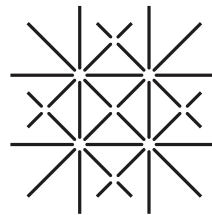


Dynamic Code Morphing in Network Embedded Systems



UNI
BASEL

Inauguraldissertation

zur Erlangung der Würde eines Doktors der Philosophie vorgelegt
der Philosophisch-Naturwissenschaftlichen Fakultät
der Universität Basel

von

Igor A. Talzi

aus Sankt-Petersburg, Russische Föderation

Basel, Switzerland, 2011



Attribution-Noncommercial-No Derivative Works 2.5 Switzerland

You are free:



to Share — to copy, distribute and transmit the work

Under the following conditions:



Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).



Noncommercial. You may not use this work for commercial purposes.



No Derivative Works. You may not alter, transform, or build upon this work.

- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.

Your fair dealing and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code (the full license) available in German:
<http://creativecommons.org/licenses/by-nc-nd/2.5/ch/legalcode.de>

Disclaimer:

The Commons Deed is not a license. It is simply a handy reference for understanding the Legal Code (the full license) — it is a human-readable expression of some of its key terms. Think of it as the user-friendly interface to the Legal Code beneath. This Deed itself has no legal value, and its contents do not appear in the actual license. Creative Commons is not a law firm and does not provide legal services. Distributing of, displaying of, or linking to this Commons Deed does not create an attorney-client relationship.

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät
auf Antrag von

1. Prof. Dr. Christian Tschudin (supervisor)

2. Prof. Dr. Gustavo Alonso (co-referee)

Day of the faculty meeting: 21/06/2011

3. Prof. Dr. Martin Spiess (dean)

Signature from the head of PhD committee:

Basel, Switzerland, 2011

Dynamic Code Morphing in Network Embedded Systems

Copyright © 2011 by Igor A. Talzi



This document is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives (CC BY-NC-ND) Works 2.5 Switzerland License, whose full text can be found on the following web-page:

<http://creativecommons.org/licenses/by-nc-nd/2.5/ch/legalcode.de>

Original document stored on the publication server of the University of Basel:

edoc.unibas.ch

To W and M, with love and remembrance forever.

“The belief that there was nothing and nothing happened to nothing and then nothing magically exploded for no reason, creating everything, and then a bunch of everything magically rearranged itself for no reason whatsoever into self-replicating bits which then turned into dinosaurs.

Makes perfect sense”.

[Unknown author]

“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil”.

[Donald Knuth]

“Do not optimize a running program, never”.

[Christian Tschudin]

(during a brainstorming session)

Abstract

The use of mobile code in embedded, resource limited systems like Wireless Sensor Networks (WSN) is an opportunity and a challenge at the same time. The opportunity lies in the dynamic re-tasking and run-time adaptation that can considerably extend the functional envelope of the deployed hardware. The price to pay, however, is additional communication overhead that results in shorter lifetime and decreased performance. That is why the system's scarce resources must be utilized with even more efficiency than usual. But optimization methods applied at design-time lead to case-restricted solutions, or the dismissal of mobile code solutions altogether.

In this work, we challenge ourselves with an integrated design of a system that addresses the increase of communication overhead by **online code compression**. The main task of the proposed method is to extract semantics from the transmitted mobile code at run-time and to tie it to the on-node holder, a dictionary of some type, avoiding costly code (re-)transmissions. By doing so, the actual code representation is brought to a near optimal form for each specific task covering both, the dynamic re-tasking and the reduction of communication overhead.

The distinctive feature of the method is that it can adapt to the changes in code structure, code content, and regional usage patterns at run-time without interruption of system operation. The low computational complexity of the method allows to use it in resource-constrained devices like WSN and to implement time-sensitive applications.

Keywords: Wireless Sensor Networks, Virtual Machines, Resource-Aware Protocols, Run-Time Optimization, Code Compression

Funding: The work was supported (partly) by the *National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS)*, a center supported by the *Swiss National Science Foundation* under grant number 5005-67322. We also acknowledge substantial support from the *Swiss Federal Office for the Environment (FOEN)*.

Acknowledgements

This thesis has more than one author despite what is stated on the front page. Many people have contributed to this work, directly or indirectly. I would like to believe that all those people are acknowledged here. My gratitude is expressed as follows.

