toward a single node or the entire network). The main advantage resides in the minimization of radio-transmission costs, considering the lightweight dimension of a script message. On the other hand, the drawback concerning the high execution cost of an interpreter makes this way of reprogramming prohibitive for small sensors.

2.3.1 Python interpreters

The idea of using an interpreted language as Python is quite a black swan for projects orbiting the WSN reprogramming topic. Starting from an analysis upon the advantages correlated to the open-source and re-sold nature of this language, two researchers from the Turku Center of Computer Science, J.Lilius and P.Paltor [36], have proposed a stripped version of Python fitted to run over resource constrained embedded systems.

The main advantage of this approach is the relatively small footprint of the resulting programs, which whit an average weight off-peaking the 200 KB could be adaptable to nodes of higher sizes (e.g. the HitachiSH1 with 256KB RAM and 64KB ROM). Another strong point, this time relative to the dynamic reprogramming perspective, refers to the well-known possibility to replace Python's object and method on-the-fly, thanks to the interpreted nature and the dynamic binding surrounding this language.

Unfortunately there are also a lot of disadvantages, mainly related to the heavy processing-overhead and resource requirements that a Python interpreter needs for its execution. This makes the use of a Python interpreter for WSN just an interesting academic case of study, but a prohibitive choice for actual implementations, considering the current sensors' technologies.

DePython is expressed developed for systems in which the interaction with user is critical, so it is not oriented to autonomous WSNs, which are often used to monitor some environmental conditions of impervious places. The DePython interpreter has been stripped from all the direct dependencies whit the operating system and uses special wrappers to interface with hardware: this approach, as said, is a very expensive one and seems to be promising just for powerful WSNs remotely manageable through an internet interface.

2.3.2 SensorWare (TCL machines)

SensorWare [37] is based on TCL, i.e. Tool Command Language, an interpreted language used to test and run prototypal applications. Also in this case the node must be equipped with a run-time interpreter which, as discussed in the previous section, requires a lot of resources. The protocol underneath this approach is event-driven, that involves a massive message exchanging among nodes during all the network evolution. The main advantage of SensorWare, besides the fast dynamical reprogramming, is the simple and powerful nature of TCL language. To the contrary, because of its heavy power-consumption and memory usage, it only applies to large nodes. The code diffusion is obtained through an in-node script replication: once received the script, if it contains the special command *replicate*, the node duplicates and sends it to any set of requiring nodes.

2.3.3 Others

Two other interesting approaches using scriping languages are COMiS [38] and Spatial Programming (SP) [39]. COMiS is a special middleware conceived to be modular with respect to script-based components, written in DCL (Distributed Compositional Language). It shows the same pros and cons of the previously discussed processes. SP takes advantage of active script dissemination through network messages, it however doesn't provide a flexible way to manage the address binding and turns out useful only for application updating rather than an out-and-out reprogramming.

2.4 Virtual Machines

Before introducing the use of virtual machines in WSN, we ought to briefly discuss some properties of the bytecode, in relation to the need of low-power consumption for radio transmissions. The program code of a VM can be generally made more compact than the classical binary of a native machine, so the burden of messages in data transmissions can be drastically reduced enabling a great power-saving as opposed to classical binary dissemination. Many virtual machines have been proposed for WSN specific environments, attempting to preserve the privileges of running an intermediate code (identical for all platforms), and, at the same time, to make the dimension of virtual engines suitable for the resources of a node. Virtual machines have not significantly penetrated the spectrum of WSN implementations because of their run-time overhead: in spite of the various optimizations, in fact, the execution costs of these mechanisms comes out expensive for almost the totality of sensor platforms. In such a contest, VMs have always been developed as a support for the node's OS, obtaining an hybrid configuration in which native applications interact with programs running on virtual machines. The double-bond that links VM and OS in sensor networks substantially differentiates these variants from the classical Java Virtual Machines: a JVM performs well on all possible kinds of applications, while a WSN-oriented VM is often developed to run on a specific operating system and to accomplish just a fistful of tasks.

2.4.1 SwissQM

Among the various virtual machines proposed for WSN, we now present one of the most recent (publication year: 2007) and surely the most interesting under the power-efficiency and instruction-set completeness profile: SwissQM by ETH Zurich [40] [41]. Thanks to its small footprint, 33KB of Flash and 3KB of SRAM, and its wide but small-sized instruction set, 59 bytecode instructions needing just ten of the 33KB of Flash footprint, SwissQM places at the top of the existing VM-based mechanisms. It offers a platform-independent programming abstraction by means of a stack-based, integer virtual machine. The structure comprises a bytecode interpreter, an operand stack and a transmission buffer. Applications running on SwissQM can reserve space through a data structure called synopsis. It supports up to six task running concurrently, because of resource limitation, but the theoretical number of parallel running tasks is unbounded. Over 59 instructions, 22 are reserved to control the physical sensors, while the remaining 37 are identical to JVM implementation. QM programs are split into fragment messages, identified by an ID and a sequence number. In downloading a new application, a service message, containing the application's metadata, is exchanged first. This message contains a 10bytes header (metadata), whereas the remaining 16bytes are dedicated to bytecode; the following messages deliver the remainder bytecode, reserving just two byte as header (ID and sequence number), while the remaining 24 bytes are used as payload.

A typical reprogramming sequence starts with a node that requests a fragment sending the first message to another node. If the receiver holds the specific program, then it generates the requested fragment and begins to send it. An ack message is sent every time a fragment delivery succeeds; an epoch-based recovery mechanism is provided to overcome eventual message losses.

The main advantage of this solution is its small footprint , the most compact and efficient bytecode ever proposed and a Touring-complete programming environment. It could be considered one of the most attractive mechanisms today available for dynamic reprogramming, so no particular flaws emerges with the exception of a quite expensive processing cost due to the programs interpretation, probably surmountable as sensors' hardware evolves. It must be noticed that this approach doesn't fit well for small nodes, e.g. TelosB, since the 33KB footprint alone would saturate the entire memory space.

2.4.2 Maté

Maté is a virtual machine developed to run on TinyOS [42]. It provides an high-level programming interface, which makes programming very easily and, at the same time, reduces the applications' sizes in a range of 100B. The code distribution is performed through a viral diffusion of small packets, each containing 24 instructions and capable of self-replicating: only bytecode programming is supported in the proposed version. Moreover, it is possible to spread a maximum of 4 subroutine capsules to speed up the diffusion of high-sized applications.

Version numbers are used to keep tracks of the various updates and nodes communicate each other these version numbers to decide if forwarding or not: if the neighbors' versions are older than the local ones, then the packets are forwarded, nothing otherwise. The communication among nodes is performed through a thick message exchanging and the transmissions are interleaved by means of a random timer. The strong point of this approach is the great reprogramming power that a viral approach enables, also for wide and scattered sensor networks. The main drawbacks concern the heavyweight processing time needed to execute the virtual machine, the frequent, often useless, radio-communications and the absence of an effective mechanism to check if a reprogramming has successfully taken place in all nodes. Mat is particularly suitable to wide sensor networks with thousands of nodes.

2.4.3 Others

Many other WSN-oriented virtual machines actual exist, as well as specific applications mounted on top of them for the dynamic reprogramming purpose. Trickle [23] is a famous code-propagation facility, running on Mat and enforcing the diffusion of an entire application in the byte-space of just a TinyOS' packet. A gossiping cadence along with a traffic regulation algorithm make this mechanism very scalable and efficient also for widely diffused systems.

CVM [43] is a virtual machine expressed developed for ContikiOS and running in symbiosis with Contiki's dynamic linker. It is a stack based VM, with separated code and data spaces, enabling remote calls to native functions by means of a special handler instruction.

MagnetOS [44] holds a full-blown JVM , powerful but heavy-sized, therefore unsuitable for medium or small nodes. VM* [45] is a small platform, enforcing Java programming while preserving low-level interactions by means of a direct access framework for sensors and I/O devices.

2.5 Distributed Databases for WSN

In dealing with network reprogramming we often mean to change on-the-fly the applicative configuration of a sensor node, e.g. if a node has been programmed to sense for temperature and report results we might then need the same node to report pressure, so a new function must be loaded and the previous discharged. A Database approach makes the problem of reprogramming fully dynamical, since it suffices to spread a query to report, for example, the temperature and then another query to report the pressure. This process is very efficient and, considering the small byte dimension of a query (that is just a string), also advantageous under the radio and processing profile.

The application of Database's logic to a distributed system of nodes represents a revolutionary and remarkably example of engineering, both for the extreme fragmentation of data source and the weak computing power of the nodes on which the distributed DBMS has to run. TinyDB is the main and most evolved example of DBMS for sensor networks and under many aspects could be consider a dare and brilliant research, for which we foresee interesting possibilities of integration in enterprise's production monitoring and data collection. For this reason, we propose a deep and accurate analysis of TinyDB below. In this platform, each node is seen as a single source of information, which produces one and only one tuple per each sample period. All the tuples produced by the various nodes flow up the network hierarchy in a continuous streaming of results, collected by the root. The streaming nature of the output makes even more difficult the coordination between participants, and represents a very hard problem also for the specialists in conventional DBMS so much so no a standard has already been reached by international DB community. TinyDB copes this tricky matter in a very efficient way and places itself among the most futuristic works in the field.

Nevertheless, this project has not reached an appropriate diffusion because of the significant power absorption needed to maintain the semantic hierarchy (the equivalent of an index for conventional DB) and others coordination mechanisms, which require a thick message exchanging. Another limitation pertains the absence of a functionality to interact with a single node, since the only way to execute queries is t broadcast them.

On the other hand, since the query are optimized at host level and sent in the form of a small binary message, the single radio interaction turns out very cheap under the costs profile. Anyway, considered the high technological profile of these WSN-oriented databases, we expect them to come back to the fore once the hardware profile of nodes will become more power-efficient.A more complete tool-suite for TinyDB is provided by TASK [46], which can be considered just an extension that doesn't modify the logic underneath.

2.5.1 TinyDB: an analysis through the filter of DB Theory

TinyDB [47] is one-of-its-kind project, attempting to deploy a fully featured DBMS applicative logic on a resource-constrained distributed environment, like a wireless sensor network. Though Database systems, because of their overshadowing complexity, are often associated with high powerful multicore machines, this experiment demonstrates that a sharp review of Database Theory enables solutions capable of running even on small nodes (10KB of RAM and 30KB of FLAH memory).

In this section the theoretical structure surrounding TinyDB is analyzed, referencing to the Fundamentals of Database Theory [48]. In drawing a parallel between the general theory and the specific implementation, many characteristics are sequentially compared: distributed database and streaming database mechanisms, query language, query dissemination and execution, active rules (trigger), query optimization.

The problem concerning transactional support will be discussed apart, since it seems to be a lack of TinyDB's deployment and literature.

Data distribution and streaming in TinyDB

The relational schema of TinyDB is composed of a single streaming table, called sensors: each row represents a node in a given instant of time, whereas each colon stands for an attribute produced by a device. In a theoretical perspective, this could be considered as an horizontal fragmentation taken to its most extreme, where each tuple lies on a distinct node.

Transparency is set at fragmentation level, relieving programmers of low-level details. Since the schema is unique, each node is allowed to put a NULL value every time an attribute is not defined for its platform.

The stream-oriented nature of TinyDB entails the records to be taken for a short while: to manipulate or join tuples locally, passing through materialization points is needed. Once a query is launched and disseminated throughout the routing tree, each node starts producing tuples at a sample rate specified by means of a SAMPLE PERIOD query command. The command just introduced follows the select-from-where clauses and has the syntax:

```
SAMPLE PERIOD <time>[FOR <time>]
```

The period of time between two successive samples is called epoch and provides a good mechanism to organize computation by minimizing power-consumption: nodes synchronize on a global time in order to start and end each epoch at the same time.

The output of a query consists in a stream of tuples that flows up through the routing tree, clustered into time intervals. Each tuple is characterized by a timestamp relative to the moment it has been produced. Because of the continuous data stream of tuples, blocking operations such sort or symmetric join are not allowed, unless a bounded subset of the stream, called window, is specified. These windows actually consist of small buffers of data, collected in materialization points over the stream. The syntax to create a materialization point is:

```
CREATE STORAGE POINT name SIZE sz
AS (SELECT )
```

It is possible to execute a join between two storage points, as well as between a storage point and the sensors table, which is used like an outer relation in a nested-loop join: when a tuple arrives in the outer table, it is joined to those of the storage point .

Query language

As the previous section beckoned to, TinyDB adopts a SQL-like query language. Along with the usual select-from-where framework, system designers have broadened the syntax with some interesting solutions, specifically oriented to dynamic acquisition of environmental data. Lifetime queries, actuation commands, temporal and stream-oriented aggregations constitute the nucleus of this extended semantics.

The LIFETIME clause reports results automatically during all node life span. The clause ”OUTPUT ACTION action” serves to take some physical actions on external devices and, in the opinion of the author, leads to critical problems of transactional support.

Temporal aggregates, like WINAVG, help users to deal with sliding-window queries, often required in monitoring dynamic conditions. Stream-oriented aggregates are following discussed.

Stream-oriented aggregates

These aggregates enable in-network aggregation as streaming results flow up. This dynamic bunching process evolves according to the aggregation function and the value-based partitioning specified in query commands.

Each aggregate record consists of a couple $< group\_id, aggregate\_value >$, time-stamped with relative epoch number, such that only results belonging to the same epoch will be aggregated. An interesting parallel with classical Database Theory emerges in drawing streaming aggregates close to those of shared-nothing databases: final output comes out from reiterate applications of three functions, respectively called merging function (f), initializer (i) and evaluator (e). The initializer i is used to instantiate the primitive multi-valued partial-states on each node. Merging function f takes two parameters, which are multi-valued partial-states, and aggregates them. The evaluator e takes a merging function output and calculates the final value of the aggregate. After the initializer has instantiated all nodes, the stream climbs the hierarchy and a sequence of merging function is applied until the evaluator calculates the final value at the root.

Active rules

In conventional commercial databases, active rules play a first class role in guaranteeing the correctness of integrity constraints or ”company rules”, in calculating correlation on data and in handling exceptions. The theoretical paradigm underneath triggers is the so-called event-condition-action, which well underlines the reactive and transparent nature of these queries [16]. In dealing with active rules it is common to think about something that acts independently from users' control: that's because transactions are directly invoked by the human operator, while triggers automatically start on the sly to check some properties. Another important characteristic of active rules concerns a new level of abstraction, called applicative independence (or knowledge independence), which tags along physical and logical independences, basically provided by Databases' engines.

In TinyDB the event-condition-action paradigm increases in usefulness, since power consumption can be drastically reduced in waking nodes only if necessary. On the other hand, triggers become a full-blown programming style and loose their background-control nature to surge in a new dimension of ad-hoc deigned queries. In such a context, the three level of independence above introduced vanish:

- The role played in preserving applicative logic no longer makes sense, as well as knowledge independence;
- Events are detected by physical devices: this means a programmer must know the specific low-level functions to be invoked. The physical independence offered by conventional databases for what concern data organization and language fails too;

- Working in a streaming environment implies a consciousness of how data flows, in order to manipulate them through materialization points: even logical independence collapses.

Let's have a look on how Tiny triggers work :

```
ON EVENT sense_for(x) :
SELECT
```

The function sense_for(x) is a formal parameter and represents a generic low-level function supported by the operating system. Such a query reports its result every time the event x occurs, given a certain sample period. In addiction to reactive behaviors, TinyDB offers the possibility to signal the occurrence of some conditions through the "OUTPUT ACTION SIGNAL" clause: this active framework turns useful when programmers want to fire queries at a certain event occurrence.

TinyDB lacks of an event-propagation distributed protocol and the trigger scope is confined to a single-node dimension. Concluding, it has to be remarked the absence of a distributed algorithm which checks for termination conditions if two or more events activate each other in an endless loop.

Query dissemination

Queries in TinyDB are broadcasted from the root to the whole network, without any control on which nodes actually satisfy the requirements. Power consumption is a critical parameter to be preserved, so all nodes in hierarchy must synchronize their wake and sleeping cycles to minimize processing time. To orchestrate network synchronism, each node is forced to transmit only when the parent is awake and listening. The route rate leads the entire network cadence and the rhythm is propagated from parent to children by means of a transfer rate parameter, forwarded along with queries. If a node needs to transmit slower, it can transmit an integral divisor of the root rate, if it must transmit faster a MIN SAMPLE RATE clause could be specified, forcing the propagation of results without any controls over energetic autonomy.

Since forwarding queries has a great cost in wasting power, a parent node should decide if the query applies to its characteristics as well as to those of its children. For what concerns constant attributes (i.e. node ids), it seems to be an affordable problem, while in dealing with range condition, it would be useful knowing if children overlap the boundaries.

TinyDB offers a *semantic routing tree (SRT)* in which internal nodes collect their children according to an attribute-range criterion. In this way, only queries whose predicates specify a coherent attribute value will be considered by a certain subtree. To create the SRT, a build request is spread throughout the network, indicating the attribute over which the network should be built. Internal nodes and leafs then organize each other by a message-exchanging sequence, until the whole network structure is created and each parent records the attribute range characterizing its subtree. A run-time algorithm then arranges eventual topology changes between parents and sons.

Query execution

The focus here concentrates on aggregate queries, since simple queries are executed conventionally. To schedule aggregate queries, each epoch is divided into fixed temporal slots and each slot is numbered in a decreasing order. The slot with highest number (early in time) is assigned to the last node in hierarchy and so on until the parent receives the lowest numbered slot (last in time). Each node activates and sends results in its timed-slot, so the stream floods sequentially ordered toward the root. For what concerns multiple queries there is no effort.

TinyDB also offers a prioritizing system: if the transmission rate is faster than the tuple arrivals rate no problem occurs, otherwise a policy to manage overflows is required. Three methods are supported:

- Nave: all tuples hold the same priority and the queue is managed as a classical FIFO, where the overflows are discarded.

- Winavg: similar to nave but the first two tuples in queue are aggregated, making an average of them and freeing space for another result.

- Delta: a timestamp-based priority is assigned to each tuple, enabling out-of-order deliveries. Tuples with lower priority are discarded when an overflow occurs.

For what concerns aggregate queries, two cases must be distinguished:

- Exemplary aggregates (MAX, MIN, etc): each node senses for its siblings results and if any has produced a value greater (smaller) than the local one, the local tuple is discarded, avoiding useless transmissions.

- Summary aggregates (AVERAGE, SUM, etc.): each node senses for its sibling values and if the local result matches on average whose of the siblings, the local tuple is discarded. The parent node, in not receiving messages from some of its children, assumes that their values are in line to those actually received.

Query optimization

In conventional DBMS, optimizations are mainly performed inside the Query-Manager module. Query-Manager's optimizer receives SQL-written queries and computes the algebraic transformations among their selections and projections. Then it optimizes the intermediate output depending on the access methods and finally generates object code. All of these operations can be executed in a centralized way, through a master-slave paradigm, or can be delivered to the peripheral nodes, in a distributed negotiation paradigm.

TinyDB adopts a master-slave paradigm, so optimizations are executed at root level and the output is a simple binary executable. The root must hold information about each node in the network to globally optimize queries. These information are provided by some metadata maintained by each node and periodically copied to the root. An interface file and an handler are provided for local use. As commercial databases do, also TinyDB allows both compile-and-store and compile-and-go procedures, depending on query frequencies. Optimizations dealing with scan, sort and direct-accesses operations are part of TinyDB as well, but in a power-saving perspective.

In conventional DBMS joins are frequent and heavy, while in TinyDB they are rare and just allowed on small-windowed materialization points. Tiny's optimizer doesn't care much about joins, except for those between a materialization point and the whole sensors table, handled through a nested-loop approach. Neither merge-scan nor hash-based mechanisms are applied.

Cost-based optimization in TinyDB is founded on a power-consumption logic: cost function is dominated by sampling sensors and transmitting results, so the main challenge is to optimize these phases.

- Samples and predicates: samplings on physical devices are scheduled at first. In that way, just one sample is performed at the source for each attribute eventually required by more than one predicate. Moreover, the physical interrogations are scheduled on the basis of the relative power consumption. In doing that, if a predicate on a less expensive physical acquisition is violated, the query is discarded together with the eventual subsequent acquisitions needed to evaluate the remaining predicates.

- Exemplary aggregate pushdown: it is more convenient to previously check if local values actually affect the aggregate. If they don't, sapling for the predicate is superfluous since the query is useless.

- Event query batching: TinyDB's optimizer creates an array of the events and treats results as joins between this array and the stream of samplings. In that way, old events can be dropped by the event-array avoiding further acquisitions.

Lack of transactional support

TinyDB de facto doesn't support transactional behaviors, though its streaming nature imposes all tuples to be marked with a timestamp. For all queries accomplishing read-only operation the problem doesn't subsist, since each tuple could be filtered by its timestamp. When a physical action (OUTPUT ACTON) is taken on a device by two or more nodes, however, some conflicts may occur. A possible solution could be introducing an action-timestamp-manager (ATM), working as the

WTM counter in traditional timestamp-based DBMS. In linking an ATM to each physically-controlled device, the system can keep memory of the actions taken on that device, preventing an eventual conflict. Anyhow, this approach should be fitted to support distributed decisions.

TinyDB offers a good level of serialization for reading operations, which in a sensing-oriented network represent almost the total charge. Referring to standard SQL, TinyDB could be classified at a ”repeatable read” level, associating the effects of an OUTPUT ACTION to those of a ”phantom insert” for conventional databases.

2.5.2 Others

TinyDB is completely developed by Berkeley University and Intel Research Laboratories. Other projects have been undertaken in trying to apply Database models to sensor networks and two of them are worthy of mention: Cougar Sensor Database Project from Cornell University [49] and SINA [50].

2.6 Reprogramming Support Network

The idea of a reprogramming support network has been brought forward by some researchers from ETH Zurich. In their article, Beutel et al. [51] propose the implementation of a Deployment Support Network (DSN) to overcome the nuisances of passing through the main sensor network to change the configuration of the nodes. The DSN plays its role only for maintenance operations and works in parallel to the controlled WSN. A DSN is conceived to be minimal, just few small nodes that creates a bridge between the Host and the various sub-trees in WSN’s hierarchy: a certain set of WSN’s nodes is assigned to one and only one DSN’s node, through which the updates are propagated from the Host. The main advantage of this proposal regards the conspicuous power saving of the WSN’s sensors, which could continue accomplishing their tasks without loosing time and power in forwarding maintenance information and updating code. On the other hand, there are many troubles and first of all the overhead to maintain a new parallel network, since the

DSN is in turn a new, even if small, sensor network. Besides, there are also problems in preserving the hierarchy between WSN's and DSN's nodes, letting alone the issue of maximizing the lifetime of DSN's nodes as well.

2.7 Agent based approaches

It's a refinement of the script-injection mechanisms, which allows the deployment of dynamic, localized and intelligent mobile agents among the network. This approach, as often happens in leafing through the literature, promises a lot of advantages for what bears on flexibility in agent dissemination, self-positioning to perform specialized tasks, state and code maintaining in migrations and locality principle, by which the computation is brought where data reside and not viceversa. Contra, we think it appropriate to underline the heavy resource requirements of agent-based platforms, hardly ever applicable to small sensors. Moreover, these techniques suffers of the bothersome problem to be not reliable when a network partitioning or a message losing occurs: in presence of those anomalies, the protocol fails. Maybe, in proceeding the development of robust coordination mechanisms and reducing the resource usage of agent controllers, this vein of WSN research might become a very attractive one. Following we present the Agilla mobile agent middleware, an important example for this type of WSN-oriented software updating.

2.7.1 Agilla

The most famous agent based reprogramming mechanism is Agilla [52] [53], a middleware purpose-made for supporting the dissemination of intelligent mobile agents. Agilla grounds on TinyOS and targets large nodes, considering the heavy size of agents' code. An agent is programmed in assembly and then injected into the network to be encompassed on top of the Agilla layer. A migration is triggered by a move or clone instruction and is performed through a store-and-forward network packets exchanging; each packet is 41B sized, very small compared to a whole agent, so many transmissions are required to deploy a module.

Every node in the network is provided with a local data space, organized in shape of a list of tuples, and the communication among agents is obtained through remote accesses to these space. Agilla stands a very promising paradigm, given the assumption an increasing availability of resources in next generation nodes' architectures. The chance of bringing the computation directly in the place where data are collected is a very attractive one, so it is plausible a focusing of the Academic Research towards this class of techniques. Many flaws must be however overcome, from the assembly-based programming, to the remote agent suspension/activation and a targeted distribution control.

2.7.2 A proposal from the University of California Davis

In 2005, some researchers from University of California Davis [54] have proposed an interesting self-regulating agent-based framework. It is self-regulating since the propagation of agents in the network is obtained through special forwarding procedures, directly invoked on the agents, following up a unicast or broadcast request from nodes. Moreover, the agents communicate by leaving some data in the nodes they have passed through, to be used by other agents which will pass through the same nodes in future. An agent can be transitional or permanent, depending on the code structure and in particular on the presence of exiting functions or endless loops. Injecting new agents is very simple, but no support is provided to update the existing ones. The main advantage of this framework is the completely self-regulating propagation and the drastic radio transmissions reduction, since communications are obtained leaving data in the node behind, for future agents to come. The problems are the usual concerned to these kind of approaches and touch the heavy byte-dimension, the low level bytecode programming language (since also this technique bases on Maté) and the difficult in managing versions upgrading and nodes configuration traceability. This could be considered a good solution for large networks, with powerful nodes and the need for autonomous configuration, requiring the introduction of new self-regulating agents rather than the upgrade or the modification of the existing ones.

2.8 Others

Many other experiments have been carried out in this direction and many other are being performed from important international centers. Actually, a group from Politecnico di Milano, leaded by Prof. Di Nitto, is involved in the research of an agent-based protocol for WSN, targeting large nodes and capable of modules' self-distribution, application interoperability and code portability. Also in the past have been proposed interesting solutions, such that of Umezawa et al. in 2002 [55]. All these mechanisms show the same advantages and disadvantages above discussed and the Research is divided in two schools of thought: one is now focusing in overcoming the resource-requirement overhead, while the other is involved in upgrading the applicative complexity disregarding the resource-consumption problem, in the perspective of a breakthrough in WSNs' nodes hardware.

2.9 Pre-linked loadable modules

Loadable modules are used to upload programs or functions in a node, without performing costly replacements of the whole system image or managing the binary differences between old and new versions. Typically, a loadable module contains the executable binary with the physical addresses of variables and functions resolved at host level. In order to assure the correct execution of binaries, all the symbolic references to global variables and functions of the target node must be replaced with the correspondent physical addresses. This process is called linking. Once the code has passed the linking phase, one more operation is needed: the symbolic references to local variables and functions have to be resolved. This applicative step is known as relocation.

In a pre-linked module, the linking phase is always executed at host level, whereas the relocation could be performed both at host and node level. Performing relocation at host level allows smaller size in the output binaries and, at the same time, relieves the node from the costs of local processing, since the module can be immediately loaded. Instead, leaving to nodes the burden of relocating local symbols leads to more onerous computations, but programmers can dispense with the need of maintaining the information about nodes' address-spaces. Choosing the first or the second approach depends on the sensors' architectures (supporting absolute address vs. relative addresses) and the network dimension.

This type of reprogramming system is not supported by all operating systems: TinyOS is unsuitable, while Contiki, SOS and Impala for ZebraNet are examples of compatible systems. The main advantages of this procedure are the lightweight nature and the fast reprogramming capabilities: the efficiency of such a kind of technique is very high and far to be comparable with the others above and below discussed. In reverse, some critics are moved referring to the presumed low effectiveness in managing large networks [9]. The bottleneck just introduced is due to many troubles in maintaining aligned the information about nodes' absolute addresses when some changes (even small) occur. In Contiki, for example, both linking and relocating phases are performed in-host by means of map files, one for each node. These files are generated at compile time when the operating system is loaded for the first time into the nodes and they're never ever updated. A map file contains the correspondences between all globally visible functions and variables in the system core and their address, that explains why a minimum change in the address-space makes the pre-linking phase unmanageable. In Contiki, a version-based control system has been recently introduced: that helps against crashes but not solves the problem.

Starting from these considerations and looking at the enormous benefits of this approach, compared to its flaws, we have reviewed the entire protocol and studied a way to retrieve the most updated information about the address to be resolved, just in time to perform the pre-linking phase. Our approach [3] is designed for msp430 architectures and is performed retrieving on-the-fly the symbol table from a node, for which we want to produce a new module. The symbol table is uploaded on the node at the moment of its initialization and contains the correspondences between the names of functions (and variables) and the pointers to the relative addresses. If a change occurs in the address space, the pointers are automatically updated, preserving the consistence in the references. Downloading from a node its symbol table (which is generally very small-sized) just before to accomplish the linking phase, we have at our disposal the current addresses of variables and functions in the target system. Thanks to its dynamical nature, this proposal reveals very good performances and takes place in half between a pre-linking and a dynamic linking procedure, so that we will discuss it in the section dedicated to dynamic linking.

2.9.1 SOS loadable modules

The way through SOS [56] supports dynamic reprogramming is based on small pre-linked modules; these modules are loaded or unloaded, depending on the configuration that a programmer wants to give to the network or to a subset of it. Just the kernel and the hardware interfaces are embedded into SOS core, all other services are supplied by the above-quoted dynamic modules. The kernel is rigid, once loaded into a node no dynamical updates are possible. When a new module is created, it is timestamped with its version number, so that a node can check its own version and determine if an update is needed or not; to ease modules management, some standard function have been defined to initialize, finish, receive message and interface with hardware. Module's specific functions are registered with their name, version and address at the global Function Control Block (FCB), in that way other modules can invoke these functions by simply accessing the FCB. The kernel automatically handles as exceptions eventual calls to unreachable or unregistered function. In loading a new version of a module, the older version is unregistered from FCB, then the new module is loaded and the FCB updated with new module's function references. This is a very efficient mechanism, suitable for network of medium-small dimensions and relatively large, homogeneous nodes, running SOS operating system. Compared to other approaches, the disadvantages are not particularly critical, with the exception of lack for heterogeneous interoperability and scarce scalability for medium-high or high sized network, with a considerable number of nodes.

2.9.2 Impala modules

ZebraNet is a famous WSN project from Princeton University and Impala [22] is its event-based middleware underneath programs updating. Applications are deployed by means of generic loadable modules, 2KB sized, fitted to nodes' specific environment through special-purpose application adapters. As seen in other proposals, also Impala uses a version numbering system to assure compatibility among the various releases of the same application and each module is provided with a complete index that identifies the reference application and some other parameters.

The entire deployment process is managed by an application updater, allowing multiple parallel updates and integrity checks, even while other tasks are running. Older or incomplete modules are kept in flash till the memory space doesn't become critical: if so, they're deleted.

When a new module has to be updated, a request for its index is sent to source nodes and, if index's parameters fit in, the download starts through a direct connection between one of the sources and the requiring node. Once a module is downloaded, the target node becomes itself a source and can eventually respond to requests from other nodes. In order to manage eventual multiple downloading requests, a version-based prioritizing mechanism takes the field in scheduling downloads among nodes. The new module is then loaded and its execution gets started.

When an event occurs, it is caught by an event filter dispatcher which interacts with the application adapter that chooses, in turn, the most appropriate application to handle that event.

The advantages and disadvantage of Impala trace the ones already discussed for SOS and the general approach.

2.9.3 Others

Similar techniques are adopted by ContikiOS, through its pre-linked loadable modules [43] , and MIT's Bertha operating system, which uses the Pushpin protocol [57] to upload applications. It is worth to spend some words about Bertha's Pushpin, since it slightly differs from standard modules-based approach. It places across a mobile-agent and a module-based technique, since the 2KB modules are substantially fragments of one process, just-in-time called during the execution. The local communication among fragments is obtained using a shared memory fragment, comparable to a bulletin board. The communication between nodes, at the same hierarchy level, happens through their bulletin boards, since every node stores the bulletin board of its neighbors.

One more solution, similar to that of SOS, is provided by ScatterWeb [58]: some special-purpose, modifiable tasks are hooked-up to a common, rigid firmware, supporting core operations. Modules containing applications can be downloaded and installed. The non-modifiable firmware is in charge of managing the updating process and checking its correctness.

2.10 Dynamic Linking

In dealing with dynamic linking, we can draw up a basic distinction between what can be considered a ”pure” dynamic linking approach versus a ”functional-equivalent” one. In a pure dynamic linking approach, the object code is exactly the same of the classical ”.exe” windows files or linux ”.elf”. These files contain references to variables and functions only in the form of symbolic names, then replaced by a run time linker with the corresponding addresses. A ”pure” application has always been considered too much expensive for a sensor platform, both for the heavy processing time and the huge memory space required. The dimension of an ”.elf” file, in fact, fluctuate in the vicinity of 200KB: a boulder, considering that the RAM and FLASH dimensions of a small node (e.g. a TelosB) are about 10KB and 30KB, respectively. Only one, interesting experiment has been registered in that way by some researchers of the Swedish Institute of Computer Science [9], who have implemented an ELF dynamic linker for ContikiOS (then ported to MontisOS). That arrangement, however, comes out completely unsuitable for small nodes (due to the discrepancy between the dimension of an ELF-CELF file and a typical FLASH memory) and continues to represent a bad compromise also for bigger nodes, inasmuch as pertains radio-transmission costs. A ”functional-equivalent” approach, to the contrary, consists in simulating the effects of a ”pure” dynamic linker using some technical expedients to reduce power and resources consumption. Hereby we mean to produce an actual change in the software configuration of a node, working in-host at a low level (i.e. relocating symbols with physical addresses, etc.) and discharging the target system from bearing the brunt of a tough computation and an heavy memory allocation. Furthermore, in order to be considered a functional variant of dynamic linking, a process must work on executable binaries and not on intermediate languages, otherwise it would fall back in one of the previous categories. In the latest family of dynamic linkers may be collocated some implementations fulfilling their work with some arrangements that render such processes less autonomous but more efficient than the ”pure” dynamic-linking ones: FlexCup for TinyOS, which requires a system reboot after a modification, SOS reprogramming system (with Position Independent Code) and Impala are the main examples of this way of operating.

2.10.1 The Contiki’s dynamic linker

In 2006, Dunkels et al. [43] from Swedish Institute of Computer Science have proposed the first implementation of a dynamic linker for wireless sensor network, running on top of ContikiOS.

This linker has been projected to link, relocate and load both ELF files and CELF files, which are a compact version of ELFs, obtained by compressing the data types from the original 32-bit to a 8-bit or 16-bit dimension, reducing the file size by an half. The reprogramming process is based on the deployment of a full ELF/CELF file towards the target nodes, which process it through their local dynamic linker. ContikiOS is provided with a virtual filesystem, that substantially ease file processing, since the dynamic linker can work on ELF/CELF without taking care of the actual physical location.

The linking process is performed in four steps:

1. The ELF/CELF is parsed and information about code, data, symbol table

and relocation tables are extracted;

2. ROM and RAM are allocated respectively for code (plus read only data) and data segments;
3. Code and data segments are linked and relocated on the basis of the information collected in the first step and the memory space allocated in the second step;
4. Code and data segments are finally written in ROM and RAM, respectively.

It could be noticed that such a process is identical to a standard dynamic linking for common PCs and workstations. The disadvantage of an ELF/CELF-based reprogramming is due to the huge size of that files (200KB in average for an ELF and about 100KB for a CELF), that sometimes overwhelms the available memory of a small node, e.g. TelosB. Another problem is related to the tough processing-effort required to relocate these files, that for a small node comes out very expensive in a power-consumption perspective. On the other hand, the advantages are alluring, considering the possibility of working with a standard ELF format, also used by common PCs as well as by powerful elaborators. To rise above the above-quoted drawbacks, the authors have proposed an hybrid implementation, based on dynamic linker and virtual machine cooperation to balance the overall workload. This approach is quite an expensive one and results practically unsuitable in dealing with small nodes. Maybe, in the proceeding of hardware's development, it could become a very attractive one.

2.10.2 FlexCup

FlexCup [59], i.e. FLExible Code UPdates, performs a local dynamic linking on pre-compiled modules sent by an host station. It uses some meta-data, that each node stores in external flash to enable the linking phase. These meta-data consist of a symbol table, containing names and address of all global functions and variables, some relocation tables, one per each component reporting the symbol table entries called by that component, and generic program information, listing the number and relative offsets of all binary components, as well as the addresses of the symbol and relocation tables.

FlexCup has been proposed in two version, the first is called FlexCup Basic and transfers the whole binary component and its meta-data without considering the data already stored on the sensor node . The second, more efficient, is called FlexCup Diff and only transfers the incremental changes between the new binary component and the one already stored on the sensor node; this version needs the host to know the current status of the target node.

The linking process could be conceptually divided in 5 phases:

1. Storage of code and metadata: this phase involves receiving the update data, including code and the meta-data of the component, and storing it into external flash memory.

2. Symbol table merge: the global symbol table is combined with that of the module just received through a merge-sort algorithm, thanks to the ascendant sorting of the entries. This operation is accomplished in a 3KB buffer area of the main memory, allowing fast and energy-efficient processing.

3. Relocation table replacement: since each component contains an individual relocation table, sent as part of the component update, this step only involves copying the new relocation table to the appropriate location and, if necessary, shifting the following tables backward by the right amount of bytes.

4. Reference patching: it consists in going through the entries of the relocation tables of all components, and checking whether any of the references needs to be updated. An update is required for all references coming from the new component code and for all references to symbols that changed their destination address during the second step.

5. Installation and reboot: it consists in copying the program code from external flash to program memory and reboots the sensor node afterwards.

The main advantages of FlexCup deal with the intrinsic scalability and flexibility that a dynamic linker bestows on reprogramming issue. The FlexCup Diff, moreover, substantially reduces radio transmissions but needs, as said, to maintain the host aligned with nodes' statuses. On the other hand, the disadvantages are not negligible, especially the energy consumption caused by a massive usage of external memory, during linking and relocating operations. More, it doesn't face the problem of wear levelling in flash memory remains.

2.10.3 Others

Actually, no more dynamic linking approaches have been proposed for Wireless Sensor Networks.

2.11 Comparison

Following (Table 2.1) we propose a comparison of the various reprogramming approaches, based on their pros, cons and possible applications.

For what concerns the goal of minimizing the power consumption, the approach based on pre-linked modules is certainly the most advantageous: it presents very interesting requirements in terms of memory usage and processor time and, at the same time, delivers good performance for what bears on the reprogramming effectiveness. There are, however,some disadvantages related to this approach because of the need, on Host side, to store all the images of the nodes at compile-time. That makes impossible, in consequence of even small changes in the configuration of a node at run-time, to link and relocate further modules for that node. A possible way to preserve the benefits of pre-linked modules, while avoiding their flaws, could be obtained by retrieving on-the-fly all the significant information relative to the actual addresses of the symbols to be relocated. This enables further linking processes also for those nodes whose images have incurred in modifications at run time. In the following chapter we describe the solution that we have conceived as an hybrid configuration combining the strong points by two of the previous techniques: pre-linked loadable modules and dynamic linking. The idea

Approach	Pros	Cons	Application	e.g.
Image Replacement	Low processing costs	Not scalable, not flexible, not incremental, high transmission costs	Small networks with few, homogeneous nodes	XNP, MNP, MOAP, Deluge, MantisOS, NutOS, etc.
Diff-based	Low processing costs, Low transmission costs	No multicast nor broadcast updates, not scalable, no heterogeneity support	Only academic proposals for small homogeneous networks	Koshy and Pandey, Reijers and Langendoen, Jeong and Culler
Script-based	Fast programming, small scripts' dimension, low transmission costs, scalability, heterogeneity	Excessive memory usage, excessive running overhead, high power-consumption	Short-living networks with large, powerful nodes	SensorWare, COMiS, Spatial Programming, DeePython
Virtual Machines	Fast programming, small-sized bytecode, low transmission costs, scalability, heterogeneity	Excessive running overhead	Short-living networks with large, powerful nodes	Maté, SwissQM, VM*, CVM, Magnet OS JVM
Distributed Databases	Query programming, small packets, scalability, heterogeneity, process integration, low processing costs	Frequent radio transmissions, maintenance overheads	Large networks	TinyDB, Cougar, SINA
Reprogramming support network	WSN's power saving, efficient reprogramming accesses	Maintenance overheads	Any kind of networks	Deployment Support Network
Agent based	Loacal data processing, easy to maintain, intelligent interactions, scalability, heterogeneity	Excessive memory usage, excessive running overheads, high transmission costs	Large networks with large powerful nodes	Agilla, Szumel et al., etc.
Pre-linked modules	Small sized packets, low processing costs, low transmission costs	Not scalable	Small-medium networks	SOS,Impala, Contiki,Bertha, ScatterWeb
Dynamic linkers	Scalability, easy to maintain	High processing costs, high transmission costs	Large short-living networks	Contiki DL, FlexCup

Table 2.1: Synopsis of the various approaches

is to provide each node with a global symbol table, which can be downloaded along with some additional information required for the linking and relocation phases. Once such information have been acquired by the Host, it is possible to process a standard executable format (ELF) and to create a ready-to-load module for that node, which encompasses all the eventual changes previously occurred on its image. A module so obtained is very small compared to the dimensions of an ELF and it is also immediately loadable: the huge gain in byte-dimensions, entailed by these modules, allows smaller radio transmissions as well as minimal processing time, relieving the nodes of expensive accesses to the external Flash memory. The Host, in fact, takes in charge both the linking and the relocation process, which are the most expensive operations in terms of processing time and also require the heavy ELF files, which have to be transformed, to be stored in local memory.

Chapter 3

Dynamic Linker

This method has been developed in the sphere of WASP European Project [2] and targets the creation of a lightweight mechanism for dynamic linking, capable of running on the TelosB motes, equipped with Mantis Operating System. In developing this system, many low-level aspects have been taken into account, from the linux ELF functioning to the MantisOS specific structure and TelosB's msp430 architecture features. Moreover, in order to build the effective implementation, some open-source code of the not-suitable dynamic linker ported from Contiki to MantisOS has been structurally modified.

Getting back to our proposal, it consists in a two macro-phases processes, the first of which lays at Host-side and regards the linking and compacting of an original ELF file into a WLF (Wasp Loadable Format) module. This module contains only the strict information needed to execute the code on the node (WLF reduce the byte dimension of the original ELF of about the 95%). The second macro-phase deals with the loading of a WLF file into the node: it's performed thanks to a message-exchanging protocol between host and node. These macro-phases could be subdivided in a set of four finer-grained phases, three of which take place at host-side and lead to the creation of a WLF module from a standard ELF, while the last phase enterily overlaps the second macro-phase.

On the node side, a server program has been developed with two main functionalities:

- transmit the Global Symbol Table from node to Host, if required;
- load a WLF from Host to Node.

This solution eliminates the need of loading an application at bootstrap time and thanks to the server running on the node many application could be loaded and unloaded, even at distance, during node lifetime without reprogramming its core.

3.1 Something about WASP Project

WASP project has been undertaken under the aegis of European Community by some of the most important Enterprises and Universities, in order to neaten the current status of WSNs' Research.

This project aims to set a standardized paradigm for all the facets composing ”‘WSN landscape”', from hardware to software design and implementation, with the ultimate purpose to make effectively available these kinds of technologies for commercial uses. WASP will be tested on the field, considering three principal areas: managing traffic conditions, monitoring elder people health parameters and controlling movements of livestock herds.

At the end of the job, three main objectives are expected to be reached: provide a global HW/SW framework to make WSN installations effective and economically affordable, maximizing motes' lifetime and defininig an high-level infrastructure to enable easy programming approaches for developers. The implementation of a specific mote and a dedicated operating system is foreseen in the sphere of this project. Since in developing our dynamic linker, these feature had not already been issued, we have based our work on two systems, supposed to be similar: TelosB motes, for what concerns HW platform and Mantis OS for what is related to the operating system layer. In sections 3.2 and 3.3 we propose a brief introduction to these features.

In a broad quantitative analysis, the objectives of WASP could be synthesized as:

1. increase of one magnitude order the number of supported applications' types, against the actual average value in WSN state of art;
2. raise to more than ten times the adaptability to different,concurrent applications;
3. reduce by a 30% factor the energy consumption of a running task in WASP environment, under standard communication and routing parameters. Moreover, bound to a top of a 10% exceeding rate the eventual power-absorption of large applications, once more compared to the actual available environments;
4. reduce by an 80% factor the code dimension for standard application exchnge;
5. reduce by a 70% share the time needed to dynamically optimize execution at application level.

Many other problems have been faced in designing WASP infrastructure: the communication protocol is one of those and a first draft has been taken out by means of two different protocol stacks, working at different layers. Each protocol, for each layer, has demonstrated good performances in touching optimization tasks and the choose of one rather than another has been delayed to additional tests based on intra-protocol compatibility, application-purpose suitability and maturity degree of the enterprise/university partner, which has proposed the specific implementation. Another issue is related to the Hardware abstraction, in trying to create a common interface, eventually sharable by different operating systems: that's consists in defining a general description of the functionalities provided by a generic node, easily adaptable to specif platforms; by now, this work has been done for what regards the use and management of low-level timers, inspecting the graphs of Hardwares and reciprocal dependencies; other components are already to be investigated.

The last, great challenge regards the creation of an Operating System abstraction: this work is actually in progress and is focusing on patterns-discovering among similar functionalities of different operating systems, such as blocking/non-blocking function calls, but a suitable solution is far from being proposed. In addition to this research on patterns, three important capabilities have been identified to become part of that operating system abstraction or an eventual WASP operating system: the need for a multi-threading architecture, a real-time support for special-purpose applications and a non-functional manager, capable of switching the operative mode of a node in dependence of power-consumption and residual autonomy.

3.2 WASP hardware platform

TelosB [1] was developed by UC Berkely researchers attempting to create an ultra low-power node, enforcing three main purposes: minimal power consumption, ease of usage and software/hardware robustness. The mass production of this kind of motes has been devolved upon Crossbow Corporation, which has also improved some features and has focused on market targets related to experimentations and projects by research community.

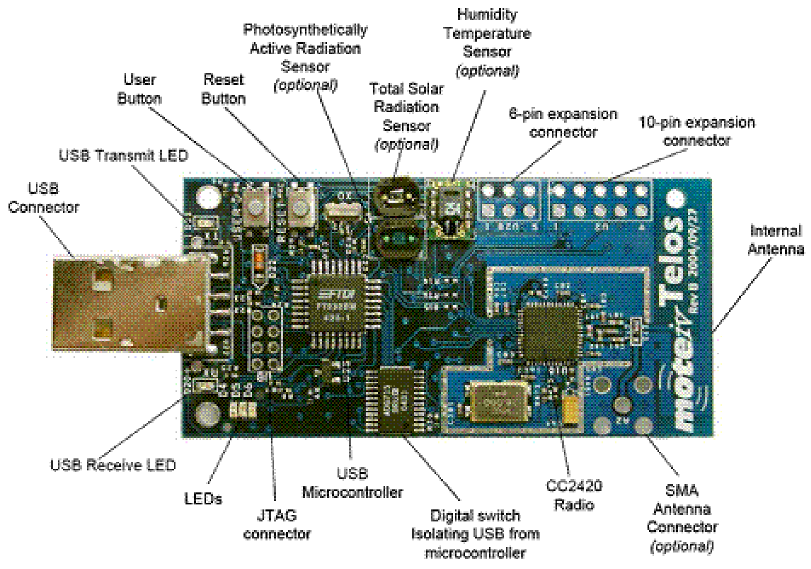

Photosynthetically Active Radiation Sensor (optional)
Humidity Temperature Sensor (optional)
User Button
Reset Button
Total Solar Radiation Sensor (optional)
6-pin expansion connector
10-pin expansion connector
USB Transmit LED
USB Connector
Internal Antenna
USB Receive LED
LEDs
JTAG connector
USB Microcontroller
Digital switch Isolating USB from microcontroller
CC2420 Radio
SMA Antenna Connector (optional)

Figure 3.1: Detailed description of a TelosB mote

The architecture itself has been developed looking at lab-studies' critical factors, providing some useful features as USB programming capability, IEEE 802.15.4 radio with integrated antenna, low-power MCU with extended memory and an optional sensor suite (TPR2420). The main characteristic of a TelosB are:

- IEEE 802.15.4/ZigBee compliant RF transceiver
- 2.4 to 2.4835 GHz, a globally compatible ISM band
- 250 kbps data rate
- Integrated onboard antenna
- 8 MHz TI MSP430 microcontroller with 10kB RAM
- Low current consumption
- 1MB external flash for data logging
- Programming and data collection via USB
- Optional sensor suite including integrated light, temperature and humidity sensor (TPR2420)

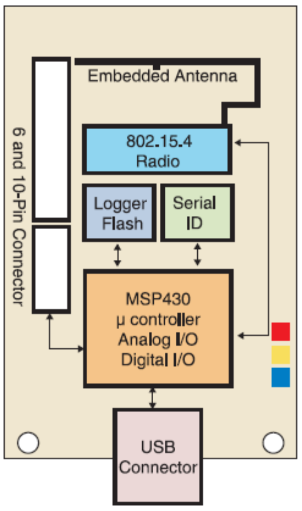
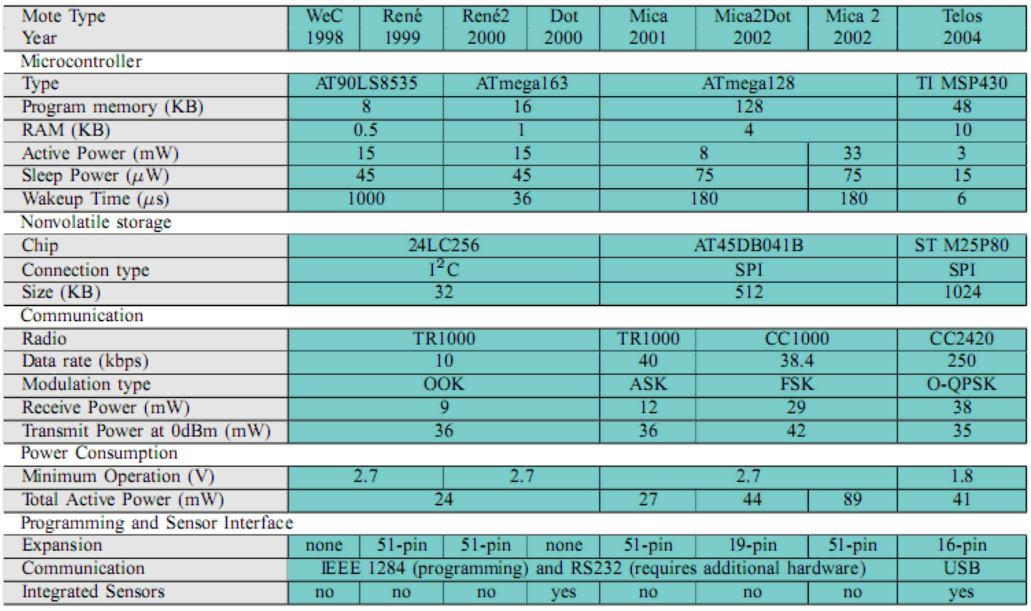


Table 3.1: Specs of most diffused sensor nodes

As mentioned, this platform delivers low power consumption allowing for long battery life as well as fast wakeup from sleep state. Some useful technical data concerning power consumption and compared to those of other devices could be found in Figure 3.2; we have used some of these parameters to estimate power consumption during our experiments:

This mote is powered by two AA batteries, but if it is plugged into the USB port for programming or communication, power is provided from the host computer, avoiding the need of batteries. Some additional devices are available, for example, two expansion connectors and on-board jumpers may be configured to control analog sensors, digital peripherals and LCD displays.Anyhow, none of



Mote Type Year	WeC 1998	René 1999	René2 2000	Dot 2000	Mica 2001	Mica2Dot 2002	Mica 2 2002	Telos 2004
Microcontroller								
Type	AT90LS8535		ATmega163		ATmega128			TI MSP430
Program memory (KB)	8		16		128			48
RAM (KB)	0.5		1		4			10
Active Power (mW)	15		15		8		33	3
Sleep Power (μW)	45		45		75		75	15
Wakeup Time (μs)	1000		36		180		180	6
Nonvolatile storage								
Chip	24LC256				AT45DB041B			ST M25P80
Connection type	I^2C				SPI			SPI
Size (KB)	32				512			1024
Communication								
Radio	TR1000				TR1000	CC1000		CC2420
Data rate (kbps)	10				40	38.4		250
Modulation type	OOK				ASK	FSK		O-QPSK
Receive Power (mW)	9				12	29		38
Transmit Power at 0dBm (mW)	36				36	42		35
Power Consumption								
Minimum Operation (V)	2.7		2.7		2.7			1.8
Total Active Power (mW)	24				27	44	89	41
Programming and Sensor Interface								
Expansion	none	51-pin	51-pin	none	51-pin	19-pin	51-pin	16-pin
Communication	IEEE 1284 (programming) and RS232 (requires additional hardware)							USB
Integrated Sensors	no	no	no	yes	no	no	no	yes

Figure 3.2: Comparison between TeloB's and other motes' parameters.

these features has been used by us during our experiment.

Each Telos node could be customized with a wide range of special-purpose sensors: typical applications include utility metering, portable instrumentation, intelligent sensing, and consumer electronics. A detailed description of a Telos mote could be find in Figure 3.1.

One more analysis is opportune for MSP430 microcontroller, since this is the device whose specs have most conditioned our choices, regarding the low-level features that our dynamic linker has had to take into account. MSP430 is a microcontroller family by Texas Instrument, providing ultra-low-power, 16-bit RISC mixed-signal processors and a specific layer for battery-powered measurement applications. Getting to the core of the architecture (Figure 3.3), the 16-bit RISC CPU, peripherals and flexible clock system are combined by using a von-Neumann common memory address bus (MAB) and memory data bus (MDB). Partnering a modern CPU with modular memory-mapped analog and digital peripherals, the MSP430 offers solutions for mixed-signal applications.

The key features of MSP430 microcontroller could be summarized in:

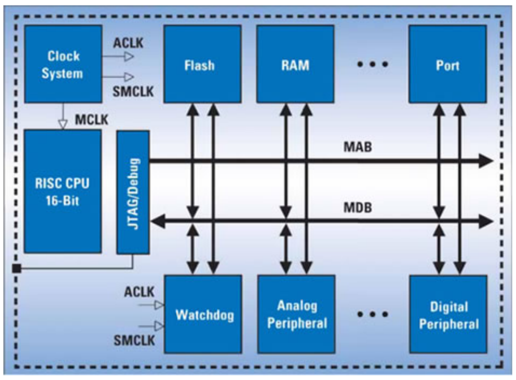


Figure 3.3: MSP430 microcontroller architecture

1. Ultra-low-power architecture extends battery life:
   - 0.1-A RAM retention
   - 0.8-A real-time clock mode
   - 250-A/MIPS active
2. Wide range of integrated intelligent peripherals offloads the CPU
3. Modern 16-bit RISC CPU enables new applications at a fraction of the code size
4. Complete development tools starting at only $20
5. Devices starting at $0.49

One consideration must be outlined for what concerns MSP430 architectures in a dynamic linking perspective: this CPU only supports 1 type of relocation, against the 19 different types of an AVR architecture [43]. More, this platform does not support Position Independent Code, which could eliminate the relocation step by using symbolic references, but no a compiler is known to make MSP430 be fitted to this purpose. The 16-bit nature of CPU ought to be treated with same care, since common WSN applications are first written and compiled at Host level, often by means of powerful 32-bit or 64-bit architectures. This means that all the address should be compacted to a 16-bit dimension before porting an application from host to node, allowing code reduction, radio efficiency and memory saving at node level.