First, I would like to thank my supervisor, *prof. Christian Tschudin*, who gave me an opportunity to carry out this work for many years and who did not lose his faith in me until the end. He was supporting me during all the ups and downs of a typical PhD path and was inspiring with his positive view on things. I have learned that there are no unsolvable problems; it is all about of how you approach a new challenge. Our brainstorming sessions were always fun with useful and sometimes crazy outcome.

Special thanks go to my partners from the *PermaSense* project, the research I was involved in for several years: *Andreas Hasler, Stephan Gruber, Jan Beutel, Roman Lim, Mustafa Yücel, Daniel Vonder Mühl, Christian Plessl, Sandro Schönborn, prof. Lothar Thiele, and Hansueli Gubler*. You all are very nice guys, and it is been a real pleasure to work with you and to share time in the beautiful Swiss Alps.

I also have to mention the very nice staff of *Shockfish SA*: *Roger Meier* and *Maxime Muller*. You were very helpful, and I appreciate your great expertise and confidence.

I am grateful to my colleges at the *Department of Mathematics and Computer Science*. Especially *Thomas Meyer* who has assisted with many aspects of the current work. Content-related discussions and million typesetting suggestions,

you gave me, have made this thesis look much better than it would have without your help. Your passion and perfectionism are inspiring. *Christopher Jelger* has also contributed a lot. Thank you for your advice and guidance at the time I needed it.

Nothing would have happened without my folks who gave me a chance to always remain myself, even if this was contrary to their own wishes, and let me be free in my choices. The “freedom” is the best word in all languages.

The last on this list but the first in my life are *W* and *M*. Without you and your support nothing would have been possible. *M*, your editing efforts are priceless even if many people find your Irish expressions a bit odd. *W*, whatever happens You will always be my little boy, and nobody will ever replace You or take You away from me. I do not believe in the afterlife, but I do believe that soul mates never die.

Glossary

AC – Algorithmic Complexity or Autonomic Component or Artificial Chemistry

AIT – Algorithmic Information Theory

ARQ – Automatic Repeat reQuest

AS – Alphabet Size

ASIC – Application Specific Integrated Circuit

CCN – Content-Centric Network(ing)

CF – Compression Factor

CISC – Complex Instruction Set Computer

CNP – Chemical Network(ing) Protocol

CPU – Central Computational Unit

CRC – Cyclic Redundancy Check

DCN – Data-Centric Network(ing)

DSP – Digital Signal Processor

DTN – Delay-Tolerant Network(ing)

ELF – Executable and Linkable Format

ES – Embedded System

FCS – Free Code Space

FL – Fragment Length

FN – Fragments Number

FEC – Forward Error Correction

FPGA – Field Programmable Gate Array

GPRS – General Packet Radio Service

GPS – Global Positioning System

IP – Intellectual Property core or Internet Protocol

ISA – Instruction Set Architecture

KC – Kolmogorov Complexity

KR – Kolmogorov Randomness

LAN – Local Area Network

LoMA – Law of Mass Action
MAC – Media Access Control
MANET – Mobile Ad-hoc
NETwork
MCU – Micro-Controller Unit
MPLS – Multi-Protocol Label
Switching
NoC – Network-on-a-Chip
OS – Operating System
PET – Program Execution
Table
PS – Program Size
RAM – Random Access
Memory
RISC – Reduced Instruction
Set Computer
ROM – Read-Only Memory
SBTSP – Skew Balance Time
Synchronization Protocol
SoC – System-on-Chip
SQL – Structured Query
Language
TDMA – Time Division
Multiple Access
UTC – Universal Time Clock
VLIW – Very Long
Instruction Words
VLSI – Very Large Scale
Integration
VM – Virtual Machine
VoIP – Voice over IP
VS – Virtual Segment(ation)
WSN – Wireless Sensor
Network
XML – eXtensible Markup
Language

Contents

Abstract	ix
Acknowledgments	xi
Glossary	xiii
Table of Contents	xv
1 Introduction	1
1.1 Embedded Systems	2
1.2 WSN – Network Embedded System	3
1.3 State-Of-The-Art in WSN	5
1.3.1 Hardware Platforms	6
1.3.2 Software Platforms	6
1.4 Example: PermaSense Project	10
1.5 Motivation and Problem Statement	14
1.5.1 Virtual Segmentation	14
1.5.2 Task-Oriented Network Morphing	15
1.5.3 Embedded Stack Composition	15
1.5.4 Dynamic Code Morphing	16
1.6 Contributions	17
1.7 Roadmap	18
1.8 Summary	19
2 On Present and Future Network Models	21
2.1 Data Packet Networks	22
2.2 Active Networking and Mobile Code	22