3.3 MantisOS

Since a dynamic linking process needs support by the Operating System, we have had to study in detail the structure and the main characteristics of MantisOS, which is the target platform, in order to properly design our DL.

3.3.1 A brief introduction

MantisOS is a multi-threading embedded Operating System. It is designed to provide energy efficiency and advanced sensor specific features: at present, MOS kernel is able to achieve multi-threaded preemptive scheduler execution with standard I/O synchronization and a network protocol stack, all for less then 500 Bytes of RAM, threads' stacks excluded. The scheduler adopts a round-robin priority-based policy, with a time slice of 10ms and 5 priority-levels (from high-to low: kernel, sleep, high, normal, idle). MantisOS Kernel is written in C language, hence kernel development can leverage the same skills used for an application development. The choice of C-language's APIs simplifies cross platform support and the development of a multi-modal prototyping environment.

The multi-threaded nature of MantisOS provides a valid solution to the classic bounded buffer (i.e. *producer-consumer*) problem, interleaving packets-processing with executions of multiple long-living complex tasks [33]. Mantis also offers a command shell, called MCS, which permits accesses from serial and radio interfaces.

3.3.2 General architecture

MantisOS consists of a kernel with an integrated scheduler, a command server and a device driver system (Figure 3.4). The scheduler supports mutual exclusion

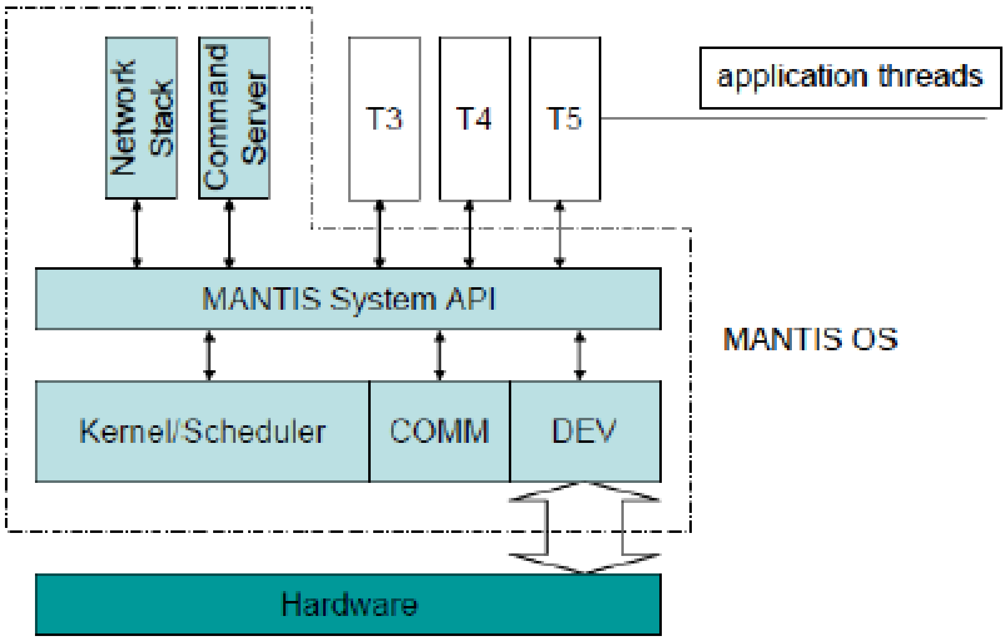


Figure 3.4: Mantis architecture

(binary) and reference-counting semaphores. Further important components of the Operating System are: a low-level communications stack for serial or radio communication interface (which services the lower layers up to and including the MAC layer) and a device abstraction layer, that provides uniform accesses to devices of all sorts. We remind to [33] and Mantis website for further descriptions.

Mantis' kernel is designed on the Unix model, with a round-robin priority scheduler and a POSIX-like threads management. Since RAM is a critical resource, MantisOS logically divides the main memory into two sections: one for the global variables (allocated at compile-time) and the other used as an heap. The core aspect characterizing MOS system is its multi-threaded nature; when a thread is created, the kernel opportunely allocates stack-space out of the heap. Since memory is very limited, the heap allocation should be performed statically, at compile-time, even though newer MOS versions implement a best-feet policy in managing memory.

The main data structures are:

- *Threads table.* This table is statically allocated, with a maximum number of 12 concurrent threads and a fixed level of memory overhead. There is a 10 bytes entry point for each thread, composed by the stack pointer (and stack-size information),the pointer to thread's function, the priority level and a next-thread pointer. All pointers are 2 bytes sized. The overall size of threads' table is 120 bytes: 12 threads * 10 bytes (10 bytes is the entry value of a single thread).

- *Threads' execution contexts.* A thread execution context (along with current register-values) must be stored when a thread is suspended. These information are used by the scheduler while alternating the execution among threads.

- *Priority queues.* The kernel maintains two pointers for each of the five priority levels: one pointer references the head, while the other points to the tail of the list. These pointers permit fast and frequent manipulations of priority lists (at input-disabled). The total size for this information is 20 bytes.

- *Semaphores structures.* Semaphores are represented by 5-bytes structures, composed by a lock (or count byte) and two pointers referencing, respectively, the head and the tail of the list. At any given time, a thread is member of exactly one of the two possible lists: ready-list or semaphore-list. The operations provided by Mantis' semaphores essentially consist in moving thread-pointers between these two lists, so that the scheduler can choose a thread from ready-list [33].

- *Other fields.* The kernel also manages a current thread pointer of 2 bytes, an interrupt status byte and 1 byte of flags. (* *Adding all the bytes needed by the previous data structures, results a TOTAL STATIC OVERHEAD of 144 bytes for the scheduler*)

Interrupts are handled by Mantis scheduler and could be categorized in:

1. *Timer interrupts.* These interrupts are directly managed by the kernel. They are used to switch between threads, according to the round-robin policy. An alternative method to trigger a thread-switch makes use of system calls or semaphore operations.

2. *Device-driver interrupts.*A device interrupt posts one semaphore in order to activate a waiting thread. This thread is responsible of handling the event that caused the interrupt.

There are four types of threads:

- *System level threads.* There are two system level threads in Mantis: the command-server thread and the network-stack thread. Both of them are loaded at system boot.

- *User level threads.* These threads are managed according to the round-robin priority policy. For each priority level is defined a list, so that lists with higher priority dominate the ones with a lower level. Also sleeping jobs are managed in a specific queue, called sleep queue.

- *Driver threads.* A driver-thread is activated when an interrupt is received from the relative device. The task of a driver-thread consists of handling the event which caused the interrupt.

- *Idle thread.* This is a low-priority thread, running when all other threads are blocked and enforcing power-aware scheduling (by optimizing CPU utilization and kernel parameters).It should be noticed, however, that moving these functionalities directly into the kernel could lead to better energetic performances.

3.3.3 How MantisOS supports dynamic reprogramming

By now, the dynamic reprogramming capability is implemented as a built-in system call library [33]. A combination of system calls to this library, a commit function and a boot loader enables the application to write a new system image.

After Mantis' boot-loader have installed a new image, a specific reset application makes the changes to become effective.

This process has been designed to enable remote differential updates by means of binary patches or alternatively an in loco full reprogramming by connecting through serial port and shell functions; this *in-loco reprogramming* does not satisfy the requirements for scalability in widespread networks. Regarding to remote differential updates, the system status must be known at compile-time, for each node, introducing heavy scalability constraints.

3.4 ELF: Executable and Linking Format

To better understand the linking process, it is necessary to briefly introduce the structure of an ELF[1], which is the file-format used by the dynamic linker to produce a smaller and ready-to-load module for a TelosB node, provided with Mantis operating system.

As the name suggest, ELFs are modular files that contain, along with the executable, all the information needed by a Linker to resolve the symbolic names of functions and variables with the actual addresses for a target platform. Tough very flexible and powerful, an ELF is not adequate to be sent and linked directly on a small node, because of its huge size (in average, from 20 KB to 200 KB). From this comes the idea of relocating and linking the executable on the Host in order to create a small loadable module for a specific target node, by means of some meta-data (the Symbol Table) previously retrieved by the node itself. This process is the core idea of our dynamic linker and will be presented in Section 3.5: here we will just present the overall structure of an ELF file, with a special focus on the ELF generated by the MSP-430 compiler for Mantis operating system.

An ELF is composed by an `ELF header` (Table3.2) and a certain number of sections, each playing a different role in the linking process. The information about these sections are contained in a specific `section header` (Table 3.3), whose function is to provide information about the various sections composing

[1]A detailed description could be found in [60].

`#define EI_NIDENT 16`		
`typedef struct {`		
`unsigned char`	`e_ident[EI_NIDENT];`	identification bytes.
`Elf32_Half`	`e_type;`	file type.
`Elf32_Half`	`e_machine;`	target machine.
`Elf32_Word`	`e_version;`	file version.
`Elf32_Addr`	`e_entry;`	start address.
`Elf32_Off`	`e_phoff;`	offset of program header.
`Elf32_Off`	`e_shoff;`	offset of section header.
`Elf32_Word`	`e_flags;`	file flags.
`Elf32_Half`	`e_ehsize;`	size of this header.
`Elf32_Half`	`e_phentsize;`	size of program header.
`Elf32_Half`	`e_phnum;`	number of entries in program header.
`Elf32_Half`	`e_shentsize;`	size of section header.
`Elf32_Half`	`e_shnum;`	number of entries in section header.
`Elf32_Half`	`e_shstrndx;`	section header strings index.
`} Elf32_Ehdr;`		

Table 3.2: Structure of the ELF header

the file[2].

Some of these sections contain the executable to be linked. In particular, our platform's ELF presents four of such sections:

.text This section contains the executable and all the compiled functions: it is the most important section of an ELF. The overall logic underneath the ELF paradigm, in fact, had been studied to enable the relocation and the execution of this section, by resolving the symbolic names of variables and functions with the actual addresses these variables or functions hold on a

[2]Linux ELF could be interpreted in two different ways: by means of sections or by means of segments. Segments are groups of sections, which share the same attributes as regards their memory destination. The information about segments are contained in a **program header**. Both segments and **program header** are commonly used in Linux ELF, but are not present in those dedicated to MSP-430/MantisOS platforms, which require a section-based linking process, because of the need for absolute address resolution.

`typedef struct {`		
`Elf32_Word`	`sh_name;`	index of the `.shstrtab` storing the name of this section.
`Elf32_Word`	`sh_type;`	section's contents and semantics.
`Elf32_Word`	`sh_flags;`	flags that describe miscellaneous attributes.
`ELf32_Addr`	`sh_addr;`	address at which the section's first byte should reside.
`Elf32_Off`	`sh_offset;`	offset from the beginning of the ELF to this section.
`Elf32_Word`	`sh_size;`	section's size in bytes.
`Elf32_Word`	`sh_link;`	section header table index link.
`Elf32_Word`	`sh_info;`	extra information.
`Elf32_Word`	`sh_addralign;`	address alignment constraint, if needed.
`Elf32_Word`	`sh_entsize;`	size in bytes of each entry, for fixed-size sections.
`}Elf32_Shdr;`		

Table 3.3: Structure of section header

specific machine. Once linked and relocated, this section has to be loaded in the FLASH memory of a mote.

.rodata In this section are stored the read-only constants, such as strings or constant variables. Sometimes, the ELFs generated by some MSP-430 compilers don't include this section, because of constant strings and variables are stored directly into `.text`. As the previous, also this section has to be loaded in FLASH memory.

.data This section contains both global and static variables which are initialized and modifiable. It has to be loaded in the RAM of a mote.

.bss This is an empty-section, whose only significant parameter is its size. It represents the amount of blank memory-space needed for storing non-initialized variables and data. The par of the RAM destined to store the `.bss` is zero-initialized when the program is loaded.

Some other sections contain auxiliary information used by the linker to perform its relocations. A standard Linux ELF holds a lot of support sections, but in dealing with our platform only a small set of them are actually used:

.rela.text, .rela.rodata, .rela.data These are the relocation sections for the three core sections above described (Table 3.4). The name ”‘rela”’ stands for ”‘relocation absolute”’, because of our platform uses absolute addresses and does not hold a memory management unit supporting a virtual address space (standard Linux ELF has ”‘rel”’ sections, i.e. relocation sections for virtual address spaces).

`typedef struct {`		
`Elf32_Addr`	`r_offset;`	offset in the relative section of the entry to be relocated.
`Elf32_Word`	`r_info;`	symbol table’s reference and type for the relocation.
`ELf32_Sword`	`r_addend;`	constant addend to be added to the relocated field.
`}Elf32_Rela;`		

Table 3.4: Structure of relas entries

.strtab This section stores the names (strings) of the symbols contained in the symbol table.

.shstrtab This section contains the names (strings) of the other sections and is used to resolve the entries of the section header.

.symtab This section contains the names and relative addresses of all static functions and static variables (Table 3.5). It is the internal symbol table, so the references points to objects contained in the ELF itself. To resolve references to non-static functions and variables another symbol table is used: the global symbol table. The global symbol table resides in the operating system and collects all the couple `<name, address>` of global variables and functions belonging to OS libraries, shared libraries etc.

This brief synopsis of the ELF’s general structure provides the basic information for understanding the linking process following described. It is worth noting that an ELF has indeed a very complex structure, if studied in its whole paradigm; we remind to [60] and [61] for further insights.

`typedef struct {`		
`Elf32_Word`	`st_name;`	offset in the `.strtab` of the relative symbol name.
`Elf32_Addr`	`st_value;`	actual address of the symbol
`ELf32_Word`	`st_size;`	symbol size.
`unsigned char`	`st_info;`	symbol type and binding.
`unsigned char`	`st_other;`	symbol visibility.
`ELf32_Section`	`st_shndx;`	section index.
`}Elf32_Sym;`		

Table 3.5: Structure of symbol table entries

3.5 Project and Implementation

The creation of a small loadable module from an original ELF is the idea underlaying the development of a dynamic linking and loading mechanism for a resource-constrained environment. An ELF file, in fact, is much larger –on average– than the available memory on a TelosB. To this purpose we have conceived a compact format, called WLF (WASP Loadable Format), which just contains the relocated executable to be loaded on a node, along with the essential information to parse and extract the various executable sections.

The way through which the linking and relocation process has been made dynamic, consists in downloading the Global Symbol Table from a node and to perform an in-Host relocation of a source ELF according to the actual addresses reported in this symbol table, which contains the symbol names and the pointer to the actual addresses of these symbols in the node's image. This fact, along with the use of absolute address in Mantis TelosB environment, gave us the chance to link the executable directly on the Host, in a first phase, and load it on nodes at an arbitrary later moment, by means of small, ready-to-load WLF modules. The above mentioned issues, along with other considerations that have driven to the definition of our model, are following summarized:

1. An ELF is composed by a lot of tables and sections. Most of these sections are not useful in terms of execution code , but they have sense only in the relocation process. Several tables, furthermore, are completely unused, also

while relocating the code; stripping the ELF file from all the useless sections in order to obtain a compact loadable format has been the first target of the project;

2. To relocate the code of an ELF file, the nodes' Global Symbol Tables are necessary. This tables contain all symbols (names of variables and functions) of the system, associated to their addresses. In the case of MantisOS, mounted on TelosB platform, the addresses are absolute and don't change during the node lifetime; once proved the invariance of absolute addresses in time, we have developed the idea of downloading the Global symbol table from node to PC and to use it's addresses to resolve symbols an relocate code in Host. The possibility of performing the linking phase on the Host-side has represented the main breakthrough of generating a small ready-to-load format, called WLF;

3. There are some cases in which is not necessary to relocate the code using local and global symbols. These cases are related mainly with programs that don't use any global variables or any system functions. For these simple programs, it suffices to extrapolate from an ELF only the `.text`, `.rodata`, `.data` and `.bss` sections, without relocating them;

4. At a node level, the prospective changes from linking to loading phase. The pre-linked binary contained in WLF is sent from host to node. The server program, running on node, receives the code and store `.text` and `.rodata` segments in FLASH ROM, while `.data` and `.bss` are allocated in RAM.

3.5.1 Retrieving global symbol table from node

The fundamental premise as well as the core feature making the linking process "'*dynamic*'" bears on the idea of providing the operating system on the node with a global symbol table. Each entry of the symbol table contains the names of the various symbols –functions and variables– along with the absolute address of these objects in the node's image (Table 3.6). The whole symbol table is configured as an array of these entries, containing all the functions and global variables of the system, and is initialized when the OS' image is firstly loaded on a node.

```
typedef struct {
const char*       name;
const char*       value;
}symbol;
```

Table 3.6: Structure of Nodes' symbol table entries

The symbol table resides on the node and is downloaded by the Host when a new program must be linked and relocated in a small module, to be loaded on that node. To assure this functionalities the nodes have been provided with a *loading server*, a branch of which implements the protocol to send the symbol table to the requiring host. The host, on the other side, has been developed with a method *symTabGetter*, that communicates to the node the intention of retrieving symbol table and initializes the connection to download it. To do this job, a specific application-level protocol has been configured (Figure 3.5).

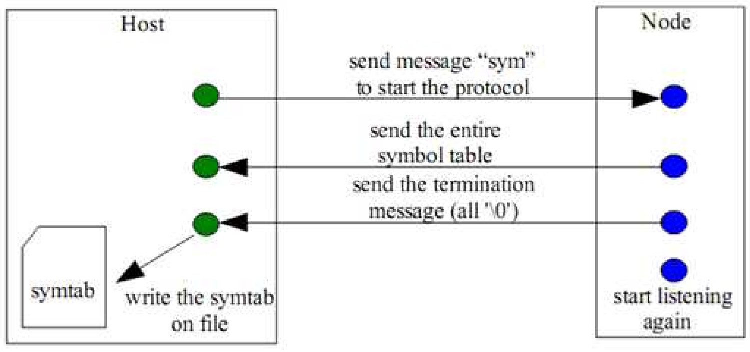

Host
Node
send message "sym" to start the protocol
send the entire symbol table
send the termination message (all '\0')
symtab
write the symtab on file
start listening again

Figure 3.5: Retrieving the symbol table from a node

3.5.2 ELF stripping

The implementation choice has been to strip the ELF file from all the sections that don't contribute to the executable. Only the relevant sections have been considered, and in particular the `.text`, `.rodata`, `.data` and `.bss` sections. Along with these four core sections, also the `.rela.*` sections, the local symbol table `.symtab` and the string table `.strtab` are extracted. This process is reported in Figure3.6.

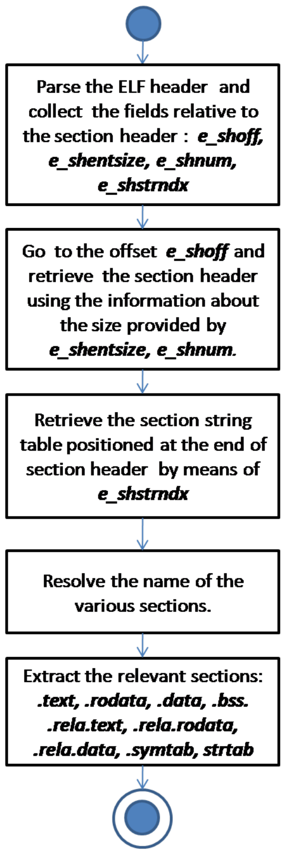

Parse the ELF header and collect the fields relative to the section header : e_shoff, e_shentsize, e_shnum, e_shstrndx
Go to the offset e_shoff and retrieve the section header using the information about the size provided by e_shentsize, e_shnum.
Retrieve the section string table positioned at the end of section header by means of e_shstrndx
Resolve the name of the various sections.
Extract the relevant sections: .text, .rodata, .data, .bss. .rela.text, .rela.rodata, .rela.data, .symtab, strtab

Figure 3.6: ELF parsing and sections extraction

For the first three sections, the relocation of addresses has been projected to be performed on host-side, once the global symbol table has been downloaded from node. The `.bss` segment is empty and represents the space that must be allocated on target system in order to contain the non-static, zero-inizialized variables.

3.5.3 Full and simplified linking processes

The complete linking process takes as input the sections retrieved through the previous stripping process, along with the global symbol table downloaded from the node as described in Section3.5.1.

With all this information, the linking and relocation phases are executed and all the undefined references are resolved and updated. This process is described in Figure 3.7.

The relocated sections are then stored in a small ready-to-load WLF module.

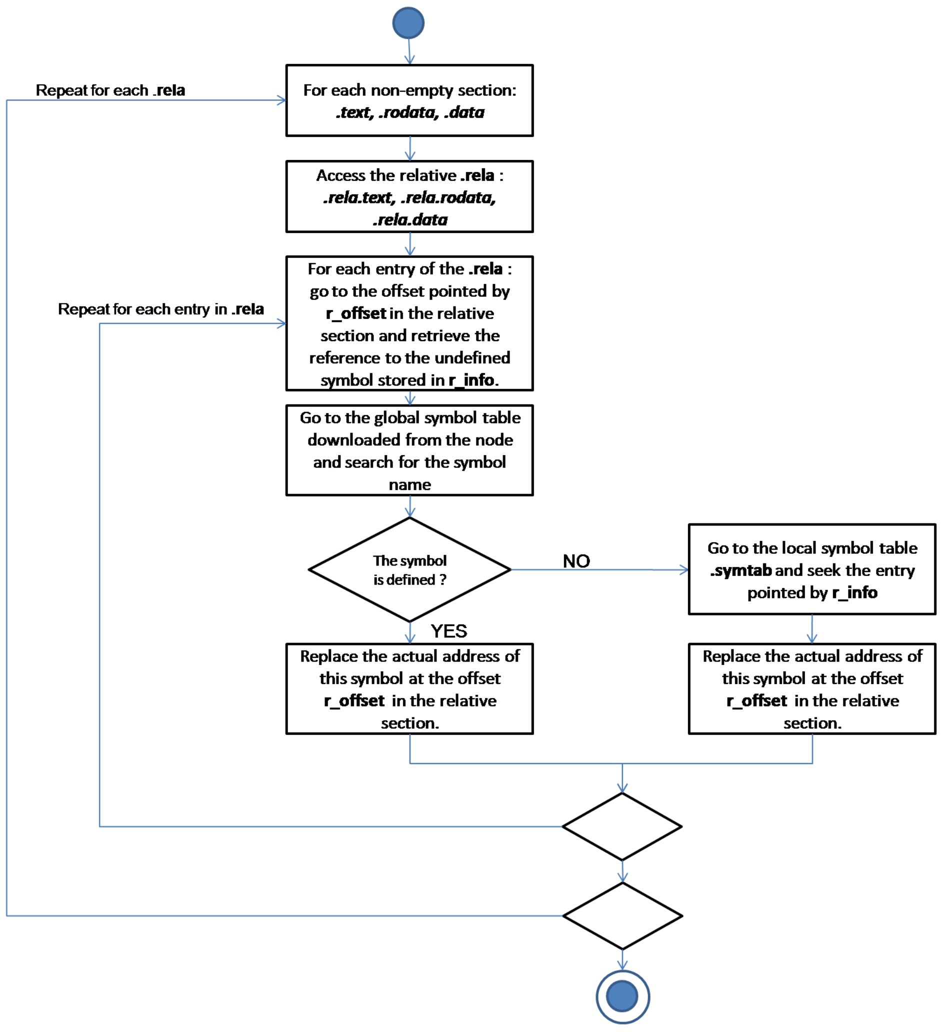

Repeat for each .rela
For each non-empty section: .text, .rodata, .data
Access the relative .rela : .rela.text, .rela.rodata, .rela.data
Repeat for each entry in .rela
For each entry of the .rela : go to the offset pointed by r_offset in the relative section and retrieve the reference to the undefined symbol stored in r_info.
Go to the global symbol table downloaded from the node and search for the symbol name
The symbol is defined ?
NO
Go to the local symbol table .symtab and seek the entry pointed by r_info
YES
Replace the actual address of this symbol at the offset r_offset in the relative section.
Replace the actual address of this symbol at the offset r_offset in the relative section.

Figure 3.7: Linking and relocation

The WLF header (Table 3.7) accomplishes two important functions:

- Store the section-size information, that will be used to correctly parse these sections from the WLF module. Once known the sections starting points, i.e. the end of WLF header (which has a fixed 11 bytes size), and their sizes, the executable sections can be easily loaded in FLASH and RAM;

- while loading, the size information are transmitted to the node, which will provide to allocate the correct amount of memory space for each segment. The strong size reduction characterizing a WLF as regards the original ELF offers the possibility to load at run-time any WLF module with a very small amount of required energy both for radio transmissions and processing time.

The overall process is summarized in Figure 3.8.

For simple programs, that don't call any system function or global variable, the relocation of symbols is not necessary. So a "light linking " mechanism has been implemented on a simplified variant of the schema described above. The main difference lies on the fact that "light linking" procedure doesn't perform any relocation action, but simply extract the four relevant segments (`.text`, `.rodata`, `.data` and `.bss`) from original ELF and put them "brute force" into the WLF file (Figure 3.9).

```
typedef unsigned int wlf_off;
typedef unsigned int wlf_size;
typedef struct {
char wlf_id[3];
wlf_size text_size;
wlf_size rodata_size;
wlf_size data_size;
wlf_size bss_size;
}wlf_header;
```

Table 3.7: WLF header

3.5.4 Load WLF on the node

To load the WLF binary on the node, we have provided the node's server with a function that retrieve the information about segments from the host, then allocate the correspondent space on RAM and ROM and finally receive the segments and store them. In enabling this mechanism, we have developed a dedicated application level protocol whose flow is shown in Figure 3.10.

3.5.5 Design choices

During the project, many problems raised, most of them related to the scarce support offered by the operating system and to the lack of documentation.

- Programming language choice: in developing the host-side application, we faced a trade off between the will of building a graphical user interface and working in a comfortable programming environment. The first idea has been to use Java and its graphical libraries. This path has been abandoned because Java doesn't support unsigned types. After a deep analysis, the choose is fallen on C++ language and its QT4 libraries, which though a more complex framework, offers the possibility of using unsigned types,

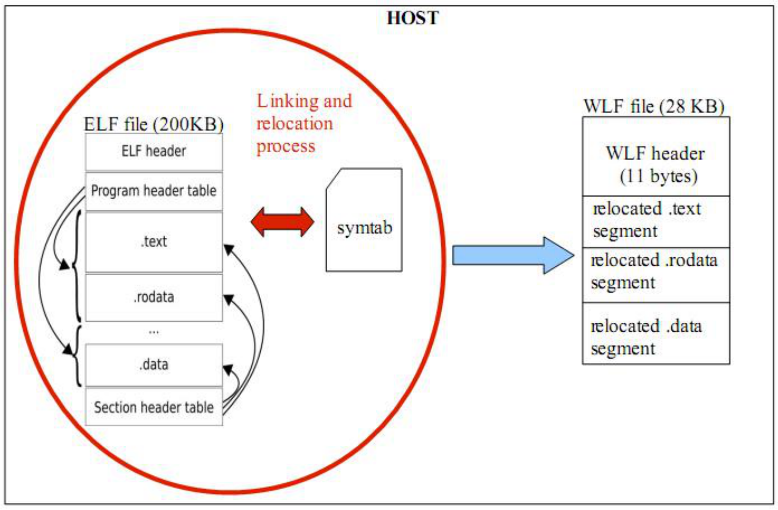


Figure 3.8: From ELF to WLF file format

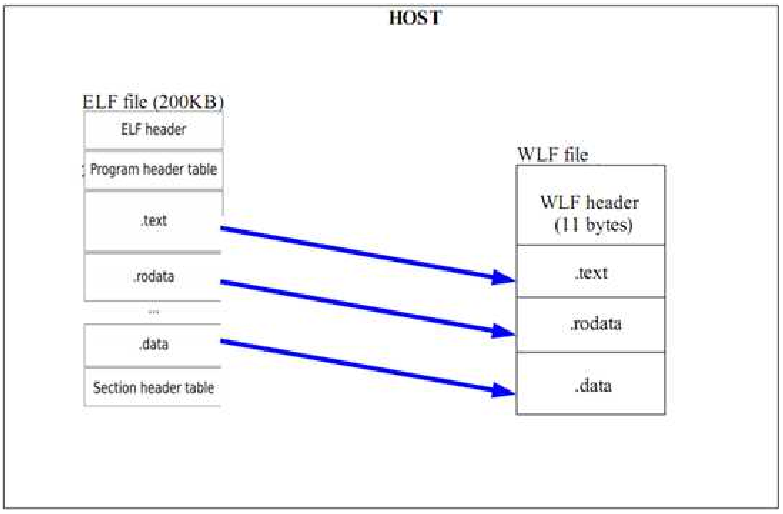


Figure 3.9: Schema of the light linking process

that well perform in dealing with absolute addresses.

- Software integration: the way that have brought us to the implementation of a common GUI for both the linking/loading mechanism and the Mantis toolchain has presented a problem related to software fragmentation. The Mantis toolchain, in fact, is constituted by code written in different programming languages. The bootstrap loader is a Python application, while all the others functionalities are C programs. To integrate the C++ GUI with the Python bootstrap loader, we have conceived a solution based on a smart use of linux system call, QT4 interfaces and linux pipe mechanism, to redirect output from the ”standard output” to the GUI’spanel:

  1. the bsl.py (bootstrap loader) application is called through the system call, activated when a specific button is pushed in GUI panel;
  2. the bsl.py does its jobs and print the results on standard output, opportunely redirected to a ”bridge file” placed in the dynamic linker Binary folder;
  3. through QT4 interfaces the content of bridge file is read and printed into the GUI’s panel, in a totally transparent way.

Host	**Tx/Rx**	**Node**
Begin		
Send message `wlf`	$\rightarrow$	Receive
Receive	$\leftarrow$	Send ack `wlf`
Send `textsize`	$\rightarrow$	Receive
Receive	$\leftarrow$	Send `.text` base address
Send `rodatasize`	$\rightarrow$	Receive
Receive	$\leftarrow$	Send `.rodata` base address
Send `datasize`	$\rightarrow$	Receive
		Allocate RAM
Receive	$\leftarrow$	Send `.data` base address
Send `bsssize`	$\rightarrow$	Receive
		Allocate RAM
		Clear RAM
Receive	$\leftarrow$	Send `.bss` base address
Send `.text`	$\rightarrow$	Receive
		Copy to Flash
Receive	$\leftarrow$	Send ack
Send `.rodata`	$\rightarrow$	Receive
		Copy to Flash
Receive	$\leftarrow$	Send ack
Send `.data`	$\rightarrow$	Receive
		Copy to RAM
Receive	$\leftarrow$	Send ack
End		

Figure 3.10: WLF transmission and loading process.

- Software version management: Mantis presents a very fragmented folder tree and both sources and binary are now hard to manage. To simplify and clarify the project process on Mantis environment an SVN version for the project has been created as well as a group under GOOGLE CODE domain.

3.5.6 Architecture diagrams

Figure 3.11 and Figure 3.12 show, respectively, the class diagram of the in-host linker and the structure of the in-node server-process.

It is worth noting that the host application is devoted to retrieve the global symbol table from the node, perform the relocation and linking actions and create WLF modules. Along with these functionalities, the host maintains a repository of the symbol tables downloaded from the nodes. It has been observed, in fact, that the entries of the global symbol table don't encounter significant variations for application neighboring in time. So there is no need for a new symbol table download if further applications must be linked for a node, whose symbol table has been recently downloaded. Moreover, in storing the various symbol tables, it

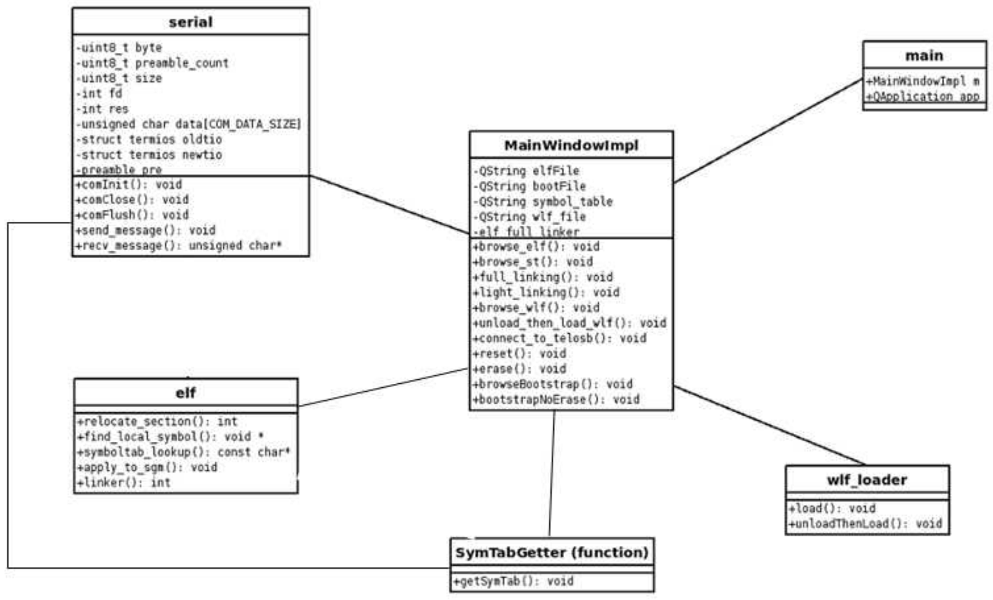


Figure 3.11: Class diagram of the WASP dynamic linker - host side

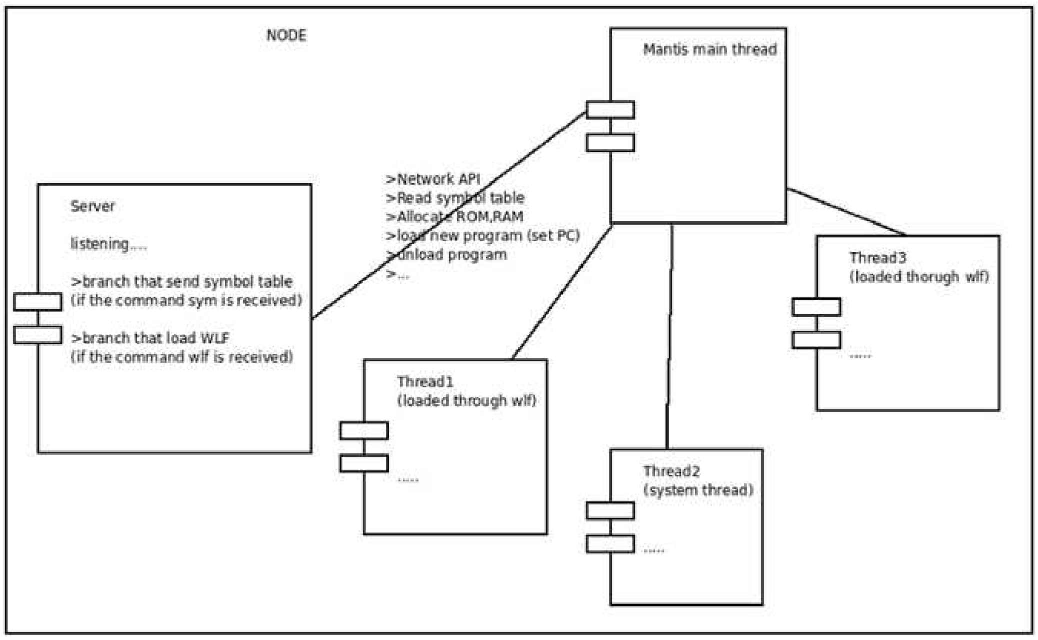


Figure 3.12: Architecture of the server process on a node - node side

is possible to implement a diff-based uploading which only register the variations without requiring the retrieval of the whole table. These optimization are actually being investigated and further details are reported in Section 5.2.1.

3.6 Graphical User Interface

The application is composed of three panels and a common display for reading the output of each operation. The three screens regards respectively: the linker functionality, the WLF loader and the Mantis toolchain.

3.6.1 Linking Functionalities (Figure3.13)

- Feature 1: shared screen on which is shown the output of each operation.
- Feature 2: here are offered the functionalities to choose an ELF file and the destination name of the WLF, that will be created after the linker starts.

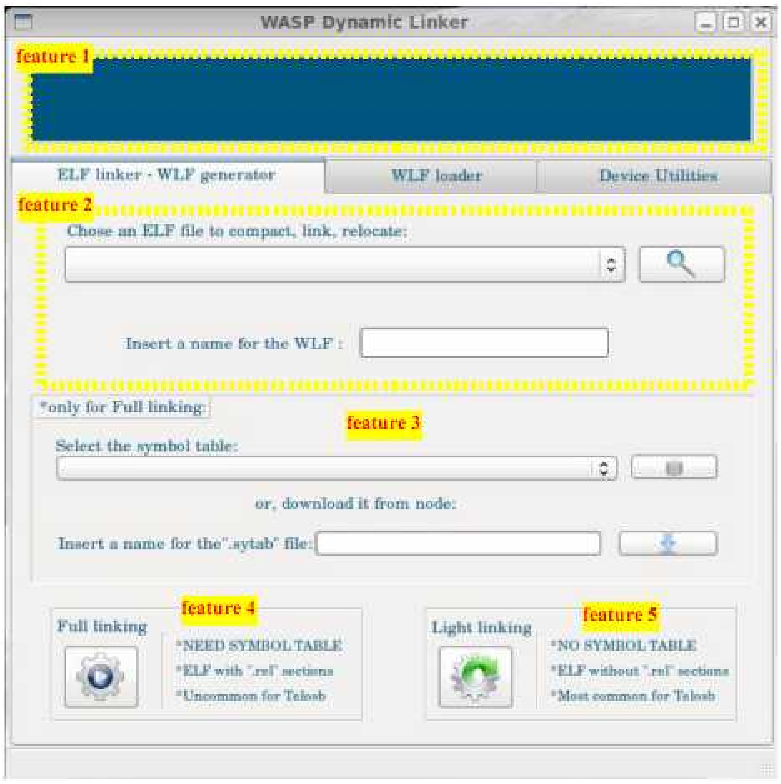


Figure 3.13: Panel of the G.U.I. application for the linking functionalities

- Feature 3: if a full linking is going to be performed, the user must select a symbol table using the relative file browser. The default start folder of this file browser is symbol table database. If the symbol table for the node is not already been downloaded, the user must download it from the node specifying a name for the destination ".sytab" file and then pushing the button with the down oriented arrow.

- Feature 4: button that perform a full linking (a symbol table must be selected). The resulting WLF is stored in WLF database.

- Feature 5: button that perform a light linking (no symbol table is needed).The resulting WLF is stored in WLF database.

REMARK: all the actions are guided by messages in global screen, if wrong.

3.6.2 WLF loader (Figure3.14)

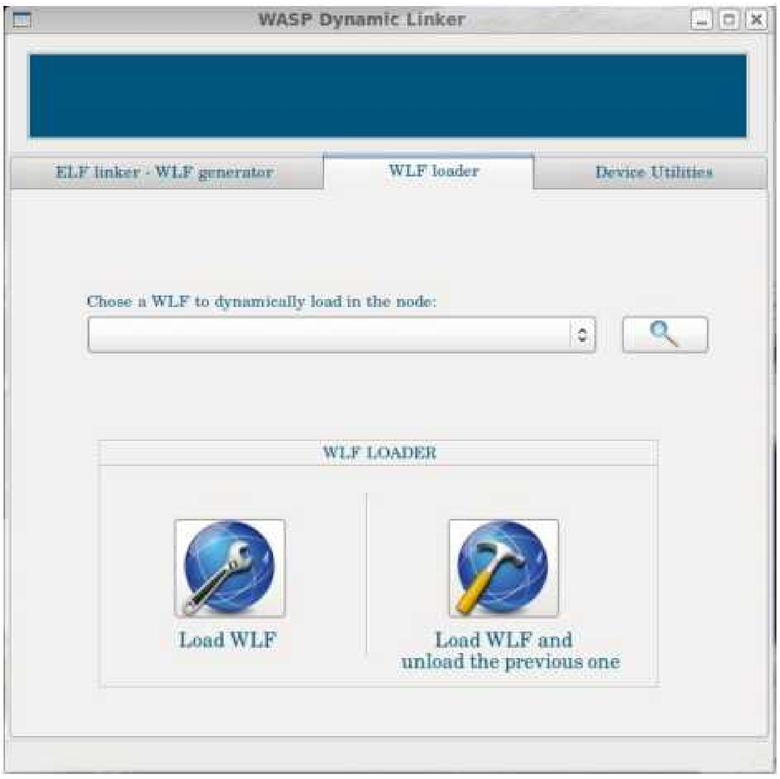

WASP Dynamic Linker
ELF linker - WLF generator
WLF loader
Device Utilities
Chose a WLF to dynamically load in the node:
WLF LOADER
Load WLF
Load WLF and unload the previous one

Figure 3.14: The WLF loader panel

- File browser: using the file browser, the user can select a WLF to load into node. The default starting folder is WLF database.
- Load WLF button: if pushed, initializes the dialog with node and load a WLF. This functionality is still in developing (because of Mantis flash problems) and is object of work by Luigi Rucco and Simone Corbetta.
- Load WLF and unload the previous one button: this functionality is for future use, it's predisposed but not yet implemented because of the lack of a Flash manager.

REMARK: all the actions are guided by messages in global screen, if wrong.

3.6.3 Mantis' Toolchain (Figure3.15)

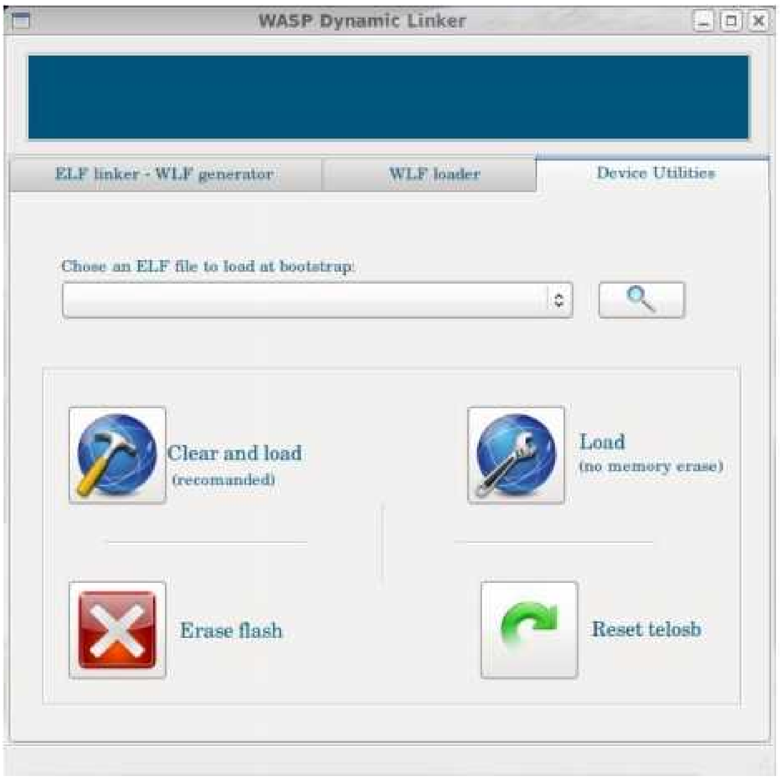


Figure 3.15: The Mantis Toolchain panel

- Erase flash: this button, if pushed, invokes the "Mantis' bootstrap-loader" function that completely erase the Telosb flash. The output of the operation, such as all the other operations' output is reported in the global screen window on the top of the GUI.
- Reset telosb: this button executes the "Mantis' bootstrap-loader" reset of the node. This operation does not change the memory state of the node, so the Global symbol table remains the same.
- Clear and load: before launching this function is necessary to select an ELF file, by means of the file browser placed above. Once selected an ELF, this command launches the "Mantis' bootstrap-loader" erase-and-load

functionality. This operation in a first moment erases the telosb flash and then loads the executable contained into the ELF. Because of a new program will be loaded in a completely erased memory, this operation will change the global symbol table, also if the same ELF is loaded again.

- Load: it performs as the previous command, with the exception that the memory is assumed to be cleared by a previous ”Erase flash” invocation. A previous ELF selection is needed.

REMARK: all the actions are guided by messages in global screen, if wrong.

3.7 Evaluation of the Linker/Loader

Following we report the results obtained during the experiments, logically divided according to the two macro-phases: the first performed at host level that targets the creation of a WLF loadable format starting from a source ELF file. The foremost gain obtained downstream this macro-phase is the strong reduction in the size of the module to be transmitted and the related benefits in terms of radio transmissions and memory requirements (Section 3.7.1).

The second macro-phase is processed at node level and embodies the operations needed for loading the WLF module appositely created by the host. The achievements in this subsequent step consist of a strong optimization of the energy required to accomplish the loading phase, thanks to the small byte-size of the modules and their just-relocated code, which doesn't need any further relocation (Section 3.7.2).

3.7.1 Size reduction

The first macro-phase (composed by three sub-phases described in Sections 3.5.1, 3.5.2 and 3.5.3) of the proposed technique leads to the creation of the WLF format. The benchmark for assessing the effectiveness of the implemented toolchain is the reduction obtained on the size of the loadable format from the original ELF. In performing this analysis we assume all the necessary information (symbol table, base addresses, etc.) already being retrieved.

The remote linker, operating on the host, takes in input the ELF file, the node's global symbol table, the base address in the RAM memory (for the relocation of the `.data` section) and the base address in the Flash memory (for the relocation of the `.rodata` and `.text` sections), performs both linking and relocation actions and finally generates a WLF module, ready to be loaded on the destination node.

We have tested this process on eight different ELF files, in order to produce the relative WLF modules. Those ELFs are characterized by different parameters for what concerns lines of code, which range floats from 15 CL to 400 CL, and number of libraries.Results by these instances have been reported in Table 3.8 and, as could be noticed, the size reduction oscillates from a minimum of 86% to a maximum of 99%, with an average value of 96.06%. These differences are caused by differences in the magnitude of `.data` and `.rodata` sections of different ELFs, but the overall ratio of compression remains extremely promising and shows a value of about 1:25.

The fluctuation entailed by the dimension of the `.rodata` section comes out remarkably in the second of the functions used for the benchmark in Table 3.8: the `printf()` refers to a particular variant of the common C-Language `printf()`, suited for the MantisOS environment in order to transmit by radio a certain amount of data toward the host in the form of strings. In the considered example the 85% of the overall byte-size, i.e. 2300 bytes, are expressly dedicated to store the constant strings in section `.rodata`, whereas only 400 bytes are dedicated for storing machine code.

3.7.2 Performance

The second macro-phase (composed by a unique great phase described in Section 3.5.4)of the procedure consists of transmitting the WLF module to the node and loading sectiorn `.text` and `.rodata` in flash and `.data` in main memory, where it is also allocated blank space for the correspondent dimension of the

Benchmark	Size (LoC)	ELF (bytes)	WLF (bytes)	Compression
blink	49	17216	132	99.23%
printf	76	20408	2742	86.56%
flash	122	20484	646	96.85%
leds	50	17063	322	98.11%
usb	68	19432	284	98.54%
crc	59	3436	140	95.93%
queue	109	17204	330	98.08%
tree	425	24900	1206	95.16%
Average				96.06%

Table 3.8: Size reduction from ELF to WLF.

`.bss` section. As advanced in the introduction to Section 3.7 this macro-phase determines the core performances of the network's nodes in terms of energy consumption, so the analysis will be deepened to estimate with the highest degree of accuracy the relative results.

To make the energy consumption assessment as precise as possible we have based on the most complete energy-characterization already available for TelosB nodes, which have been used during the experiments. This characterization has been formulated in a research from Berkeley University [1] and is reported in Table 3.9.

These measures have been acquired through high-precision oscilloscopes, provided by the Intel Lab (Berkeley) and present the values of current absorption calculated at 1.8V power supply: starting form these values, the energy consumption has been obtained by measuring the time intervals of the different stages through which a node passes while loading the various WLF modules used as benchmark.

Let's call E the total energy consumption for this phase, then it could be logically regarded as a composition of two contributions: E_c as the energy required for radio communications and E_p as the energy absorbed by the microcontroller for supporting radio operations, allocating memory and storing the various sec-

Node state	**Current**	**Symbol**
Standby (RTC on)	5.1 μA	
MCU Idle (DCO on)	54.5 μA	
MCU Active	1.8 mA	$I_{\mu C}$
Radio RX	20.0 mA	I_{rx}
Radio TX (0dBm)	17.7 mA	I_{tx}
Flash Read	2.3 mA	
Flash Write	13.3 mA	

Table 3.9: TelosB energy characterization [1]

tions of a WLF in the relative allotted spaces.The following equations state the way in which these two energy contributions have been determined:

$$E_c = \left[\frac{P_{rx} \cdot S_p}{B} \cdot I_{rx} + \frac{P_{tx} \cdot S_p}{B} \cdot I_{tx}\right] \cdot V_{dd} \tag{3.1}$$

$$E_p = N_p \cdot T_{ck} \cdot I_{\mu C} \cdot V_{dd} \tag{3.2}$$

where P_{rx} and P_{tx} are the number of packets to be transmitted and received, S_p represents the byte-size of a single packet, B is the bandwidth associated to the radio transceiver, V_{dd} is the power supply (1.8V), N_p is the number of clock cycles associated to both copying and cleaning memory actions, T_{ck} (precisely 125ns) is the period for a complete clock cycle of the microcontroller and I_{rx}, I_{tx} and $I_{\mu C}$ the three average values of current absorption as follows from Table 3.9.

As it has been formulated, this model constitutes the core bearing for estimating the energy actually absorbed by a node for loading a benchmarked set of WLF modules, expressly selected according to their meaningfulness among a broader repository used for the experiments. The detailed results for a representative module in the sample-set can be found in Table 3.10, where each sub-phase involved in the loading protocol (see Figure 3.10) is associated to the relative energy consumption cost. Along with these energy costs, there is another fundamental contribution that must be considered upstream, that is the energy required for retrieving the symbol table from the node before generating the WLF itself. The size of the symbol table used for the test-set amounts to 4400 bytes and approximately re-

Phase	M	P	$E_c(\mu\text{J})$	$E_p(\mu\text{J})$	$E(\mu\text{J})$
Receive WLF	3	2	220.2	21.0	241.2
Receive sizes	8	8	881.2	84.0	964.8
Allocate RAM for bss[1]	59		0.0	0.0	0.0
Clear RAM for bss	59		0.0	0.1	0.1
Receive .text	1503	25	2906.6	262.8	3169.4
Copy .text to Flash	1503		0.0	8.5	8.5
Receive .rodata	0	1	103.4	10.5	113.9
Copy .rodata to Flash	0		0.0	0.0	0.0
Receive .data	102	3	337.0	31.5	368.5
Copy .data to RAM	102		0.0	0.5	0.5
Total					**4866.9**

[1] This contribution is hard to determine due to the short time taken to complete. Its energy is thus negligible.

Table 3.10: WLF link & load protocol energy characterization.

quires 8.9mJ to be retrieved, according to the present implementation of the node server, which transmits the entire symbol table in a continuous stream of bytes, using packets of 64 bytes. Clearly, this symbol table should be retrieved only once for experiments brought closer in time, storing its copy locally to the Host. Possible optimizations to avoid retrieving the whole symbol table, synchronizing the local copy with the variations occurred on the nodes at run-time, are presented in Section 5.2.1.

The experiments have been iterated according to the schema presented in Table 3.10 for each module belonging to the sample-set, whose aggregate values are reported in Table 3.11. In this table are listed the values E_c and E_p referencing the respective contributions for radio transmissions and microprocessor effort. It's worthy to notice the presence of a constant factor affecting the E_c terms, because of protocol overheads mainly related to the exchange of acknowledgment packets between node and host.This constant factor amounts to 640.6 μJ and is equal for all the benchmarked modules.

In deeply analyzing the results obtained on the overall sample, it could be

Benchmark	E_c(μJ)	E_p(μJ)	E(μJ)
blink	1267.2	0.544	1267.7
printf	4937.4	13.229	4950.6
flash	1983.3	3.159	1986.5
leds	1535.7	1.468	1537.2
usb	1535.7	1.286	1537.0
crc	1267.2	0.583	1267.8
queue	1535.7	1.507	1537.2
tree	2789.0	5.764	2794.7

Table 3.11: Energy consumption for dynamic linking.

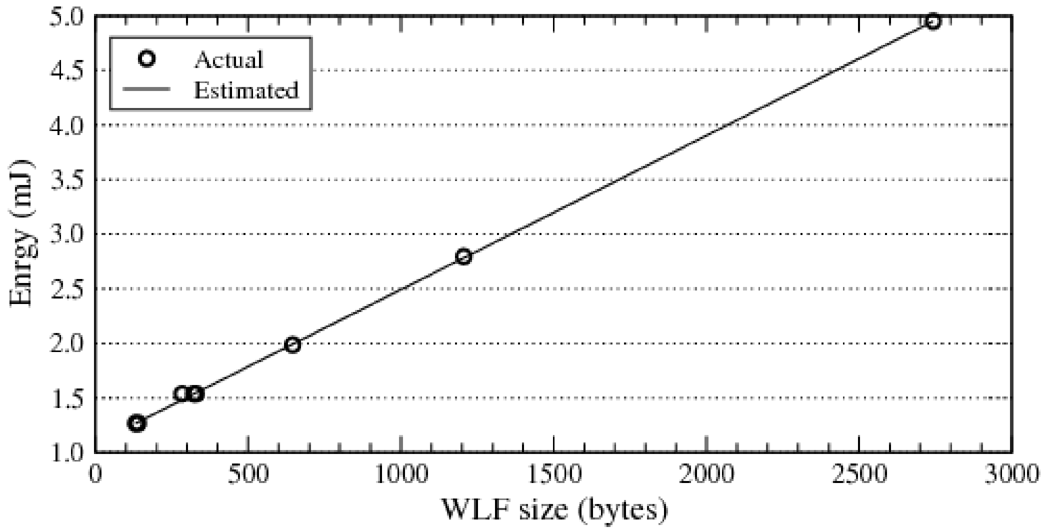


Figure 3.16: Energy consumption as a function of WLF size.

noticed a certain regularity among results, entailed by the very limited amount of energy consumed in microcontroller operations and the structure of Equation (3.1) and (3.2): let's now apply a linear regression to the results in Table 3.11, such that:

$$E = 1.411 \cdot S_{wlf} + 1081.5 \tag{3.3}$$

where S_{wlf}, expressed in bytes, is the size of the WLF and both the total energy and its two components are expressed in μJ. This equation provides a simplified view of the model, which demonstrates a great adherence to the results obtained through the detailed analysis above reported.