Contents

2.3	Autonomic Architectures	25
2.3.1	Self-Organizing Networks	25
2.3.2	Self-Healing Protocols	26
2.3.3	Self-Optimizing Systems	27
2.4	Chemical Networking and Fraglets	28
2.5	Unconventional Networking Styles	29
2.5.1	Delay-Tolerant Networking	29
2.5.2	Gossip Protocols	29
2.5.3	Content-Centric Networks	30
2.6	Summary	31
3	Configurable VMs for Embedded Networking	33
3.1	Existing Solutions for WSN Morphing	34
3.1.1	Frameworks for Re-Programming	40
3.1.2	Frameworks for Re-Tasking	40
3.1.3	Netware	44
3.2	ChameleonVM	49
3.2.1	System Architecture	54
3.2.2	Programming Model	57
3.2.3	Execution Model	62
3.2.4	Instruction Set	63
3.2.5	Code Verification	65
3.2.6	Code Propagation and Deployment	66
3.2.7	Packet Processing	68
3.2.8	Node-To-Node Communication	69
3.2.9	On-Board Dictionary	69
3.2.10	Dictionary Updates and Synchronization	72
3.2.11	Compression and Decompression	76
3.2.12	Support for Existing (External) Software	76
3.2.13	Exported Functions	78
3.3	FragletVM	78
3.3.1	Fraglets Language	79
3.3.2	System Architecture	82
3.3.3	Programming Model	85
3.3.4	Execution Model	86
3.3.5	Instruction Set	89
3.3.6	Code Propagation and Deployment	90
3.3.7	Node-To-Node Communication	90

3.3.8	On-Board Dictionary	91
3.3.9	Dictionary Updates and Synchronization	92
3.3.10	Compression and Decompression	94
3.4	Implementation Remarks	94
3.5	Summary	94
4	Overview of Code Optimization Techniques	95
4.1	Code Shrinking	96
4.2	Code Compression	99
4.2.1	Primitive Methods	103
4.2.2	Transform-based Encoding	104
4.2.3	Prediction-based Encoding	104
4.2.4	Dictionary Encoders	105
4.2.5	Entropy Encoding	108
4.2.6	Distributed Source Coding	111
4.2.7	Compression in Embedded Systems	111
4.3	Code Polymorphism	113
4.4	Code Versioning and Lifecycle	115
4.5	Code Robustness	117
4.6	Summary	122
5	Online Code Compression Framework	123
5.1	Kolmogorov Complexity of Code Streams	124
5.1.1	Translation of KC for a Common Compression	125
5.1.2	Translation of KC for Online Code Compression	125
5.2	Analysis of Existing Solutions	126
5.2.1	Data Compression Techniques for Code	128
5.2.2	Compression of Native vs Bytecode	129
5.2.3	Instruction Set Compression	129
5.2.4	User-Definable ISA	133
5.3	Requirements	134
5.4	System Architecture	136
5.4.1	Pair-Wise Stream Search	137
5.4.2	Selection Algorithm	139
5.4.3	Pairs vs Multi-Sequences	142
5.4.4	Continuous vs Fragmented Code Stream	143
5.4.5	Data and Code Mix	143
5.4.6	Dictionary Sub-Classing	144

Contents

5.4.7	Instruction Nesting and Unfolding	145
5.4.8	Parameters	146
5.4.9	Algorithm Speed	147
5.4.10	Algorithm Complexity	148
5.4.11	Reference Compression Methods	148
5.4.12	Test Setting For Random Code Streams	148
5.5	Single-Node Compression Method	149
5.6	Group Compression	158
5.7	Distributed Compression	158
5.8	Cloud Compression	159
5.8.1	General Setting	159
5.8.2	Quasi-Real Setting	161
5.9	Convergence Speed	164
5.10	Summary	165
6	Experimental Setup	177
6.1	Overview	178
6.2	Test Settings	179
6.2.1	Code Pre-Processing	181
6.3	Energy Model	182
6.4	Optimizing ChameleonVM's Code	184
6.4.1	Mobile Code Version of "Hello World!"	184
6.4.2	Route Discovery	188
6.4.3	Spanning Tree	192
6.4.4	Count the Number of Nodes	196
6.4.5	ID Assignment	200
6.4.6	Skew-Balance Time Synchronization Protocol	203
6.4.7	Data Collection Application	210
6.5	Optimizing FragletVM's Code	215
6.5.1	Disperser	215
6.6	Optimizing Foreign Code: Fire Tracker in Agilla	221
6.7	Final Considerations	224
6.8	Summary	227
7	Discussion and Future Directions	229
7.1	Formal Evaluation of the Proposed Method	230
7.2	Application Fields	230
7.3	Open Questions	232