The trend pointed out after the analysis of results by Equation (3.3) is shown in Figure 3.16: it could be observed the high level of accuracy provided by this coarse-grained abstraction by comparing its results with those shown in Table 3.11 and the relative sizes reported in Table 3.8.

3.8 Some considerations

The work we have done has required a great effort in studying the dynamic reprogramming issue and conceiving a suitable solution.Many topics have been considered, starting from the linking processes to the ELF structure and optimizations. In this period many solutions have been tested, such as system call and basic loading mechanisms, but at the end the more complete and potentially improvable system has been chosen. The dynamic linking in a resource-constrained environment has always been considered as an heavy solution, because of the large byte dimension of ELF files. Our propose represent the very first light approach to the dynamic reprogramming by means of small pre-linked loadable modules. This is an innovative approach that promises scalability and flexibility in sensor network dynamic reprogramming, along with a broad variety of loadable applications and common Linux developing environment.
Following we present some possible improvements and optimizations that are foreseen to be applied at the present version.

Chapter 4

A Genetic Model for functional allocation and lifetime maximization

Leafing through the literature regarding Wireless Sensor Networks' performances, most of the effort could be found on the assessment and improvement of networking capabilities, which have been ever considered preeminent both for the overall efficiency and the power-consumption retaining of the network. This was rightly considered as the main priority during an early stage of this Technology, since the architectural design was far to be settled in adequate paradigms, capable of providing good coverage as well as safe transmissions.

Nowadays WSNs have reached a quite good degree of maturity and many studies for defining actual applications of this technology both for civil and military purposes have recently been undertaken by the foremost academic institutions all around the world. That has caused not only the networking and hardware but also the applicative domain to arise as a fundamental field with which researchers have to confront their work and their investigations. The reference system, that delineates both goals and methods, orbits the Power-consumption issue and all the possible solutions allowing its reduction, while preserving functional effectiveness.

In this new phase, it seems perfectly clear that a WSN turns out to be useful for a certain task not only if it is energy-efficient enough, but also if it is able to preserve its energy efficiency while accomplishing a well-defined applicative workload, for which that network has actually been deployed. Thus, the focus is now broadening to encompass also applicative aspects and particularly the possibility of providing the nodes with all the functions needed for monitoring the target phenomena, preserving, at the same time, as much power as possible to enforce a longer lifetime. This topic leads to a trade-off between the need for functional completeness and effectiveness on one hand and memory, energy and lifetime constraints on the other.

The importance of providing a Wireless Sensor Network with a mechanism for dynamic reprogramming its nodes is obvious as regards the possibility of propagating updates, fixing bugs and adding new functionalities without withdrawing the nodes from their displacements. That turns out particularly evident when the network is composed by a huge number of nodes. Moreover, among the various reprogramming mechanisms, the advantages offered by approaches like a dynamic linker or a pre-linked based technique overtake all the others because they don't require to refresh the entire image of the nodes, with consequent gain in terms of radio transmissions (it is not necessary to forward an heavy new image) and status maintenance.

These characteristics have ever been outlined as essential in literature (see Table 1.2) and the conceptual framework here presented has been expressly designed to give to this assumption a concrete verification, working on resource-constrained environments. Moreover, this model aims to provide the designers with a powerful tool for configuring and optimizing the Functional Domain of a network, organizing and balancing the workload on nodes in terms of allocated functions, in order to maximize the lifetime and overcome memory boundaries, while preserving the completeness and an opportune redundancy of the function-set.

Though the model has been conceived general enough to cover the topic of the Dynamic Reprogramming in its whole nature, i.e. regardless of the specific dynamic reprogramming approach, the experiments has been focused on the dynamic-linker we have created, whose costs, in terms of power requirement, have been utilized for the model test-set described in Section 4.4.

The level of abstraction has been set such that both the topological and functional facets of a WSN are actually encompassed in the analysis. The functional view has been designed on the notion of task, i.e. a set of functions that must be executed to control a given phenomenon. These functions could be either statically or dynamically assigned to the nodes and the aim is to pursue an optimal allocation of these functions for maximizing the overall lifetime of the network.

The topological view, on the other hand, draws on the concept of network-clustering, at the base of the well known *coverage problem*. The constraints, as well as the fitness function, have been designed as continue functions, in order to take advantage of the peculiarities of the Genetic Algorithm (GA).

Results obtained running the GA on the model demonstrate great accuracy and point out a promising perspective for a systemic adoption of this technique in configuring efficient WSNs'applicative domains.

4.1 Related Work

Models to define and eventually optimize the functional configuration of a network come as a natural consequence of the advancement in WSNs studies. Actually, models such the one here proposed are a cutting-edge novelty in a context sill oriented toward networking concerns, but, in all likelihood, they are destined to get a great foothold as far as WSN applications will creep in to real business.

A slightly comparable, though very different in substance, effort has been spent in building simulation tools, most of them designed to provide support in estimating:

a) specific Operating System performances (e.g. [62, 63]);

b) protocol-level behaviors (e.g. [64–66]);

c) instruction-level/clock-dependent interactions (e.g. [67, 68]);

d) fault-sensitive software testing (e.g. [69]);

e) power-consumption (e.g. [70, 71];

f) mixed applicative and physical layers evolution (e.g. [72, 73]).

In addition to those reported, many other simulation tools actually exist for WSN specific environments, even though no multi-domain models have already been proposed [74];some experiments in this direction have been undertaken in trying to combine different simulators in a common platform to concur at a final multi-layered comparison [75].

As countered, simulators differ from our model both in scope and purposes: a simulation tool, in fact, is designed to provide a coherent representation of the evolution of a system, given some parameters to be monitored. On opposite, our goal is to create an abstraction of the network applicative domain, in order to perform a global optimization of the SW configuration. This configuration should maximize the lifetime and preserve both functional and non-functional requirements.

The only point of contact between these two approaches is the need for a model that describes not only the HW and networking features, but also the functional aspects of a WSN. In that perspective, simulation tools come about to be used in a consequent verification step, to be accomplished after our model-based optimization to validate its results.

A similar consideration could be stated about the Genetic Approach: in literature could be found some examples of genetic algorithms fitted for tackling the problem of area-coverage and clustering (e.g. [76–79]), but, as the definition suggests, these problems only target the network layer of a WSN. In these works only the topology and the radio efficiency are taken into account, while none of the applicative facets is covered. Once again, these variants of the genetic algorithm are not comparable with our model, but they are complementary indeed. In fact, the coverage analysis of the network places up-stream as regards our model, which takes as an input-parameter the clusters identified in this phase.

After this outlook on the neighboring work, it is possible to draw some conclusions about the relations between a model like the one of ours and those other two families of abstraction. The simulation tools, which are similar to our approach for the need of a complete characterization of both functional and non-functional aspects, differ in the objectives and place in a down-streaming phase. The above-quoted clustering algorithms are similar for what bears on the methodology, i.e. the genetic approach, but substantially diverge in the nature of the problem and place in an up-streaming phase. Our work lays amid these two areas, in a class of problems quite new for the WSN topics and typical of a maturing field, which focus is now broadening to include, along with classical low-level issues, also the need for an high-level optimization.

4.2 Model

As countered in previous section, the model aims at accurately reproducing the structural characteristics of a Wireless Sensor Network both at hardware and software levels. The target is to find the software configuration, as a mix of statically and dynamically allocated functions, that maximize the lifetime of the network while satisfying memory constraints and functional completeness.

4.2.1 Premises

The model takes in input the results of some activities of basic analysis that are already part of the best practices for developing efficient WSN infrastructures. These operations places in a previous phase as regards our model and their methodologies have been broadly studied and defined in the correspondent fields:

P.1 *Coverage and routing analysis.* This phase is always performed before implementing a WSN and consists in determining the partition of the network in *clusters* of nodes which best overgrows the area of interest, along with the corresponding routing trees.

P.2 *Task definition.* The identified clusters are a total partition on the set of nodes. The subsequent step is aimed at defining the *tasks* that each cluster should perform. Such tasks represent a set of activities (i.e. the measurements and processing that nodes accomplish in monitoring their proximal area) and are expressed as a collection of functions to be allocated to the nodes in a cluster. Different clusters are often associated with different

Let **N** be the set of all the nodes n_i composing the network, with cardinality $|\mathbf{N}| = \nu$, such that:

$\mathbf{N} = \{n_1, n_2, \cdots, n_\nu\}$

From the coverage analysis we inherit the optimal clustering for the network topology. Let call C the overall set of the clusters c_i, with cardinality $|C| = \kappa$, such that:

$C = \{c_1, c_2, \cdots, c_\kappa\}$
where:
$(c_i \neq \emptyset) \wedge c_i = \{n|$ for some $n \in \mathbf{N}\}, \qquad \forall c_i : i \in (1, \cdots, \kappa)$

These clusters determine, by definition, a total partition on the overall set of nodes **N**. Let $\sim_c$ be an equivalence relation among the set of nodes, such that:

$\forall n_i, \forall n_j$: $(i \neq j) \Rightarrow (n_i \sim_c n_j) \Leftrightarrow (n_i \in c) \wedge (n_j \in c)$

where n_i, $n_j \in \mathbf{N}$ and $c \in C$. From the properties of the *Equivalence Relation*, immediately follows:

$c_i \cap c_j = \emptyset \qquad \forall c_i, c_j \in C,$ such that $(i \neq j)$
and:
$\bigcup_{i=1,\cdots,\kappa} c_i = \mathbf{N}$

Table 4.1: Algebraic characterization of the output from coverage analysis

tasks, due to spatial variations in the environmental conditions (and hence in the parameters that must be monitored and/or controlled) among different parts of a macro-area. Could however happen that the same task must be executed by the totality of the clusters, that does not influence the model, but at most simplifies it. Considering most of the real applications, it is reasonable to assume that a single task is assigned to each cluster. Moreover, all nodes in a cluster may be considered *homogeneous* in the sense that all functions composing the task can be indifferently allocated to all nodes. Special cases that require a function to be redundant on several nodes may be expressed by means of a *redundancy factor*, suitably considered in the model.

Let **F** be the set of all the functions f_i have been implemented, with cardinality $\lvert\mathbf{F}\rvert = \xi$, such that: $\mathbf{F} = \{f_1, f_2, \cdots, f_\xi\}$ From the task-definition analysis we inherit a classification of the various functions in sets which represent macro-activities that should be performed by the relative clusters. Let call T the overall set of the clusters t_i, with cardinality $\lvert T\rvert = \tau$, such that: $T = \{t_1, t_2, \cdots, t_\tau\}$ where: $(t_i \neq \emptyset) \wedge t_i = \{f \mid \text{for some } f \in \mathbf{F}\}, \qquad \forall t_i : i \in (1, \cdots, \tau)$ Despite what happens between clusters and nodes, the set T does not define a total partition on the overall set of the functions **F**, so the intersection between two tasks t_i and t_j in T, such that $(i \neq j)$, does not necessary result in the *Empty Set*.

Table 4.2: Algebraic characterization of the output from task-definition analysis

P.3 *Energy estimation.* Each function is characterized with a set o f parameters indicating its frequency, average energy absorption per execution, average energy absorption for dynamic loading/unloading and average energy absorption for forwarding that function code through the routing tree.

The output of these three activities is the starting point for the definition of the model as described in the following. All relevant information for characterizing nodes and functions are expressed by a small set of parameters summarized in Table 4.3.

4.2.2 Domain definition

The purpose of the model is to create an adequate level of abstraction for coping with both the physical and functional (applicative) layers of a WSN, i.e. defining the costs and benefits of certain SW configuration, given a certain structure of

Parameters of node n_i		
M_i	(bytes)	available memory
E_i	(mAh)	energy capacity
$E_{\text{os},i}$	(mJ)	energy capacity
H_i		set of the successor nodes in the routing tree
Parameters of function f_i		
S_i	(bytes)	memory footprint
Φ_i		executions per hour
$E_{\text{ex},i}$	(mJ)	execution energy
$E_{\text{ld},i}$	(mJ)	loading energy
$E_{\text{fw},i}$	(mJ)	forwarding energy
$R_{L,i}$		minimum redundancy

Table 4.3: Nodes and function parameters

the network in terms of sensor hardware, networking structure and routing trees. Moreover, since we are working in a genetic environment through the use of a dedicated instance of the Genetic Algorithm, the formalization choices have been taken to fit, at the same time, the abstraction purposes and the GA's requirements.

The elementary entities constituting the network are *nodes* and *functions*. From the analyses described in the premises we can partition the nodes in a set of clusters and the functions in a set of tasks, as shown in Tables 4.1 and 4.2. As these tables point out, clusters are intrinsically defined as disjoint sets of nodes (example in Figure 4.1), while tasks are set of functions whose intersection is often not empty. From this consideration raises that clusters and tasks theoretically present a one-to-many relation. Nevertheless, for what is stated in premise (**p.2**) comes that it is always possible to arbitrary define each task such that it maps to one and only one cluster, by merging tasks allotted to the same cluster in a new macro-task or split a task in sub-tasks such that every cluster maps exactly to one, and only one, cluster. This statement essentially introduces the possibility to define tasks

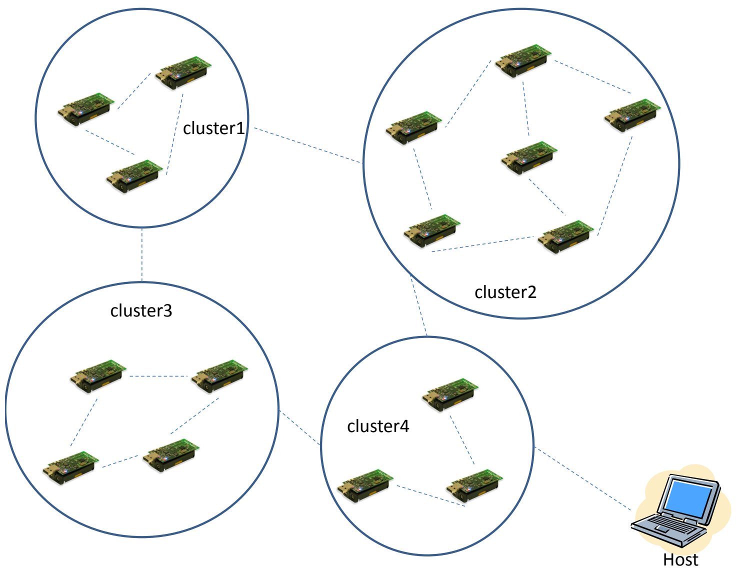


Figure 4.1: An example of clustering

and to map them on clusters in a way that always leads to a one-to-one relation. It can be formalized through the following:

Assumption 1 *A task collects all and only the functions that must be executed by a given cluster. Conversely, each cluster is assigned to a single task.*

Since clusters define a total partition on the network, on condition to opportunely take in account the routing paths (as we do), it is always possible to obtain a global optimization for the whole network, by optimizing the configuration of each cluster separately. For that reason, in drawing up our formalization, from now on the focus will be set on a single cluster c to which a single task t is assigned. Let $\mathcal{N} = \{n_i\}$ be the set of the nodes that constitute the cluster c and $N = |\mathcal{N}|$ the number of such nodes. Let also $\mathcal{F} = \{f_i\}$ be the set of functions composing the task t assigned to the cluster c and let $F = |\mathcal{F}|$ be the number of such functions[1]. As introduced, functions can be either statically or dynamically

[1]As regards the characterizations introduced in Tables 4.1 and 4.2, the set $\mathcal{N}$ of all the nodes composing the cluster c is a subset of the set $\mathbf{N}$ of all the nodes composing the network. The set $\mathcal{N}$ is also an equivalence class entailed by the equivalence relation $\sim_c$ on the cluster c (as shown in

allocated to nodes; to keep track of these two types of function allocation, we introduce the following definitions:

$$\text{sta}(f_i, n_i) = \begin{cases} 1 & f_i \text{ statically allocated on } n_i \\ 0 & \text{otherwise} \end{cases} \quad (4.1)$$

$$\text{dyn}(f_i, n_i) = \begin{cases} 1 & f_i \text{ dynamically allocated on } n_i \\ 0 & \text{otherwise} \end{cases} \quad (4.2)$$

Another important assumption comes from a practical evidence, which suggests that in the greatest part of real applications, all the nodes in a cluster can be considered *functionally equivalent*, i.e. every function in task t could be mapped indifferently on every node in cluster c. This happens because the nodes composing a cluster are 'topologically neighboring' and usually cover a small area in which the phenomenon to be monitored does not significantly change. The following assumption clarifies this concept.

Assumption 2 *All nodes of a cluster are functionally equivalent, that is, they are suited for executing any of the functions of the corresponding task.*

The third and last assumption for the model is expressly stated to consider an important design choice, characterizing all the actual implementations of Wireless Sensor Networks: the availability of timestamp-based mechanisms to maintain all the nodes aligned to a global shared time. This is a pillar of the Distributed System Theory (e.g. [80]) and of WSNs consequently [81]; the practical incarnation of this theoretical paradigm is obtained by means of special modules, included in most of WSN's operating systems, in charge of 'scheduling' events and operations according to a global policy. This turns out particularly useful to determine the functions execution policy, when a certain frequency of execution is specified[2].

Table 4.1) and it also disjointed by all the other sets of nodes characterizing other clusters. The set $\mathcal{F}$ of the functions characterizing the task t is a subset of the set $\mathbf{F}$ of all the functions defined for that network, but, as said, it is not necessarily disjointed from the sets of functions characterizing other tasks.

[2]The execution frequency is an input of the model and it is specified for each function as shown in Tale 4.3. The execution frequency is useful when a function must monitor a certain parameter

Since all the nodes composing a cluster are *functionally equivalent*, according to the **Assumption 2**, each execution of a function could be run on each node of the cluster indifferently, so the frequency characterizing that function could be divided among the number of nodes of the cluster on which that function has been allocated.

When a function is allocated to a certain set of nodes in a cluster, these nodes arrange the operating system modules in charge of scheduling shared events to synchronize each other in order to schedule the execution of this function in a way that guarantees the global accomplishment of its frequency. The architecture we are working on [29] holds a special module of the operating system dedicated to the above-quoted purpose, such that the frequencies of execution of the functions are scheduled among the nodes according to the relative allocation.

Assumption 3 *There exist a component of the operating system capable of executing a given function according to the configuration resulting from the optimization and synchronized on a common time basis.*

It is worth noting that this module needs not to be *explicitly* isolated from the application, i.e. it is not just a peculiarity of the operating system but could also be implemented as a built-in utility of a specific application. Since such a module could be implemented as part of the overall logic of the application itself, there could be found specific environments autonomously providing these capabilities and this is exactly the case of the framework used for our experiments: as said, WASP holds an ad-hoc module, referred to as *non-functional manager*, conceived to schedule and organize events on a shared time-line [29].

4.2.3 Constraints

The target of the model's optimization is to maximize the cluster lifetime L.

There are, however, some constraints that must be satisfied, concerning both the functional and non-functional requirements/limits of a wireless sensor net-

for a certain number of times per hour, per day, etc. For example: the temperature must be sampled 5 times per hour by the nodes belonging to a cluster positioned along a lake reef...

work. The first constraint is a non-function one and refers to the amount of free allocable memory for each node in the cluster. There are then two functional requirements, the first of which states that every function in the task must be allocated on at least one node in the cluster, while the second asserts that each function must be allocated according to a certain *redundancy factor*[3].

These constraints can be formalized as:

C_1 The amount of memory required to store all static and dynamic functions of each node must not exceed the amount of memory available.

C_2 The functionality of the task must be guaranteed, that is, each function must be statically or dynamically allocated to one node at least.

C_3 The functionality of the task must be robust to node loss, according to predefined criteria. This means that certain functions may be required to be statically allocated to nodes with a certain redundancy.

Let L be the lifetime to be maximized and let C_1, C_2 and C_3 be condition variables evaluating to 1 if the corresponding constraint is met and 0 otherwise. The three condition variables have thus the following form:

$$C_1 = \begin{cases} 1 & \text{mem}(n_i) < M_i, \quad \forall n_i \in c \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

$$C_2 = \begin{cases} 1 & \text{inst}(f_i) > 0, \quad \forall f_i \in t \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

$$C_3 = \begin{cases} 1 & R_{L,i} \leq \text{red}(f_i), \quad \forall f_i \in t \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

[3]The redundancy factor is an input of the model (see Table 4.3) that indicates the minimum percentage on the cardinality of nodes, on which a function must be statically allocated. This parameter is particularly useful when a certain function is critical and must statically reside on a certain percentage of nodes, because it is event-triggered or monitors important parameters. In these cases, in fact, the nodes should perform in real-time and the overtime for dynamically retrieving and loading that function must be avoided: from this, the choice of statically allocate such functions.

The function to be maximized is thus:

$$Q = L \cdot C_1 \cdot C_2 \cdot C_3 \tag{4.6}$$

It should be noted that this function shows a stepwise behavior on all boundaries where the three conditions of the constraints change from true to false or vice versa. This kind of function may degrade the performance of the genetic algorithm when using certain math-tools, especially for very large problems. A solution to this potential drawback might consist in re-defining C_1, C_2 and C_3 by means of sigmoid functions such as the error function or the logistic function, as shown at the end of this section. In the experiments considered in this work, the stepwise behavior of the condition variables did not influence the evolution of the genetic algorithm[4].

Let us now consider the three conditions and express their mathematical form in full detail.

The memory required for a set of functions allocated on node n_i can be expressed as the sum of a static and a dynamic contribution:

$$\mathrm{mem}(n_i) = \mathrm{mem}_{\mathrm{sta}}(n_i) + \mathrm{mem}_{\mathrm{dyn}}(n_i) \tag{4.7}$$

where the static contribution equals the sum of the size of all statically allocated functions, that is:

$$\mathrm{mem}_{\mathrm{sta}}(n_i) = \sum_{j=1}^{F} \mathrm{sta}(f_j, n_i) \cdot S_j \tag{4.8}$$

while the dynamic contribution equals the maximum size of all the dynamically allocated function, that is:

$$\mathrm{mem}_{\mathrm{dyn}}(n_i) = \max_{j=1..F} \mathrm{dyn}(f_j, n_i) \cdot S_j \tag{4.9}$$

This last equation is motivated by the assumption that dynamic functions can always replace each other when loaded, if there is not sufficient memory space to allocate them at the same time.

[4]The Linux GAUL libraries we have used to implement the genetic algorithm, in fact, are not affected by the stepwise behavior of the constraints. Moreover, since GAUL is an utility for defining genetic algorithms as C-Programs, the stepwise way in which C_1,C_2 and C_3 are stated well fit the functional programming style of C-Language.

As regards the second constraint, the instances of a function are counted regardless of the nature (statical or dynamical) of their allocation, that is:

$$\mathrm{inst}(f_i) = \sum_{j=1}^{N} \mathrm{dyn}(f_i, n_j) + \mathrm{sta}(f_i, n_j) \tag{4.10}$$

Finally, in dealing with the third constraint, the redundancy of a function considers all the statically allocated instances of each function and is expressed as a fraction of the potentially allocable instances[5], such that:

$$\mathrm{red}(f_i) = \frac{1}{N} \cdot \sum_{j=1}^{N} \mathrm{sta}(f_i, n_j) \tag{4.11}$$

***A continuous variant based on sigmoid function**

It is possible to remove from the model the discontinuities introduced by the stepwise functions by means of a sigmoid function such as the error function or the logistic curve:

$$\text{sigmoid(t)} = \frac{1}{1 + e^{-\alpha t}} \tag{4.12}$$

where $\alpha > 0$ is an arbitrary parameter which forces the *sigmoid* to be mashed up toward a *step function*, insofar as the designer thinks it appropriate for the specific tool on which the GA has been built. In that way, the three constraints become respectively:

$$\mathbf{C_1} = \prod_{i=1}^{N} \text{sigmoid}(M_i - mem(n_i)) \tag{4.13}$$

$$\mathbf{C_2} = \prod_{i=1}^{F} \text{sigmoid}(\mathrm{inst}(f_i) - \epsilon) \tag{4.14}$$

[5]The maximum number of potentially allocable instances for a function is equal to the cardinality N of the cluster: that means the function is allocated to every node in the cluster.

Where ϵ in (4.14) is a small differential ($\epsilon \approx 0.1$), which forces[6] the argument of the *sigmoid* to be negative, in the case of $\mathrm{inst}(f_i) = 0$.

$$\mathbf{C_3} = \prod_{i=1}^{F} \Big(\mathrm{up\_lower}(f_i)\Big) \tag{4.15}$$

where $\mathrm{up\_lower}(f_i)$ is defined as:

$$\mathrm{up\_lower}(f_i) = \textbf{sigmoid}(\mathrm{red}(f_i) - R_{L,i}) \tag{4.16}$$

The meaning of Equations (4.13),(4.14), (4.15) and (4.16) should be clear in the light of what stated previously about the three constraints and the properties of the sigmoid. For a brief explanation, let's consider that the Equation (4.13) forces C_1 to zero if, for one or more nodes, the value of the required memory overcomes that of the available memory: since the argument of the sigmoid in this case becomes negative and the sigmoid returns zero if its argument is negative, the product forces C_1 to zero. The meaning of the other two equations should be clear for what has just been shown.

The extension of the model by means of the sigmoid function could be useful for some tools based on the GA which show better performances in manipulating derivable functions (this is not the case of GAUL libraries for unix/linux systems, that only require a function to be continue, not necessarily derivative).

4.2.4 Optimization goal

Provided that all constraints are satisfied, the goal of the model's optimization is to maximize the lifetime L of the entire cluster, so let's now analyze the nature of this variable.

[6]If the argument of the sigmoid is exactly zero, the function would return the value 0.5, that is meaningless if we want to define a binary condition of the type true-or-false (in this case, satisfied/unsatisfied constraint). With a negative argument, on the other hand, the sigmoid returns zero, hence becoming appropriate to the purpose.

We assume that a cluster could be considered *alive* if and only if *all* of its nodes are operating. This definition is clearly pessimistic since after the first node finishes its energy and thus dies, the remaining nodes can take its workload in charge thanks to either static redundancy or dynamic reconfiguration. On the other hand, however, the condition in which all nodes are alive is the best among all the possible states through which a cluster can pass: if a node dies, in fact, the extra-workload that other nodes must bear strongly compromises the remaining lifetime. In addition, the presumably small amount of energy to the nodes' disposal[7] along with the super-linear discharging trend of the batteries under a certain autonomy threshold, makes the remaining nodes to exponentially converge toward their death. We can thus define the lifetime$L$ of the cluster as:

$$L = \min_{i=1..N} L_i \tag{4.17}$$

where

$$L_i = E_i \,/\, \overline{P}_i \tag{4.18}$$

is the lifetime of node n_i and can be calculated as the ratio between the available energy E_i of node n_i and its average power consumption $\overline{P}_i$ which depends on the specific functional configuration of the node.

The power consumed by a node comes from two different sources. The first source is related to the *continuous* activity of a portion of the application and the operating system that is always executing and is out of the scope of this analysis. This first contribution can thus be modeled as the energy $E_{\mathrm{os},i}$ consumed per unit time (hours in our formalization). The second contribution is related to the specific configuration of the cluster and it is in turn composed by three terms.