7.4	Future Directions	235
7.5	Summary	240
8	Conclusions	241
	List of Figures	247
	List of Tables	251
	List of Listings	253
A	Pseudo- and Program-Code for SBTSP	255
B	Pseudo- and Program-Code for Fire-Tracking using Agilla	265
C	ChameleonVM's Basic Instruction Set	269
D	FragletVM's Reduced Instruction Set	273
	Bibliography	277
	Curriculum Vitae	283

1

Introduction

We begin this work with giving a quick overview of embedded systems (ES) and their distinguishing features. We are particularly interested in the special network-oriented sub-class – Wireless Sensor Networks (WSN). We discuss the state-of-the-art in hardware and software support for WSN. We refer to the environment monitoring project PermaSense as an example, one which we have contributed to and which has been a key factor in the motivation for this research. We then state the main research challenge of the current work as to create a model of building task-optimized configurations for ES at run-time. Our assumption is that this goal can be achieved through dynamic code morphing. We have a closer look at the concepts of code morphing, stack composition and Virtual Segmentation. Code compression turns out to be an essential part of task optimization process. Finally, we briefly outline our contributions to this work and provide a roadmap for the rest of the document.

1.1 Embedded Systems

An embedded computer system is normally designed to implement one or a few specialized functions. This is somehow dictated by the fact that an Embedded System (ES) is traditionally associated with a number of constraints such as limited computational (memory, CPU) and energy resources (battery), high requirements on reliability (some systems must be able to cope with harsh environments), real-time support, etc. Note that usually not all of the limitations above exist. For example, if an ES is AC-powered, there is obviously no power constraint. Another example would be wireless sensor networks, discussed later in Section 1.2, which generally do not provide support for real-time operation. Every ES is dedicated to a specific task and that task poses a certain level of constraints. For example, WSN feature radio chips with a very short reliable range of 20–30 meters. Constraints vary widely. Even within a little sub-class of ES there might be a huge variation of available resources on different platforms (e.g., compare two WSN platforms *Mica* and *iMote2* from Table 1.1, Section 1.3.1). And, therefore, a system can be optimized in terms of reducing the size and cost and increasing reliability and performance in various ways.

An ES is specified using a set of hardware and software components it incorporates. The hardware part normally consists of a microprocessor (plus an optional DSP-core), memory unit, Flash and a set of specialized controllers. The controllers may include communication units (e.g., radio, serial interfaces, etc.), actuators, and sensing elements. Often and most commonly components such as CPU, memory and I/O ports are integrated into one chip. This significantly reduces form-factor, energy budget, and cost. Typical examples of this approach are microcontrollers (MCU), ASIC/FPGA arrays and SoC. The software part (often referred to as **firmware**) provides support for the hardware components and makes the system work as a whole. The type of the software used depends on the level of abstraction and interactivity the system must provide. It can be just a set of independent hardware drivers (simple control loop or interrupt controlled system) or a complex distributed network-oriented operating system with support for preemptive multi-tasking. We discuss the state-of-the-art in hardware and software for WSN in Section 1.3.

1.2 WSN – Network Embedded System

The variety of ES is huge. They range from small portable devices (e.g., PDA, GPS receivers) to large stationary installations (e.g., traffic light control). In this work, we limit our research to the very special type of ES called Wireless Sensor Networks (WSN). These miniature devices (sometimes referred to as **motes**) add two main distinctive features to the design of ES: 1) support for wireless radio communication, and 2) sensors. This enables a new way of creating monitoring and control systems. The latest trends show a rising interest to incorporate actuators into motes design. With this, the passive monitoring systems of today will become reactive to changes in the environment.

Motes are completely self-contained devices featuring the hardware architecture shown in Figure 1.1.