The first term is the *static energy* $E_{\mathrm{sta},i}$, that is the energy consumed by the functions statically allocated to node n_i, and can be expressed as:

$$E_{\mathrm{sta},i} = \sum_{j=1}^{F} \mathrm{sta}(f_j, n_i) \cdot E_{\mathrm{ex},j} \cdot \frac{\Phi_j}{\mathrm{inst}(f_j)} \tag{4.19}$$

[7]Because if a node dies it is plausible stating that a certain time is elapsed since the network has been deployed.

where Φ_j is the number of required executions of the function f_j per unit time (hours, again). According to Assumption 3 the overall number of executions of a function is distributed over all nodes where the function is allocated, that is $\text{inst}(f_j)$. In conclusion, the number of executions of function f_j on the node n_i is $\Phi_j/\text{inst}(f_j)$.

The second term is the *dynamic energy* $E_{\text{dyn},i}$, that is the energy consumed by a node to load and execute all the functions dynamically allocated on it. Similarly to Equation (4.19), the dynamic energy can be expressed as:

$$E_{\text{dyn},i} = \sum_{j=1}^{F} \text{dyn}(f_j, n_i) \cdot (E_{\text{ex},j} + E_{\text{ld},j}) \cdot \frac{\Phi_j}{\text{inst}(f_j)} \tag{4.20}$$

The last term is the *routing energy* $E_{\text{route},i}$, that is the energy due to forwarding of the code from the base station to a destination node (different from n_i) through a routing path including the node n_i. This contribution is the summation of forwarding energies associated to those functions that are dynamically allocated on successors of the current node in the routing tree[8]:

$$E_{\text{route},i} = \sum_{j=1}^{F} \sum_{n_k \in H_i} \text{dyn}(f_j, n_k) \cdot E_{\text{fw},j} \cdot \frac{\Phi_j}{\text{inst}(f_j)} \tag{4.21}$$

where the inner summation extends over all nodes n_k, with $k \neq i$, in the set H_i of the successors of node n_i in the routing tree.

The average power $\overline{P}_i$ consumed by node n_i per unit time is thus:

$$\overline{P}_i = \frac{E_{\text{os},i} + E_{\text{sta},i} + E_{\text{dyn},i} + E_{\text{route},i}}{\tau} \tag{4.22}$$

In the present paper we assume that the unit time τ equals one hour and thus the lifetimes L_i and L are expressed in hours.

[8]Because a dynamically allocated function could be loaded and then unloaded to make space for a new dynamic function. If this happens, when the previous function has to be loaded again, the node must re-download it from the host or the gateway node that stores the repository of the functions for that cluster.

4.3 Implementation

The model has been implemented in C-Language for Unix-Linux architectures, taking advantage of the features provided by the Genetic Algorithm Utility Libraries (GAUL) [28], expressly developed for these architectures.

The design of the optimizer has flooded through two conceptually separated steps: the first has been related to the logical mapping of the model on the Genetic Domain, while, in the second, coding and testing activities have been accomplished.

4.3.1 Mapping to the Genetic Domain

Due to the formalization of the model, intentionally conceived to meet the particularities of the Genetic Domain, the mapping has been quite a natural operation.

Morphism 1 *Each* ***cluster*** *is an* ***individual****, whose genetic patrimony is composed by as many* ***chromosomes*** *as many are the* ***nodes*** *composing the* ***cluster***

$$(c \mapsto \iota), \quad \forall c \in C, \forall \iota \in P \tag{m.1}$$

where C is the overall set of clusters and ι is an individual belonging to the population P.

Morphism 2 *Each* ***node*** *of a* ***cluster*** *corresponds to one of the* ***chromosomes*** *of the* ***individual*** *on which this cluster has been mapped. For each* ***individual****, the number of* ***chromosomes*** *is thus equal to the cardinality of the relative* ***cluster****.*

$$(n \mapsto \chi), \quad \forall n \in c, \forall \chi \in \iota \tag{m.2}$$

where n is a node belonging to the cluster c, while χ is a chromosome in the genetic patrimony of the individual ι, such that (m.1) is verified and $|\iota| = |c|$.

Morphism 3 *Each* ***chromosome*** *has a number of alleles equal to the cardinality of the* ***task*** *assigned to the* ***cluster****. Since we can consider a chromosome as and array of* ***alleles****, each of them is associated to the* ***function*** *with relative index*[9] *in the* ***task****. Each* ***function*** *is then mapped to an* ***allele*** *and this* ***allele*** *can only assumes the values {0,1,2} indicating respectively the three possible states of a* ***function*** *on a* ***node****, i.e. {not allocated, statically allocated,dynamically allocated}.*

$$(f \mapsto \gamma), \quad \forall f \in t, \forall \gamma \in \chi \tag{m.3}$$

where γ is an allele and χ a chromosome, such that $\gamma \in \chi$ and $|\chi| = |t|$. As usual f represent a function and t the task to which this function belongs.

Morphism 4 *We state that an* ***individual*** *is* ***well fitted*** *if it satisfies the Constraints C_1, C_2 and C_3 described in Section 4.2.3, thus, whose fitness function (4.6) is not 0. The* ***best fitted*** *individual in a population is the* ***well fitted*** *individual who holds the global maximum for the fitness function.*

4.3.2 Application design

This step comes immediately from what has been presented in Section 4.3.1 and the structure of GAUL libraries. Each GAUL *entity* represents a cluster, the *chromosomes* of an entity are the nodes composing the cluster, initialized as arrays of integers *alleles*.The number of these *alleles* is equal to the cardinality of the task assigned to the cluster. Each *allele* represents a function of the task and can only assume values {0,1,2}, indicating that the relative function is respectively {not allocated, statically allocated, dynamically allocated} on this node.

The fitness function has been implemented through a specific hook function, checking the constraints and calculating the optimization goal as defined in Sections 4.2.3 and 4.2.4.

[9] We here assume that an arbitrary order has been defined on the functions in the task, such that each function can be identified by means of its index.

In the experiments described in the following the genetic optimization is performed for a single cluster, iterating for some generations (generally from 150 to 300), each composed of a population of 1000 individuals with the same structure but different configurations as regards functions' allocation. The population's individuals are instances of the same cluster, bearing different configurations as regards the functions allocation of its nodes(i.e. the configuration of the allele in the chromosomes).It has been observed that a number of 150 generations is a good compromise to converge toward the fitness asymptote in a reasonable time.

The evolution scheme is Darwinian. The two best fitted parents are re-scored and brought to the next generation. The mutation is random and the crossing-over is based on a single-point scheme.

Further remarks are needed for what concerns the insemination function, i.e. the function that generates the initial configurations of the individuals on the basis of a random-driven logic.

Initial insemination

The initial insemination is a critical issue concerning the evolution of the genetic algorithm. The constraints introduced in Section 4.2.3, in fact, strongly reduce the probability of generating an admissible configuration of a cluster, working in a totally random logic and running the algorithm for a reasonable number of generations on a reasonable population size. It could be noticed that also problems with a medium complexity offer a probability space unmanageable through a completely casual insemination, considering the tight requirements entailed by the three families of constraints. Let's observe that also a relatively small cluster of 10 nodes, in charge of executing a relatively small task of 20 functions, each presenting three different types of allocation, leads to an enormous number of possibible configurations:

$$\text{Possible configurations} = (3^{20}) \cdot 10 \approx 35 \cdot 10^9$$

Running the algorithm on a population of 1000 individuals over 150 generations, through mutation and crossing over, it is possible to obtain just a negligible number of configurations as regards the overall possibilities:

$$\text{Generable configurations} = 1000 \cdot 150 = 15 \cdot 10^4$$

in other words, five orders of magnitude under the overall design space.

It is clear that such a discrepancy mashes the probability of finding an admissible solution[10] up to zero. Since the algorithm needs to find an admissible solution from which starting, to be improved by means of matings, mutations and crossing-over among the generations, the insemination function must be optimized in order to increase the probability of generating admissible solutions as soon as possible. This optimization, however, should be as much random as possible, in order to preserve the intrinsic casual nature of genetic evolutions.

At this purpose we have studied a solution which preserves an high-degree of variability, while providing a good statistical confidence of generating an admissible solution in a reasonable time. The logic is based on a partial stochastic coverage of two of the three constraints, i.e. the completeness and redundancy constraints, by seeding the nodes in a random way, such that:

- For each function, the seeding-algorithm guarantees that at least one casually chosen node presents an allocated instance of this function. The allocation is in turn casually elected between the two possible choices, i.e. static or dynamic $\{1 \text{ or } 2\}$;

- In presence of thigh redundancy requirements for a given function, the seeding-algorithm guarantees that a randomly chosen set of nodes presents a statical allocation for that function in a casual percentage, neighboring the one required by the redundancy parameter.

This arrangement preserves an high rate of variability and, at the same time, enable a consistent insemination as regards the effectiveness of the algorithm. It is worth noting that such an optimization does not affect the memory constraints and

[10]An admissible solution is a configuration that does not violate the constraints, hence not forcing the fitness function to be zero.

also act regardless the energy requirements of the various configuration. That implies a lot of individual to be completely unfitted, once inseminated, and also those constituting admissible solutions practically never represent the optimal ones. It is just a coherent implementation that enables the algorithm to find a randomly-generated admissible configuration to gradually optimize by means of matings, mutations and crossing-over in proceeding through the generations.

4.4 Experimental results

To validate the model as well as its implementation through the genetic algorithm, we have performed several sets of experiments, testing the most relevant properties and behaviors. It must be noted that, according to the assumption in Section 4.2.3, dynamic functions are allocated *one at a time*, which is clearly a pessimistic situation.

All the experiments refer to the same task, made of 32 functions with significantly different complexity, ranging from simple calculations to more demanding sensing and communication. The task size has been chosen according to the probability space it entails in terms of possible configurations. Since each function in the task represents an allele, which can assume three different values $\{0,1,2\}$, and since each chromosome holds 32 alleles, the possible configurations for a single node are about $\mathcal{P} = 1.8 \cdot 10^{15}$ and so the possible configurations for the entire cluster turn out to be $\mathcal{P}_{tot} = \mathcal{P} \cdot N$, where N is the cardinality of the cluster (i.e. the number of nodes in that cluster). With such a number of possibilities, the assessments on model's robustness as regards the optimization power are particularly reliable.

The four most relevant parameters of the functions, namely the memory footprint, the energy required for one execution, the energy required for loading and the energy required to forwarding a function are summarized in Table 4.4.

The nodes composing the cluster are all the same but have different amounts of available memory. All nodes are powered with a 1150mAh battery, supplying

Id	Size (B)	Freq. (exec)	$\mathbf{E_{ex}}$ (μJ)	$\mathbf{E_{ld}}$ (μJ)	$\mathbf{E_{fw}}$ (μJ)	Id	Size (B)	Freq. (exec)	$\mathbf{E_{ex}}$ (μJ)	$\mathbf{E_{ld}}$ (μJ)	$\mathbf{E_{fw}}$ (μJ)
F01	96	0.1	131	1268	634,1	F17	123	0.3	168	1632	816,2
F02	2742	0.005	2569	4951	13632,8	F18	1933	0.001	1812	3491	9614,4
F03	280	0.05	1417	1665	1585,2	F19	315	0.005	1600	1880	1789,0
F04	284	0.1	133	1537	1585,2	F20	335	0.6	157	1816	1873,1
F05	140	5	3	1268	951,1	F21	115	6	3	1050	788,0
F06	330	3	3	1537	1902,3	F22	329	8	3	1537	1902,0
F07	1206	2	3	2795	6023,8	F23	1552	3	4	3597	7754,0
F08	164	1	65	1573	951,1	F24	163	1	64	1565	946,0
F09	940	0.09	549	2585	4755,6	F25	767	0.6	448	2111	3884,7
F10	940	0.005	4015	2585	4755,6	F26	1206	0.006	5154	3318	6105,0
F11	74	0.1	104	1390	634,1	F27	69	0.3	97	1296	591,6
F12	268	0.1	100	1665	1585,2	F28	256	0.2	95	1594	1517,0
F13	70	1	57	1481	634,1	F29	90	0.6	73	1920	822,1
F14	444	1	85	1941	2219,3	F30	385	0.4	73	1686	1928,0
F15	300	0.4	97	1757	1585,2	F31	255	0.7	82	1494	1347,8
F16	492	0.007	3545	1941	2536,3	F32	508	0.008	3665	2007	2622,7

Table 4.4: Summary of task's functions characteristics

a voltage of 1.8V, and execute a common set of tasks (refereed to as *basic tasks*[11] in the following) consuming an average energy of 600μJ per hour. The number of nodes vary from experiment to experiment as described in the next Section.

Furthermore, nodes are arranged in a routing topology where they have a variable number of successors distributed as summarized in Table 4.5. This means that, in any given cluster, 15% of the nodes belong to group 1 and have 6 successors (statically and randomly chosen on initialization), 15% of the nodes belong to group 2 and have 4 successors, and so on.

4.4.1 Fitness stability on memory variations

The first set of experiments aims at measuring the fitness evolution, while progressively relaxing the boundaries introduced on memory usage. What we expect

[11]Energy consumed in running Operating System and low-level functions

Group Id	Fraction	Number of Successors
1	15%	6
2	15%	4
3	15%	2
4	55%	0

Table 4.5: Compact representation of routing tables

to observe is a trend of the fitness that increases as the constraints become less requiring and the algorithm has more boundary-free optimization choices. Moreover, this trend should converge toward a common asymptote for values on the memory overlaying a certain threshold, giving to the model a sufficient degree of free-optimization. Another expected behavior concerns the composition of the optimal configuration: in giving to the algorithm more independence from the constraints we foresee a uniform distribution of the static allocations among nodes. That's because statically allocated functions consume less energy compared to the dynamically allocated ones.

This series of experiments refer to a cluster of 10 nodes, accomplishing the above-presented task of 32 functions without redundancy. The available memory has been set to different values, ranging from 2742 bytes (i.e. the maximum memory footprint reported in Table 4.4) to 12KB. The population is composed of 1000 individuals and the algorithm runs for 150 generations. Evolving more than 150 generations and using a larger population would still increase the fitness, but the gains obtained are negligible in most cases.

Figure 4.2 shows the evolution of the fitness (that equals the lifetime in hours) for the five memory sizes. It can be noticed that the lifetime reaches an asymptotic maximum in a number of generations between 50 and 100. As expected, augmenting the available memory has the effect of increasing the lifetime until a certain threshold. The model confirms the expectations and shows a good degree of stability, since it converges toward a unique asymptote when the available memory exceeds a certain threshold (4KB, in this case). As countered, this happens because the constraint on the available memory becomes looser as its size

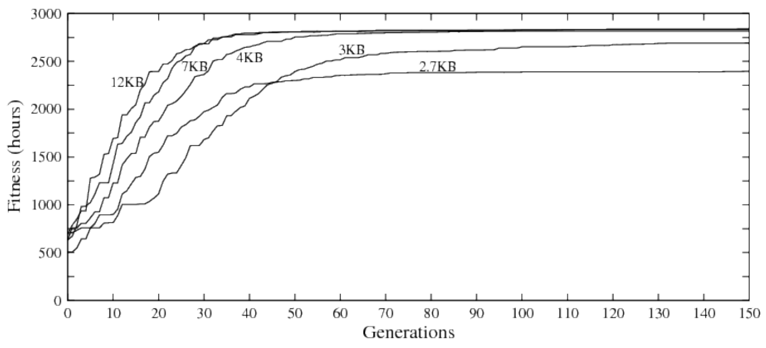


Figure 4.2: Fitness evolution for five memory sizes

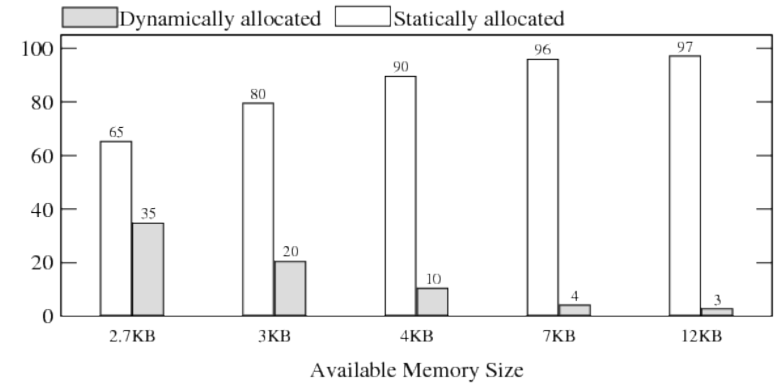


Figure 4.3: Static/dynamic allocation fractions

increases.

Moreover, increasing the memory has the natural effect of statically assigning more functions to nodes, as expected. The fraction of statically and dynamically allocated functions is shown in Figure 4.3. Since no redundancy is required, functions are allocated on the minimum number of nodes with the sole goal of maximizing the lifetime. In particular, the average fraction of allocated functions, per each node, varies from 25%, with 2.7KB of memory, to 40%, with 12KB of memory.

4.4.2 Redundancy

The second set of experiments has the goal of demonstrating the capability of the model and of the tool to satisfy redundancy constraints. This means that the resulting configuration must statically allocate each function on a percentage of nodes greatest or at least equal to that specified in the redundancy parameter of the function. Moreover, the algorithm must accomplish this task satisfying the other constraints and maximizing the fitness. For this series of experiments have used a cluster with 10 identical nodes, providing 4.5KB of available memory, to implement the task described in Table4.4. Some functions belonging to this task have been required to be redundant on some nodes in different percentages. The test has been executed for three different combination of redundancy requirements. In defining the redundancy we have classified five different degree of saturation: Each function has been characterized with the respective redundancy level, whose

Level	Minimum Percentage	Comment
N	0	Don't care
E	1	static on at least one node
L	25%	static on at least the 25% of nodes
M	50%	static on at least the 50% of nodes
H	75%	static on at least the 75% of nodes

Table 4.6: Level of redundancy

values are reported in Table 4.7.

Following we present the results obtained by the three instances of this class of experiments, each exploring a different criticality concerning the introduction of a redundancy constraint.

Red1: This first experiment is performed on a very variegated redundancy distribution. The target is to test the algorithm on redundancy requirements for a large number of functions in a memory-constrained environment.

The results demonstrate a very good level of configuration since, as Figure 4.4 shows, the redundancy constraints for all the functions have been satisfied. The overall fitness has been optimized as well and its value amounts to 2785 hours. The average fraction of the overall allocated function, per each node, is 36.9%. Among these, the fraction of statically allocated func-

Id	Red1	Red2	Red3	Id	Red1	Red2	Red3
F01	E	N	N	F17	L	N	N
F02	N	H	N	F18	N	N	E
F03	E	N	N	F19	N	N	N
F04	N	N	N	F20	N	N	N
F05	L	N	N	F21	E	N	N
F06	N	N	N	F22	N	N	N
F07	N	N	E	F23	N	N	E
F08	M	N	N	F24	L	N	N
F09	N	N	E	F25	N	N	N
F10	N	N	E	F26	N	N	E
F11	H	N	N	F27	M	N	N
F12	N	N	N	F28	N	N	N
F13	E	N	N	F29	M	N	N
F14	N	N	N	F30	N	N	N
F15	E	N	N	F31	E	N	N
F16	N	N	N	F32	E	N	N

Table 4.7: Values of redundancy for the three simulations

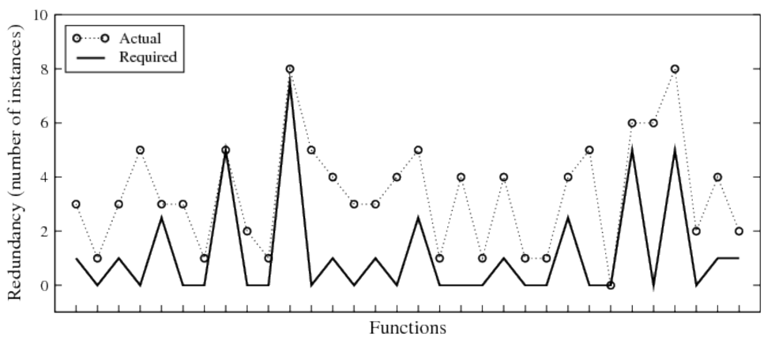


Figure 4.4: Redundancy obtained by experiment Red1

tions is about the 91.1%, coherently with the redundancy requirements and the behavior pointed out in the Section 4.4.1.

Red2: This second test presents an high redundancy for the function in the task with the heaviest memory footprint. The aim is to prove the model robustness in satisfying redundancy constraint on an high percentage of nodes for a very large function, that considerably restricts the field of free-configuring choices for the algorithm.

The resulting redundancy is reported in Figure 4.5, that once again confirms a coherent behavior in respect to the constraints. The fitness has reached a value of 1348 hours, a very low lifetime if compared to the experiment *Red1*, but perfectly aligned to our forecasts, because in statically allocating the heaviest function on an high percentage of nodes, there is very few space to allot other large functions statically. The great energy consumption associated to the dynamic loading and forwarding of these functions makes the fitness to be reduced considerably.

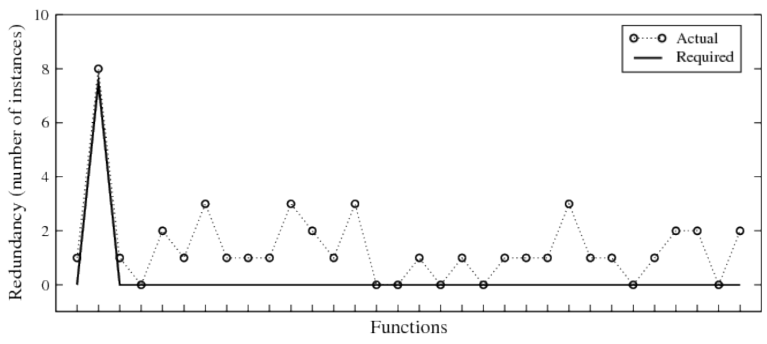


Figure 4.5: Redundancy obtained by experiment Red2

Red3: The last redundancy experiment lays on the need to control the algorithm response when some of the heavier functions, concerning the memory footprint, are required to be statically allocated at least on one node in the cluster. We expect this condition not to worsen the overall fitness, because a uniform static allocation of large functions among nodes would theoretically reduce the overall power-consumption of the cluster.

The optimization output is reported in Figure 4.6. The algorithm has satisfied redundancy constraints, scoring an overall fitness of 2748 hours: a very good threshold, as expected.

Finally we have performed a free optimization on the same cluster of nodes, each preserving its energy and memory characteristics.

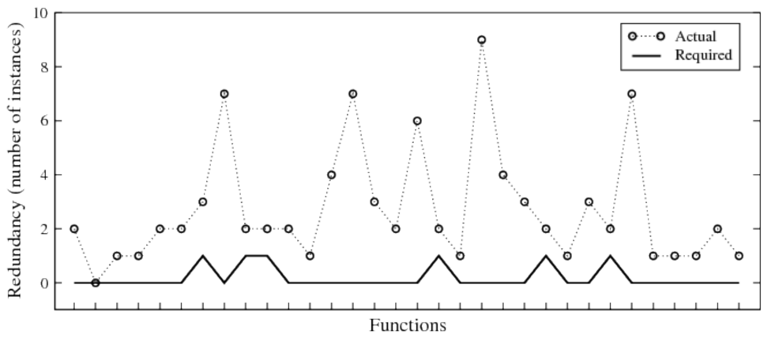


Figure 4.6: Redundancy obtained by experiment Red3

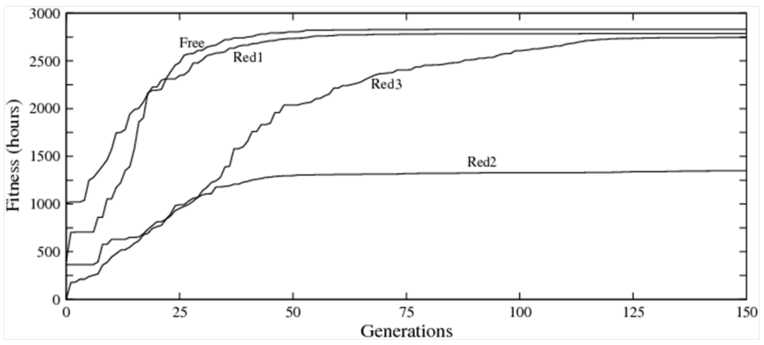


Figure 4.7: Fitness for the three experiments on redundancy

The fitness trend has then been compared to those of the three experiments above in order to evaluate the capability of the model in maintaining its optimization power, even under redundancy boundaries. Results are shown in Figure 4.7 and, as can be noticed, they demonstrate a very good behavior of the algorithm. An exception could be found for the fitness curve of experiment *Red2*, but, for what has been stated above, this evolution matches the expectations for the relative kind of redundancy requirements. It's worth noting the optimization effort in maximizing the lifetime while proceeding through the generations, even though the redundancy boundaries. Constraints on task completeness have been perfectly respected as well.

4.4.3 Heterogeneous clusters

This experiment tests the algorithm's robustness in optimizing a cluster composed by an heterogeneous set of nodes, some of which characterized by an available memory smaller than that required by the largest function in Table 4.4.

The aim is to demonstrate the capability of the model to allocate the various functions—according to their memory footprint—even to very small nodes, which cannot accommodate larger functions. The composition of the cluster includes seven nodes: $N1$ with 0.3KB of available memory, $N2$–$N3$ with 1KB, $N4$–$N6$ with 4KB and $N7$ with 7KB.

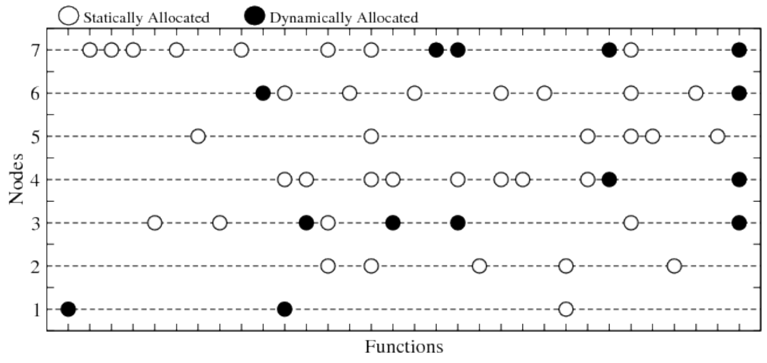


Figure 4.8: Configuration of an heterogeneous cluster

Figure 4.8 shows the configuration obtained by the algorithm. This result shows two important achievements: on one hand, *all* nodes have been used, including the smallest, and on the second the available memory is exploited in a very efficient way, as Table 4.8 shows.

This configuration has been obtained running the algorithm for 150 generations over a population of 1000 individuals. The reached fitness is 2473 hours and the final allocation satisfies the three constraints described in Section 4.2.3. Moreover, as could be observed in Figure 4.8 all the functions composing the task has been allocated, either statically or dynamically, hence satisfying the constraint on task completeness.

Node	Available	Static	Dynamic	Overall
N1	300	54.3%	32.0%	86.3%
N2	1000	95.8%	0.0%	95.8%
N3	1000	44.3%	50.8%	95.1%
N4	4000	66.5%	30.2%	96.7%
N5	4000	71.3%	0.0%	71.3%
N6	4000	69.1%	23.5%	92.6%
N7	7000	71.6%	27.6%	99.3%

Table 4.8: Memory usage

4.4.4 Functions' frequency

This test aims at monitoring the behavior of the algorithm in relation to the execution frequency of a function. Though no constraints exist on the allocation logic concerning the frequency according to which a function must be executed, for what has been stated by Assumption 3 in Section 4.2.2, the execution of a given function is distributed among the various nodes by means of a non-functional manager. This non-functional manager is a special module of the operating system on a node, which coordinates with the other instances of non-functional manager acting on the other nodes to accomplish the distributed execution of each function according to its frequency and the functional configuration of each node.

What we expect is that in increasing the execution frequency of a given function, the algorithm will statically allocate this function on an high percentages of nodes in the cluster. That's because if a function needs to be executed with an high rate of instances per hour, it would be very expensive to dynamically loading and unloading its code every time. Moreover, since the frequency is distributed among the nodes who have that function alloted, the model should assign such a function to the greatest number of possible nodes, taking advantages of the distributed scheduling of non-functional managers.

The experiment has been conducted on a cluster of 10 nodes, each providing 4KB of available memory. The functions' task is the one reported in Table 4.4 and the first instance has been run according to the frequencies defined in this table.

Results from this preliminary test are following referred as *base exp*, to which the perturbed solutions are compared.

Then we have perturbed the frequency parameter for the function with the largest footprint and for that with the smaller one.

exp1 In this experiment the execution frequency of *F2* has been brought from 0.005 to 50 execution per hours. The number of statical allocations in presence of such a perturbation are shown in Figure 4.9, compared to those of the *base exp*.

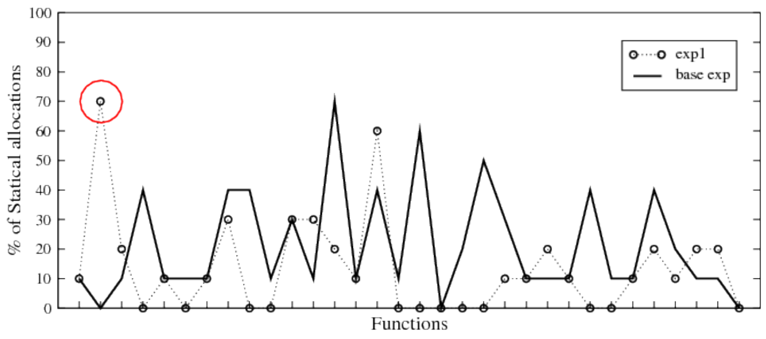


Figure 4.9: Result of perturbation on F2 frequency.

It's worth noting that the algorithm has shown a good sensitivity to the frequency increment, since the number of statical allocation has passed from the 0% of nodes –in the base exp– to the 70% of nodes when the frequency has been increased.

exp2 The same operations has been done for the second test with the smallest function *F27*, whose frequency has been increased from 0.3 to 50 executions per hours. Even in this case the results demonstrate a very good degree of sensitivity to the frequency variations even for a small function, whose percentage of static allocations has passed from the 10% up to 90% of nodes in the cluster.

These results acquire great importance since no constraints are specified for the execution frequency. So the behavior of the algorithm is completely driven

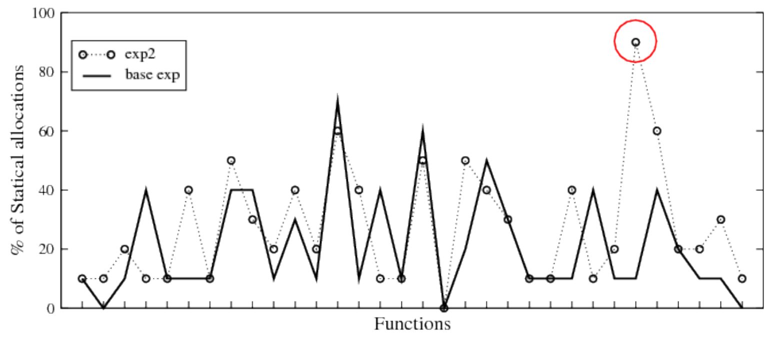


Figure 4.10: Result of perturbation on F27 frequency.

by the optimization logic, which target the minimization of energy consumption in order to maximize the lifetime of the network. So the algorithm well behaves according to the Assumption 3. One last word should be spent in commenting the two results in relation to the function footprint:

- For what concerns large functions, such as *F2*, when the execution frequency grows it is good to observe a consequent increase of the static allocations. This is because dynamically loading those function with an high frequency would mean to consume the network energy in few hours. Moreover, thanks to the distributed scheduling performed by non-functional managers, the overall workload for executing such functions may be balanced among a certain number of nodes in the cluster;

- Bearing on small functions, on the other hand, it ought to be noted that they occupy just a small amount of the available memory, thanks to their small footprint. Hence, statically distributing such functions even on a large percentage of nodes, don't significantly affect the degree of memory-free optimization of the algorithm. At the same time, this choice drastically reduce the overall energy consumption for the above-mentioned reasons. Concluding, the algorithm has demonstrated a good behavior in managing frequency variations.

4.4.5 Allocation coherence

The test on allocation coherence has the purpose of monitoring the algorithm's behavior in configuring the applicative domain on the nodes of a cluster, according to the three constraints and in presence of an increasing exacerbation of the memory requirements. In defining the test set, we have foreseen five different optimizations on as many clusters composed by an increasing number of nodes. All nodes are provided with 2742 bytes of available memory, i.e. the maximum footprint among the 32 functions in the benchmark task.

Cluster	**Number of nodes**	**Node Ids**	**Fitness**
C1	1	{1}	41.1
C2	2	{2,3}	154.2
C3	3	{4,5,6}	297.7
C4	4	{7,8,9,10}	413.1
C5	10	{11,12,13,14,15,16,17,18,19,20}	2394.6

Table 4.9: Clusters with an increasing number of identical nodes

What we expect is that the algorithm configures the bigger clusters with more statically allocated function, because of the greater amount of memory at its disposal. Nevertheless, since the constraints on cluster completeness must be satisfied, even the smallest cluster should be allocated with all the functions, eventually dynamic in all their instances. Obviously, we foresee a decreasing level of fitness little by little the total available memory decreases. The configuration of the five Clusters, along with the resulting fitness, have been reported in Table 4.9.

Let's now have a look at the five configurations coming out from the optimization. The first cluster *C1*, has been configured as shown in Figure 4.11.

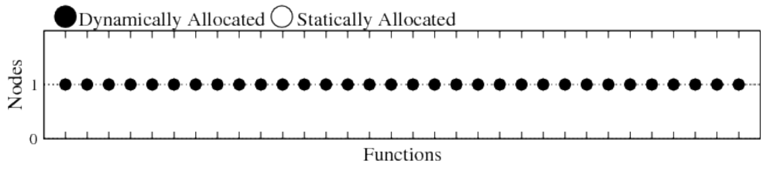


Figure 4.11: Configuration output for the cluster C1

It comes up evident the correctness in the algorithm's behavior: all the functions have been allocated, satisfying the task completeness, and the allocation type is dynamic for all of these ones. It is, in fact, the only admissible configuration, since the available memory equals the maximum among functions' footprints, needing a fully-dynamic management in order to guarantee the execution of the task. The capability of the model in finding the only admissible configuration in just 150 generations and 1000 individuals of population represents a remarkably clue of robustness and effectiveness.

Results for clusters *C2*, *C3* and *C4* are reported in Figures 4.12–4.14.

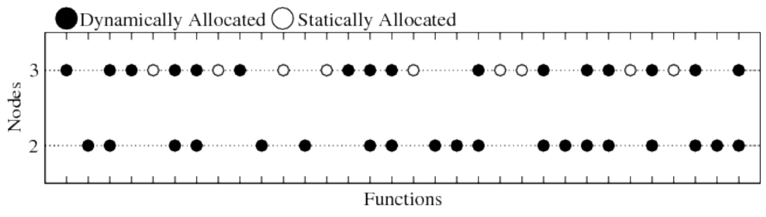


Figure 4.12: Configuration output for the cluster C2

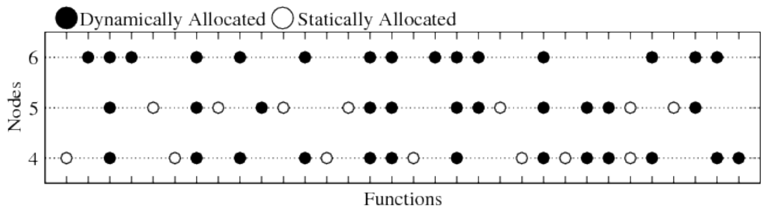


Figure 4.13: Configuration output for the cluster C3

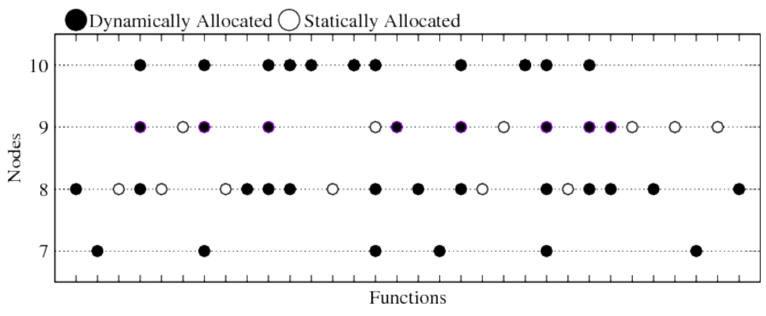


Figure 4.14: Configuration output for the cluster C4

The configurations of these intermediate clusters point out three interesting properties of the model:

1. Increasing the number of nodes and the overall available memory consequently, the percentage of statically allocated functions becomes remarkable, notwithstanding the tight memory characterizing a single node;

2. The heaviest function, i.e. *F2*, has been dynamically allocated in all of these cases. This implies a twofold benefit: on one hand, the space has been better exploited since many functions could run on a node, where just the heaviest function would reside if it was statically allotted. On the other hand, according to the Assumption 2 in Section 4.2.2, also the dynamically allocated functions concur to the distribution of the execution frequency of a function among the nodes. Hence, the heavy energy requirement entailed by the forwarding and loading of a dynamic function is partially counterbalanced by the gain in frequency distribution. At this purpose, we remind the pessimistic assumption of *one-at-time* execution for dynamic functions, which well define worst cases, but clearly does not apply when two or more medium or small-sized functions execute at the same time.

3. Cluster *C4* presents less statically allocated functions than cluster *C3*. This might seem a contradiction, since *C4* has more nodes than *C3*, but it is perfectly in-line with the nature of genetic-based optimizations. Running the genetic algorithm for only 150 generations, in fact, may not give a sufficient time to the algorithm for reaching the absolute optimum: that's because there are not enough iteration to perform an adequate number of mutations and crossings-over, especially when the number of nodes grows and the number of combinations along with it. Nevertheless, the overall fitness increases of about 30% in passing from $C3$ to $C4$, i.e. adding just one more memory-constrained node. This attests the general efficiency of the algorithm.

Finally we discuss the configuration of the biggest cluster in this test-set. Taking a look to the configuration of cluster $C5$ (Figure 4.15) two of the three observations above reported find a stronger evidence.

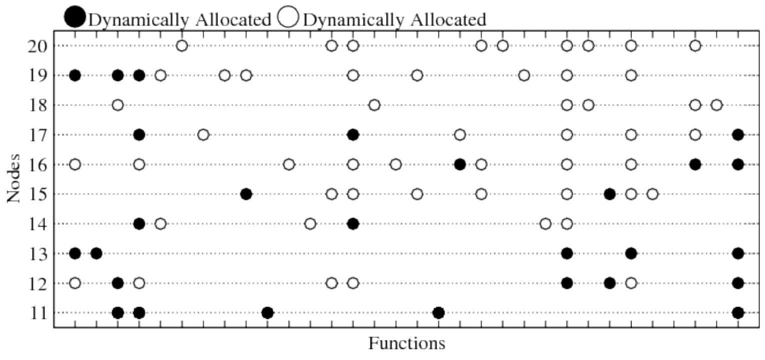


Figure 4.15: Configuration output for the cluster C5

First, the number of statically-allocated functions has dramatically increased thanks to the more availability of memory on the cluster. Secondly, *F2* has been once again dynamically alloted, this time on *Node 13*, offering the possibility to dynamically allocate other functions on the same node (in this case *F1,F24,F27* and *F32*). The overall fitness is 6 times higher than that of cluster $C4$, which has 6 less nodes. It could be noticed a strong discontinuity in the growing ratios of the fitness functions among the various clusters (Table 4.10).

Prev. Cluster	**Succ. Cluster**	**Δ nodes**	**Δ memory (%)**	**Δ fitness (%)**
C1	C2	+1	100.0%	275.2%
C2	C3	+1	50.0%	93.1%
C3	C4	+1	33.3%	38.8%
C4	C5	+6	150.0%	479.7%

Table 4.10: Fitness improvement in dependence of nodes' augment.

This aspect should not be a surprise, since the two major variations happens between $C1$ and $C2$, as well as between $C4$ and $C5$: in both this cases, in fact, the constraint on the available memory receives two great lifts that completely redefine the nature of the problem. Passing from $C1$ to $C2$, in fact, means to evolve from the most constrained among the possible configuration to another one that enables a certain degree of free-configuring. On the other hand, passing from $C4$ to $C5$ means to jump from a constrained configuration, though not the most constrained, to a superior degree of free optimization. In $C5$, in fact, the total amount of memory is more than 10KB greater than the sum of all the footprints of the functions composing the task. In $C4$, on opposite, the total amount of memory is about 7KB smaller than the sum of all functions' footprints. Concluding, it's worth raising a last remark for what concerns the task completeness requirements, the adherence to which turns out evident from the configuration of the clusters reported in Figures 4.11–4.15.

4.4.6 Comparative experiments

In this section we propose a critical comparison between three classical ways of programming WSNs –based on hand-made, empirical solutions– which we called *standard configurations*, and the correspondent allocations obtained by running the algorithm on the same clusters.

Moreover, we have created another variant of the algorithm, that does not perform a casual insemination, but seeds, at time-zero, all the individuals with the standard configuration given as input by the user. We have called this version of the algorithm as *improvement algorithm* and the relative results as *improved configurations*, since they are refinements of the hand-made configurations, obtained by running the algorithm for a certain number of generations. To distinguish this class of solutions from those obtained by running the random-inseminated version of the algorithm, we called the ones obtained by the latter as *free configurations*, where the word *free* means that the model randomly create the initial configurations at time zero.

What has been observed, analyzing the results, is that the *free configurations* obtained by running the random-inseminated version of the algorithm, always overwhelm both the *improved configurations* and the *standard configurations*. Moreover, the *improved configurations* have ever scored a better fitness, compared to that of the corresponding *standard configurations*. Furthermore, a more basic benefit tagged along the better optimization-power as regards the complexity of the problems: hand-made solutions, in fact, are possible only for small clusters, which also become unmanageable when a more sophisticated constraint, such as the redundancy, is required. In more complex situations, the support of computer-based computations is a necessary condition.

The combination of these three advantages point out an overall convenience in using genetic algorithm's solutions. Following we report results and configurations obtained by three different comparisons.

Benchmark 1

This first experiment considers a clusters composed of six nodes provided with 3KB of available memory, in charge of executing the task of functions described in Table 4.4. The routing tree characterizing that cluster is described in Table 4.11.

Node	Successors
1	{2,3,4,5}
2	{3,4,5}
3	{5}
4	-
5	-
6	-

Table 4.11: Routing tree for the cluster in benchmark 1.

The logic used in defining the hand-made solution, that we following call *initial solution* is based on the classical saturation problem, widely used in actual WSNs implementations. This consists of statically allocating to each node the maximum number of functions, until the memory is completely saturated. The resulting configuration is shown in Figure 4.16.

The resulting fitness for this configuration amounts to 2068.57 hours. Though it could appear an optimal configuration, since the task completeness has been respected and all the functions have been statically allocated, the following optimizations show that it's not so. The results obtained by running the *improvement*

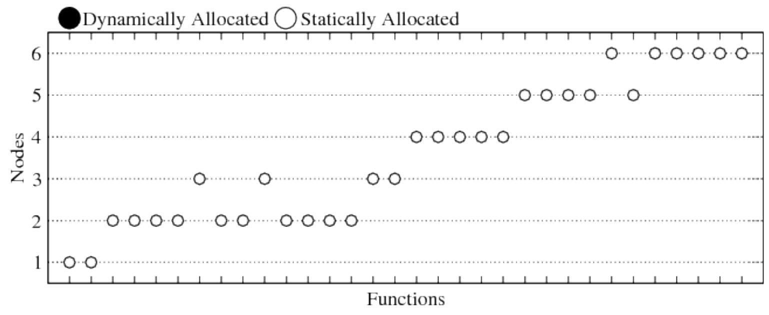


Figure 4.16: Hand-made configuration of benchmark1.

algorithm starting from the configuration reported in Figure 4.16 have registered a consistent gain in fitness score, which as growth up to 2282 hours. The algorithm has been run for 150 generations, over a population of 1000 individuals. The resulting configuration is shown in Figure 4.17.

Finally we have performed on that cluster an optimization through the base version algorithm, i.e. the randomly-inseminated one. This optimization has demonstrated to perform better than the other two variants obtaining a fitness of 2465 hours. The resulting configuration is reported in Figure 4.18.

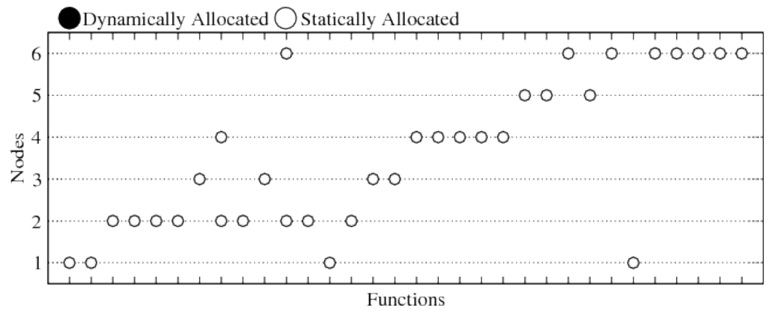


Figure 4.17: Improved configuration of benchmark1.

As could be noticed, this configuration also considered some dynamically allocated functions: that's prove the benefits of introducing a dynamic reprogramming mechanism to enhance the life span of the overall network, by introducing more flexibility in its configuration.

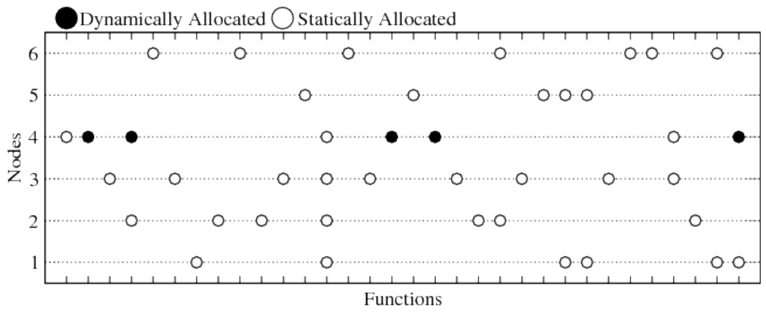


Figure 4.18: Free configuration of benchmark1.

Benchmark 2

The second test bases on a cluster of three nodes, each providing a memory of 3KB, in charge of executing the same task of *benchmark 1*. The routing tree for that cluster is reported in Table 4.11.

Node	**Successors**
1	{2,3}
2	{3}
3	-

Table 4.12: Routing tree for the cluster in benchmark 2.

In this case the total available memory is smaller than the overall memory required by the task, hence not allowing the static allocation of the whole set of functions. What commonly is done in configuring the cluster by hand, in such a situation, is to statically allocate the heaviest functions and dynamically allot the remaining ones. That's because the functions with a larger footprint require a significant amount of energy to be dynamically loaded and forwarded, if compared to the smaller ones. The hand-made solution has thus been made by statically allocating for each of the three nodes the heaviest functions, i.e. *F2*, *F18* and *F23*. The remaining functions have been dynamically allocated. The resulting configuration is shown in figure 4.19, which has scored a fitness of 76.47 hours.

Giving in input this configuration to the *improvement algorithm*, the fitness

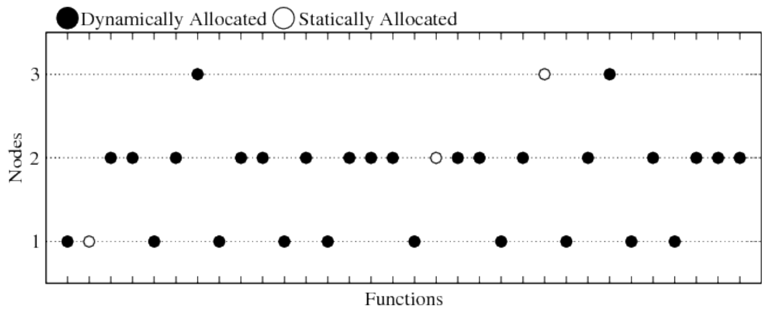


Figure 4.19: Hand-made configuration of benchmark2.

reported after 150 generations amounts to 297.12 hours, presenting an overall enhancement of about the 300%. Once again it raises an overall gain in improving standard solutions by means of the genetic model here presented. The *improved configuration* is shown in Figure 4.20, where could be observed a differt allocation logic compared to that commonly adopted in hand-made configurations.

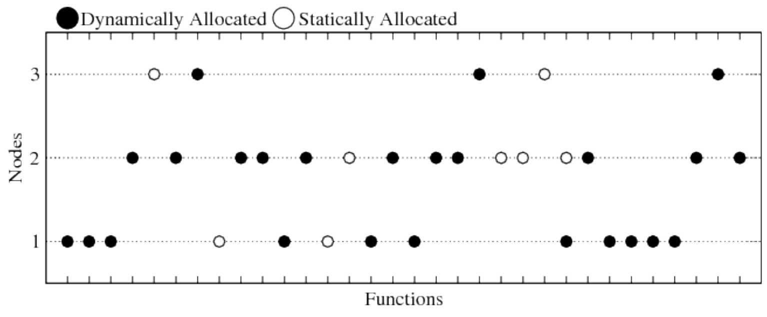


Figure 4.20: Improved configuration of benchmark2.

Finally, we have performed an optimization through the randomly-inseminated algorithm, obtaining a fitness of 371.15 hours, i.e. an improvement of about the 400% on the hand-made configuration. This allocation is described in Figure 4.21.

As could be noticed, the way by which the algorithm has chosen statically and dynamically allocations has changed once again, leading to straightforward enhancement of the scored fitness.

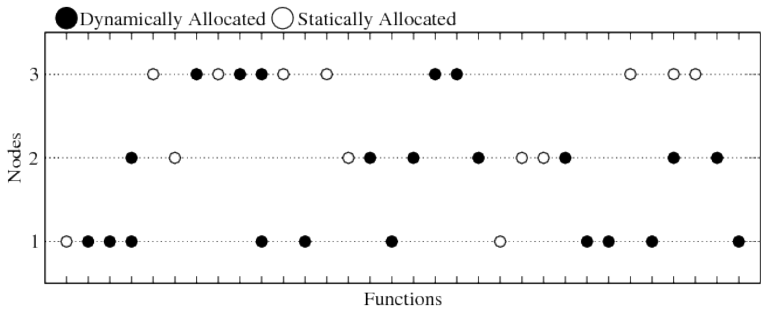


Figure 4.21: Free configuration of benchmark2.

Benchmark 3

This last experiment further restricts the boundaries on available memory, presenting a cluster of just two nodes providing 3KB of available memory. The task is the same used in previous experiments and the routing tree is very simple, since *Node 1* has the *Node 2* as its successor, while *Node 2* does not have successors. Here we want to prove another common allocation logic, used in hand-made configurations. When the memory resources are particularly constrained, it is worth trying to allocate the maximum number of functions statically. Since the maximum footprint among the set of functions amount to 2742 bytes, i.e. the 91.4% of available memory on a single node, the only way to maximize the number of statically allocated functios is to dynamically allot *F2* on one of the two nodes. Then, the larger functions are dynamically allocated to the same node in which *F2* resides, while all the remaining functions are statically assigned until the memory is saturate. The hand made configuration is reported in Figure 4.22, showing a fitness of 24.5 hours.

Running the *improvement algorithm* on this configuration for 150 generations, we have obtained a lifetime of 74 hours, showing on overall gain of about the 200% on the standard salution. The resulting allocation is reported in Figure 4.23.

It's worth noting that also in this case the genetic model has opted for a more intensive use of dynamic allocations for the smallest functions, confirming the usefulness of a dynamic linking mechanism in maximizing the network's lifetime.

Finally, we propose the result obtained in running the randomly-inseminated

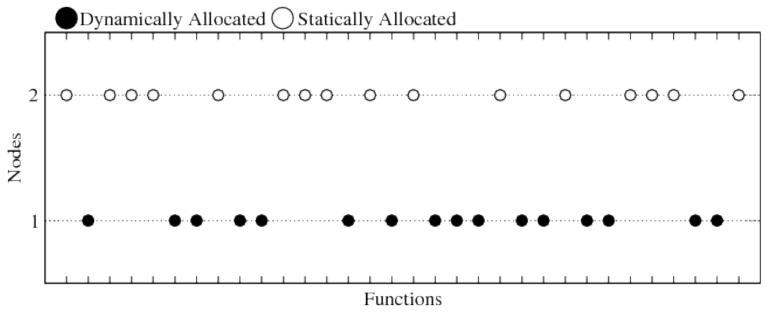


Figure 4.22: Hand-made configuration of benchmark3.

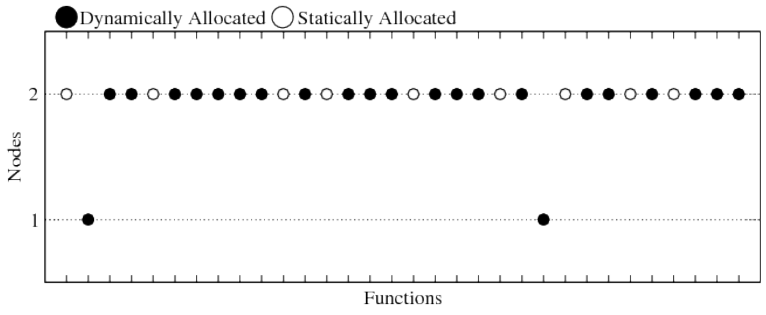


Figure 4.23: Improved configuration of benchmark3.

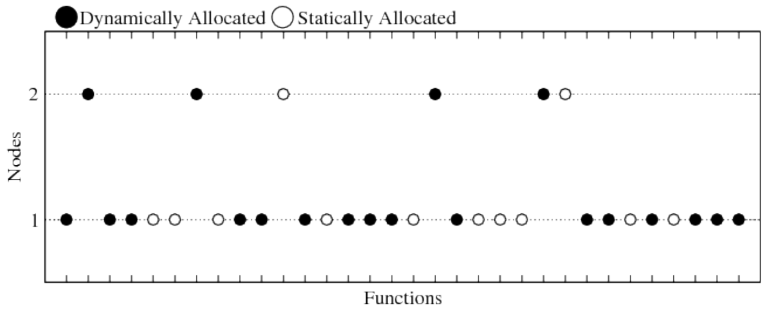


Figure 4.24: Free configuration of benchmark3.

algorithm on this cluster. Figure 4.24 shows the resulting configuration, which leads to a fitness of 149.6 hours, greater than the initial one of about the 500%. The performed allocation outlines one more time an efficient usage of the dynamic linker, which heads to a very siginificative improvement of the cluster's lifetime, if efficiently allocated.

Evidences synopsis

The analysis conducted on the comparative experiments among the most diffuse configuring practices, shows a remarkably advantage in using the systematic approach proposed by our model over empirical allocations. The gain in terms of fitness improvement for the three benchmarks are summarized in Figure 4.25

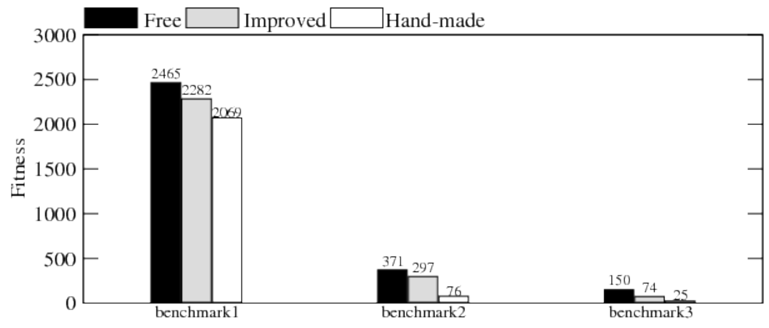


Figure 4.25: Comparison between empirical practices and algorithm's optimizations.

Concluding we can draw three important considerations about the systemic use of the genetic model for WSNs' programming:

1. the algorithm, in its original version, always perform better than common empirical practices in configuring WSNs' applicative domain;
2. it's also possible to exploit the algorithm optimization-power to improve the fitness of a standard configuration, using it's variant that takes in input an hand-made configuration instead of performing random inseminations;
3. results obtained by the randomly-inseminated algorithm always overwhelm those returned by other versions. This confirms the correctness of the algorithm, since it shows superior optimizations when made free to choose it's own optimization path.

4.5 Sensitivity Analysis on Population Size

The sensitivity analysis on population size aims at testing the algorithm sensitivity to the number of individuals composing the population.

Theoretically, in increasing the number of individuals the algorithm should enhance its optimization power, especially in the very first generations. That's because when a larger number of randomly-inseminated individuals are generated, the probability to find an admissible solution significantly grows. Thus, if it would be possible, the ideal population should counts thousands or more individuals. That's, however, dramatically worsen the execution time of the algorithm and make the problem practically unresolvable with a general-purpose dual-core machine.

The idea is then to test the sensitivity of the model as regards the population size, such that it is possible to determine a good compromise between the probability of finding an admissible solution in a reasonable time, while not charging the models with an excessive processing overhead, that would make it fitted just for supercomputers.

At this purpose we have run the algorithm on a cluster of 10 nodes, provided with 4KB of available memory, accomplishing the benchmarking task in Table 4.4. The simulation has been set to proceed for 150 generations and the population size has been tested on the following values[12]:

pop1 100 individuals;

pop2 500 individuals;

pop3 1000 individuals;

pop4 2000 individuals.

The resulting trend of the fitness function for the different population sizes is reported in Figure 4.26.

From what emerges by the fitness curves, population sizes underlaying the 1000 individuals show worst evolution trends. That's because the probability of

[12]Values exceeding the 2000 individuals have not been tested because it would exploit an excessive computation time, without leading to significant improvements. As following discussed, in fact, the algorithm reaches the asymptote for population sizes greater or equals to 1000 individuals. To have further enhancement, the size would be increased by an order of magnitude, but this would make impossible the algorithm to run on a general-purpose computer.

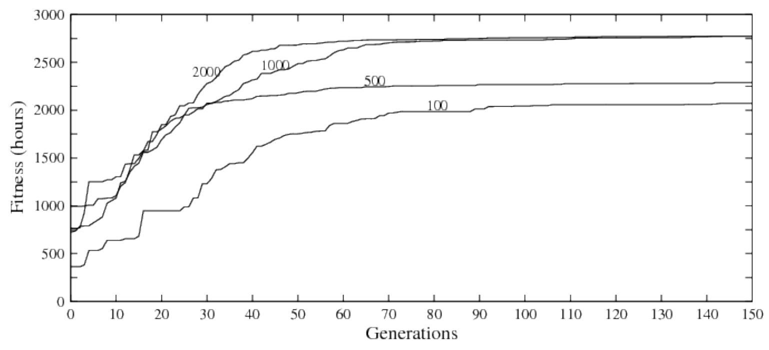


Figure 4.26: Fitness trend for different sizes of the population.

finding an optimal solution in the small number number of combinations implied by those populations is very limited. On the other hand, populations containing 1000 or more individuals reach a common asymptote after a certain number of iterations orbiting the 100 generations. Moreover, even though the population of 2000 individuals grows faster than that of 1000 individuals, the computational effort required by the latter is far more efficient in terms of execution time and processor effort. Population sizes exceeding the 2000 individuals are too expensive for running on general purpose machines, against a minimal gain in fitness scores. Other tests, in fact, have been performed on population sizes greater than those above reported and they have always shown negligible variations with respect to those presented, along with an heavier execution time and an expensive processor effort.

The best compromise between efficiency and effectiveness consists thus in using population of 1000 individuals, what we have actually done in accomplishing all these experiments.

A note should also be spent for what concerns the number of generations. It's clear that protracting the number of iteration for more than 150 generations would certainly increase the probability of scoring a better fitness, but the gain is very limited. As could be noticed, in fact, the greatest part of optimizations reach the fitness asymptote before the 150 generations. We omit a dedicated study on generations sensitivity since it would lead to an easily foreseeable conclusion, indeed. The best mix could then be obtained in using populations of 1000 individuals and

150 generations.

4.6 Concluding remarks

Some final considerations are here provided to complete the picture concerning the model and the relative experimental results.

Few insights are worth to be furnished about the initial insemination, that does not belong to the sphere of model's core issues but may help in better understanding the application design.

The individuals composing a population are randomly generated according to user-defined percentages of static and dynamic allocation. The algorithm always shows better performance with initializations characterized by a small ratio of overall allocation, mainly composed by dynamic functions. This predilection for rarefied initializations grounds on the risk-reduction they offer in avoiding memory boundaries overflows and redundancy constraints violations in the early generations. That's because at the beginning of its iterations, the model needs just to find an admissible solution, tough not optimal, to canalize its evolution in enhancing the configuration in next generations.

Preliminary experiments have shown that starting with a population satisfying (or *almost* satisfying) all the constraints leads to better optimization results. Studying this issue will be a central topic of our future work for further optimizations.

Another remark should be raised to point out that the present model has been based on several *worst case* hypotheses, in order to fortify its robustness and cover any potential exception at the same time. There are cases, although, in which these limitations could be relaxed and studying such circumstances will be object of our future work, as well.

Chapter 5

Conclusions

5.1 Directions

In this Thesis research we have faced the problem of reprogramming and configuring wireless sensor networks, characterized by tight resource constraints.

The work has been driven by the twofold target of implementing a power-efficient mechanism for dynamically updating software on remote nodes and to define a robust and effective mathematical framework for optimizing the functional configuration of the network. The main guideline has been that of optimizing the network lifetime by minimizing energy consumption, while providing all the necessary flexibility to make WSNs' deployment as much effective as possible.

The various experiments have demonstrated that both the dynamic linking/loading mechanism and the genetic model offer very promising perspectives in enhancing WSNs' capabilities. The first because of its energy efficiency and reprogramming effectiveness; the second thanks to its powerful support in defining compelling configurations of the network, even in presence of an high rate of complexity, which would otherwise make impossible human-defined solutions.

We have also demonstrated that in boosting the flexibility degree of the functional domain, aiming at the same time to minimize non-functional aspects such as the energy consumption, there comes an overall improvement of the network potentialities.

The dynamic reprogramming mechanism we have proposed places at the interception of two already known approaches in WSN literature: dynamic linker and pre-linked loadable modules. Both these techniques, however, suffer from very harsh limitations: the first, as regards energy consumption and memory requirements, the second for what bears on the lack of scalability and unsynchronized alignment between host and nodes. Our proposal takes advantages of the strong points of each of the previous and introduces an hybrid configuration, which enables the creation of small ready-to-load modules by dynamic linking standard ELF files on host. The host, on opposite to what happens for classical pre-linked modules approaches, is always maintained aligned to the actual configurations on remote nodes by means of some meta-data (i.e. the symbol table), that contain information about actual references of variables and functions in motes' images.

The genetic model has been defined on strong mathematical basis, considering the applicative facets as well as the networking issues and hardware specifications of a WSN's domain. The main goal has been that of taking the most from the flexibility introduced by the dynamic reprogramming mechanism to find a (sub)optimal allocation for a certain set of functions, which can be statically or dynamically assigned to a set of memory-constrained nodes for maximizing the network's lifetime. This constitutes an original contribution for the WSN's field of study, whose focus is still entrapped on networking and low-level issues that no longer constitute the only concerns, being now at the threshold of real-word WSNs' deployments which require an adequate functional support too. Results obtained in running the algorithm on real WSNs parameters have revealed a bright way for further researches and investigations. Moreover, it has been opened a new front of research for what orbits the WSN functional evolution.

These works have required a great effort, both in terms of time and intellectual commitment. The experimental nature of our proposals keeps the door widely opened for future works and inquiries, aiming at improving both their theoretical and applicative frameworks. Following we present an outlook on future directions for the present works.

5.2 Future Works

Some future works have already been planned for both the dynamic linker/loader and the genetic model. There are many interesting veins of development in front of the current versions and there is also an outstanding opportunity of integrating their frameworks with those by other components of WSNs' operating systems and applications.

5.2.1 Dynamic Linker/Loader

Symbol table hashing: The large size of the global nodes symbol table is due for the most part to the strings representing symbol names. By defining a suitable hashing technique, long names can be substituted with constant-length signatures. This has the twofold advantage of reducing the symbol table size (and thus saving memory on the node and diminishing the energy required for downloading) and of making the search routines faster.

Differential symbol table download: When a symbol is needed for remote linking and the contents of nodes symbol table does not coincide with the local copy stored on the node, then the entire nodes symbol table must retrieved. It is very likely, though, that only a very small subset of the symbols might has changed. Differential download allows retrieving only the information related to the symbols that have changed. This can either be achieved by adding a compact dirty bit vector to the table or by modifying the structure of each symbol table entry.

Protocol optimization: First of all we have noticed that the protocol overhead accounts for a fraction of the overall energy consumption that ranges from a minimum of 13% to a maximum of 50%. A first obvious optimization consists in simplifying the protocol currently implemented by reducing the number of request and acknowledgment packets. The first results obtained show that this energy overhead can be reduced to 80J.

Distributed database: at the moment, only local databases are implemented to store the retrieved symbol tables. This constitute a good solution if all the nodes of wireless sensor network are managed by a single host. If the dimension of WSN is such that many hosts are needed to control it, a distributed database for symbol tables has to be implemented. This database will contain all the updated symbol tables.A generic host, who needs to modify the status of the network, can access this database to have consistent and updated information about the node. A possible solution to synchronize the global database with the local ones could be a differential accumulation mechanism, based on delta-plus/delta-minus tables, well known in Database's literature.

5.2.2 Genetic Model

Managing pessimistic hypotheses: The model has been conceived to be as much robust as possible. To this purpose we have toughen up some hypotheses to encompass the largest number of possible eventualities. Though these pessimistic hypotheses make the model very reliable, in some cases they inevitably cause the fitness value to appear underscored respect to its potential maximum. The most relevant example of such a kind of hypotheses regards the dynamically allocated functions, which, according to the model, could be executed only in *one-at-time* mode. This surely happens when the functions' footprints cover a large percentage of the available memory but, on the other hand, it does not apply to the case of small-sized functions, that can be executed all together until the node's memory is saturated. Managing this hypothesis such that it could intelligently adhere to the specific context will certainly be object of further investigations.

Investigating optimal inseminations: It has been noted that starting at generation-zero with all the individuals satisfying model's constraints make the optimization process to converge more rapidly toward its asymptote and, in some cases, to reach also a better fitness. Another line of research will thus be followed in studying further optimizations of the insemination function, such that it could generate a larger number of admissible configurations, while preserving a certain degree of variability.

On-line reconfigurations: Another interesting possibility consists in binding the applicative logic of the model to the real-time evolution of a network. In doing that, it would be possible to on-line reconfigure the functional allocation among the nodes in response to some changes of the run-time conditions.

Integration with dynamic linker and NF managers: Finally, the most ambitious goal is that of integrating the logic of the model with the dynamic linker and the non-functional managers. In other words, the foremost achievement would be that of providing a network with the maximum degree of local ”‘intelligence”’, such that of reducing at the minimal terms the need for human control. Moreover, the energy-efficiency entailed by the adoption of an integrated framework for self-configuring, would lead WSNs toward more useful and effective real-world deployments.

Bibliography

[1] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: enabling ultra-low power wireless research. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 48, Piscataway, NJ, USA, 2005. IEEE Press.

[2] Wasp: Wirelessly accessible sensor populations, ist project n. 034963, sixth framework programme, at: www.wasp-project.org, cordis.europa.eu/fetch?caller=proj_ict&action=d&cat=proj&rcn=79339.

[3] Chee-Yee Chong and S.P. Kumar. Sensor networks: evolution, opportunities, and challenges. *Proceedings of the IEEE*, 91(8):1247 – 1256, aug. 2003.

[4] P. Coy and N. Gross. 21 ideas for the 21st century. *Business Week*, pages 78 – 167, aug. 1999.

[5] N. Habermann. Dynamically modifiable distributed systems. In *Proceedings of the Distributed Sensor Net Workshop*. Carnegie-Mellon University, Pittsburgh, Penn., December 1978.

[6] Julian W. Gardner and Vijay K. Varadan. *Microsensors, Mems and Smart Devices*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[7] Vladimir Trifa, Lewis Girod, Travis Collier, Daniel T Blumstein, and Charles E. Taylor. Automated wildlife monitoring using self-configuring sensor networks deployed in natural habitats. In *International Symposium on Artificial Life and Robotics (AROB07)*, Beppu, Japan, January 2007.

[8] Geoffrey Werner-Allen, Konrad Lorincz, Matt Welsh, Omar Marcillo, Jeff Johnson, Mario Ruiz, and Jonathan Lees. Deploying a wireless sensor network on an active volcano. *IEEE Internet Computing*, 10(2):18–25, 2006.

[9] Paritosh Padhy, Rajdeep K. Dash, Kirk Martinez, and Nicholas R. Jennings. A utility-based sensing and communication model for a glacial sensor network. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1353–1360, New York, NY, USA, 2006. ACM.

[10] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM.

[11] Victor Shnayder, Bor-rong Chen, Konrad Lorincz, Thaddeus R. F. Fulford Jones, and Matt Welsh. Sensor networks for medical care. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 314–314, New York, NY, USA, 2005. ACM.

[12] Malik Tubaishat, Peng Zhuang, Qi Qi, and Yi Shang. Wireless sensor networks in intelligent transportation systems. *Wirel. Commun. Mob. Comput.*, 9(3):287–302, 2009.

[13] Yadong Wan, Lei Li, Jie He, Xiaotong Zhang, and Qin Wang. Anshan: Wireless sensor networks for equipment fault diagnosis in the process industry. In *SECON*, pages 314–322, 2008.

[14] Constantin Volosencu. Identification of distributed parameter systems, based on sensor networks and artificial intelligence. *WTOS*, 7(6):785–801, 2008.

[15] Tian He, Sudha Krishnamurthy, John A. Stankovic, Tarek Abdelzaher, Liqian Luo, Radu Stoleru, Ting Yan, Lin Gu, Jonathan Hui, and Bruce Krogh. Energy-efficient surveillance system using wireless sensor networks. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile*

systems, applications, and services, pages 270–283, New York, NY, USA, 2004. ACM.

[16] Deborah Estrin Thanos Stathopoulos, John Heidemann. A remote code update mechanism for wireless sensor networks. Technical report, November 26 2003.

[17] Niels Reijers and Koen Langendoen. Efficient code distribution in wireless sensor networks. In *WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 60–67, New York, NY, USA, 2003. ACM.

[18] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. pages 354 – 365, jan.-2 feb. 2005.

[19] Lui Sha, Ragunathan Rajkumar, and Michael Gagliardi. Evolving dependable real-time systems. In *IEEE Aerospace Applications Conference*, pages 335–346, 1995.

[20] Chih-Chieh Han, Ram Kumar, Roy Shea, and Mani Srivastava. Sensor network software update management: a survey. *Int. J. Netw. Manag.*, 15(4):283–294, 2005.

[21] John A. Stankovic, Chenyang Lu, Lui Sha, Tarek Abdelzaher, and Jennifer Hou. Real-time communication and coordination in embedded sensor networks. In *Proceedings of the IEEE*, pages 1002–1022, 2003.

[22] Ting Liu, Christopher M. Sadler, Pei Zhang, and Margaret Martonosi. Implementing software on resource-constrained mobile sensors: experiences with impala and zebranet. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 256–269, New York, NY, USA, 2004. ACM.

[23] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *NSDI'04: Proceedings of the 1st conference on Symposium*

on Networked Systems Design and Implementation, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[24] P. Devanbu, M. Gertz, and S. Stubblebine. Security for automated, distributed configuration management. In *In Proceedings, ICSE 99 Workshop on Software Engineering over the Internet*, 1999.

[25] G. Racherla and D. Saha. Security and privacy issues in wireless and mobile computing. pages 509 –513, 2000.

[26] L. Sha. Upgrading real-time control software in the field. *Proceedings of the IEEE*, 91(7):1131 – 1140, july 2003.

[27] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11 – 33, jan.-march 2004.

[28] Gaul libraries reference guide, at: http://gaul.sourceforge.net/gaul_reference_guide-0.1847-5.htm.

[29] C. Brandolese and W. Fornaciari. A framework for compile-time and run-time management of non-functional aspects in wsns nodes. In *Proceedings of the 12th EUROMICRO Conference on Digital System Design*, Patras, Greece, Sep. 2009.

[30] J. Jeong, J. Kim, and A Board. Network reprogramming. 2003.

[31] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA, 2004. ACM.

[32] Prabal K. Dutta, Jonathan W. Hui, David C. Chu, and David E. Culler. Securing the deluge network programming system. In *IPSN '06: Proceedings of the 5th international conference on Information processing in sensor networks*, pages 326–333, New York, NY, USA, 2006. ACM.

[33] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. Mantis: system support for multimodal networks of in-situ sensors. In *WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 50–59, New York, NY, USA, 2003. ACM.

[34] Jan Beutel. Fast-prototyping using the btnode platform. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 977–982, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[35] Jaein Jeong and D. Culler. Incremental network programming for wireless sensors. pages 25 – 33, oct. 2004.

[36] J. Lilius and I. Paltor. Deeply embedded python, a virtual machine for embedded system. Technical report, Turku Centre for Computer Science at: http://tucs.fi/magazin/output.php?ID=2000.N2.LilDeEmPy.

[37] A. Boulis and M.B. Srivastava. A framework for efficient and programmable sensor networks. pages 117 – 128, 2002.

[38] D. Janakiram, R. venkateswarlu, and J. Nitin. Comis: Component oriented middleware for sensor networks. In *LANMAN '05: Proceedings of the 14 th International Workshop on Local and Metropolitan Area networks*, Washington, DC, USA, 2005. IEEE Computer Society.

[39] Liviu Iftode, Cristian Borcea, Andrzej Kochut, Chalermek Intanagonwiwat, and Ulrich Kremer. Programming computers embedded in the physical world. *International Standard ISO*, 8327, 1987.

[40] René Müller, Gustavo Alonso, and Donald Kossmann. Swissqm: Next generation data processing in sensor networks. In *CIDR*, pages 1–9, 2007.

[41] René Müller, Gustavo Alonso, and Donald Kossmann. A virtual machine for sensor networks. In *EuroSys '07: Proceedings of the 2nd ACM*

SIGOPS/EuroSys European Conference on Computer Systems 2007, pages 145–158, New York, NY, USA, 2007. ACM.

[42] Philip Levis and David Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95, New York, NY, USA, 2002. ACM.

[43] Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 15–28, New York, NY, USA, 2006. ACM.

[44] Hongzhou Liu, Tom Roeder, Kevin Walsh, Rimon Barr, and Emin Gün Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 149–162, New York, NY, USA, 2005. ACM.

[45] Joel Koshy and Raju Pandey. Vmstar: synthesizing scalable runtime environments for sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 243–254, New York, NY, USA, 2005. ACM.

[46] P. Buonadonna, D. Gay, J.M. Hellerstein, W. Hong, and S. Madden. Task: sensor network in a box. pages 133 – 144, jan.-2 feb. 2005.

[47] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[48] P. Atzeni, S. Ceri, P. Fraternali, S. Paraboschi, and R. Torlone. *Basi di dati Architetture e linee di Evoluzione*. McGraw-Hill, 2007.

[49] Wai Fu Fung, David Sun, and Johannes Gehrke. Cougar: the network is the database. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 621–621, New York, NY, USA, 2002. ACM.

[50] Chavalit Srisathapornphat, Chaiporn Jaikaeo, and Chien-Chung Shen. Sensor information networking architecture. In *ICPP '00: Proceedings of the 2000 International Workshop on Parallel Processing*, page 23, Washington, DC, USA, 2000. IEEE Computer Society.

[51] Jan Beutel, Matthias Dyer, Lennart Meier, Matthias Ringwald, and Lothar Thiele. Next-generation deployment support for sensor networks. In *207. Computer Engineering and Networks Lab, Swiss Federal Institute of Technology*, 2004.

[52] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, pages 653–662, Washington, DC, USA, 2005. IEEE Computer Society.

[53] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Towards a flexible global sensing infrastructure. *SIGBED Rev.*, 4(3):1–6, 2007.

[54] L. Szumel, J. LeBrun, and J. D. Owens. Towards a mobile agent framework for sensor networks. In *EmNets '05: Proceedings of the 2nd IEEE workshop on Embedded Networked Sensors*, pages 79–87, Washington, DC, USA, 2005. IEEE Computer Society.

[55] Takeshi Umezawa, Ichiro Satoh, and Yuichiro Anzai. A mobile agent-based framework for configurable sensor networks. In *MATA '02: Proceedings of the 4th International Workshop on Mobile Agents for Telecommunication Applications*, pages 128–140, London, UK, 2002. Springer-Verlag.

[56] Chih chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. Sos: A dynamic operating system for sensor networks. In *Proceedings of the Third International Conference on Mobile Systems, Applications, And Services (Mobisys*. ACM Press, 2005.

[57] Joshua Lifton, Deva Seetharam, Michael Broxton, and Joseph A. Paradiso. Pushpin computing system overview: A platform for distributed, embedded, ubiquitous sensor networks. In *Pervasive '02: Proceedings of the First International Conference on Pervasive Computing*, pages 139–151, London, UK, 2002. Springer-Verlag.

[58] Jochen Schiller, Achim Liers, Hartmut Ritter, Rolf Winter, and Thiemo Voigt. Scatterweb - low power sensor nodes and energy aware routing. In *HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, page 286.3, Washington, DC, USA, 2005. IEEE Computer Society.

[59] Pedro Jos Marrn, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *In Proceedings of the Third European Workshop on Wireless Sensor Networks (EWSN 2006*, pages 212–227, 2006.

[60] TIS Committee. Tool interface standard (TIS) executable and linking format (ELF) specification, May 1995. Version 1.2.

[61] Mark Wilding and Dan Behman. *Self-service Linux: mastering the art of problem determination*. Bruce Perens' Open Source series. 2006.

[62] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003. ACM.

[63] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with cooja. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, pages 641 –648, 14-16 2006.

[64] Jiming Chen, Jianhui Zhang, Weiqiang Xu, Lei Shu, and Youxian Sun. The development of a realistic simulation framework with omnet++. In *Future Generation Communication and Networking, 2008. FGCN '08. Second International Conference on*, volume 1, pages 497 –500, 13-15 2008.

[65] C. Hanle and M. Hofmann. Performance comparison of reliable multicast protocols using the network simulator ns-2. In *Local Computer Networks, 1998. LCN '98. Proceedings., 23rd Annual Conference on*, pages 222 –237, 11-14 1998.

[66] I.S. Hammoodi, B.G. Stewart, A. Kocian, and S.G. McMeekin. A comprehensive performance study of opnet modeler for zigbee wireless sensor networks. In *Next Generation Mobile Applications, Services and Technologies, 2009. NGMAST '09. Third International Conference on*, pages 357 –362, 15-18 2009.

[67] B.L. Titzer, D.K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, pages 477 – 482, 15 2005.

[68] J. Polley, D. Blazakis, J. McGee, D. Rusk, and J.S. Baras. Atemu: a fine-grained sensor network simulator. In *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on*, pages 145 – 152, 4-7 2004.

[69] Lewis Girod, Nithya Ramanathan, Jeremy Elson, Thanos Stathopoulos, Martin Lukac, and Deborah Estrin. Emstar: A software environment for developing and deploying heterogeneous sensor-actuator networks. *ACM Trans. Sen. Netw.*, 3(3):13, 2007.

[70] Sung Park, Andreas Savvides, and Mani B. Srivastava. Sensorsim: a simulation framework for sensor networks. In *MSWIM '00: Proceedings of the 3rd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, pages 104–111, New York, NY, USA, 2000. ACM.

[71] G.V. Merrett, N.M. White, N.R. Harris, and B.M. Al-Hashimi. Energy-aware simulation for wireless sensor networks. In *Sensor, Mesh and Ad Hoc Communications and Networks, 2009. SECON '09. 6th Annual IEEE Communications Society Conference on*, pages 1 –8, 22-26 2009.

[72] S. Sundresh, Wooyoung Kim, and G. Agha. Sens: a sensor, environment and network simulator. In *Simulation Symposium, 2004. Proceedings. 37th Annual*, pages 221 – 228, 18-22 2004.

[73] A. Köpke, M. Swigulski, K. Wessel, D. Willkomm, P. T. Klein Haneveld, T. E. V. Parker, O. W. Visser, H. S. Lichte, and S. Valentin. Simulating wireless and mobile networks in omnet++ the mixim vision. In *Simutools '08: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, pages 1–8, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[74] Milos Blagojevic, Majid Nabi, Teun Hendriks, Twan Basten, and Marc Geilen. Fast simulation methods to predict wireless sensor network performance. In *PE-WASUN '09: Proceedings of the 6th ACM symposium on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks*, pages 41–48, New York, NY, USA, 2009. ACM.

[75] Thiemo Voigt, Joakim Eriksson, Fredrik Österlind, Robert Sauter, Nils Aschenbruck, Pedro J. Marrón, Vinny Reynolds, Lei Shu, Otto Visser, Anis Koubaa, and Andreas Köpke. Towards comparable simulations of cooperating objects and wireless sensor networks. In *VALUETOOLS '09: Proceedings of the Fourth International ICST Conference on Performance Evalua-*

tion Methodologies and Tools, pages 1–10, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[76] Jianming Zhang, Yaping Lin, Cuihong Zhou, and Jingcheng Ouyang. Optimal model for energy-efficient clustering in wireless sensor networks using global simulated annealing genetic algorithm. In *Intelligent Information Technology Application Workshops, 2008. IITAW '08. International Symposium on*, pages 656 –660, 21-22 2008.

[77] Sajid Hussain, Abdul W. Matin, and Obidul Islam. Genetic algorithm for energy efficient clusters in wireless sensor networks. In *Information Technology, 2007. ITNG '07. Fourth International Conference on*, pages 147 –154, 2-4 2007.

[78] Hyun-Sik Seo, Se-Jin Oh, and Chae-Woo Lee. Evolutionary genetic algorithm for efficient clustering of wireless sensor networks. In *Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE*, pages 1 –5, 10-13 2009.

[79] E. Heidari and A. Movaghar. Intelligent clustering in wireless sensor networks. In *Networks and Communications, 2009. NETCOM '09. First International Conference on*, pages 12 –17, 27-29 2009.

[80] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[81] Kay Rmer, Philipp Blum, and Lennart Meier. Time synchronization and calibration in wireless sensor networks. In *Handbook of Sensor Networks: Algorithms and Architectures*. 2005.