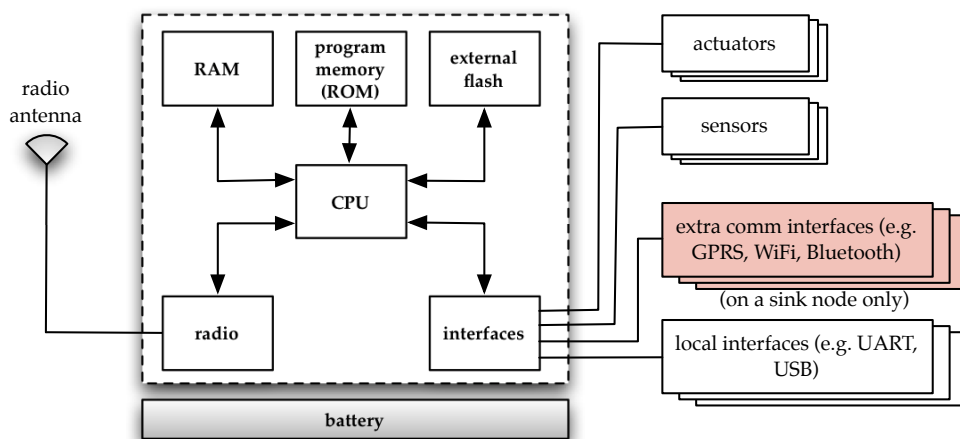


Figure 1.1: Typical Hardware Architecture of a WSN Mote

All central components (ROM, RAM, CPU) are integrated into the microcontroller. All peripheral parts are connected to it via I/O ports. These include: radio chip, extra Flash (to store measurements), sensors, actuators, additional communication interfaces, indicators. Typically, a WSN mote is battery operated but many platforms provide an AC-supply interface too. Hardware design allows motes to operate on a pair of single AA-batteries for years (at 1% duty cycle *TelosB* can last for almost three years, *Mica2* mote for one and a half, *MicaZ* for

1. Introduction

one; see Table 1.1). Energy harvesting methods [Mos09] can prolong it even further. Future architectures will probably integrate all parts including radio, sensors/actuators and power source in one chip thus making the initial concept of *Smart Dust* [WLLP01] real. For the time being, nodes are still being composed mainly of independent components. Figure 1.2 demonstrates the sensor module used in *PermaSense* project (see Section 1.4). The node's architecture (for more details see [BGH⁺09]) reflects the typical one shown in Figure 1.1.

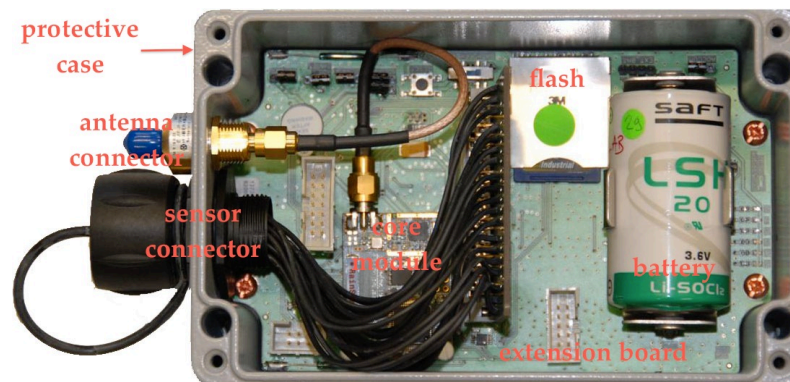


Figure 1.2: PermaDAQ SIB Module

The new ideology and the extra hardware components require a new level of software support. Motes are supposed to be organized in a network. The network must be able to collect and forward data (measurements) towards the access point in the network where it can be sucked out and post-processed. This point is often called a **sink** or a **gateway** node. There can be multiple sinks in a single network. The sink node normally features extra communication interfaces (Ethernet, GPRS, WiFi, etc.) to provide connection to the Internet. Alternatively, it can be directly connected to a PC. Also, a sink traditionally has more computational and power resources as it is supposed to process and transmit large amounts of data and stay active most of the time. Between regular motes, the situation is different. Since the provided radio communication range is relatively short (20–30 meters) networks have to be organized in a multi-hop fashion with all-to-one directional packet flow. In order to better utilize the available energy budget WSN normally work in a duty cycle mode: wake-up → sense → store locally

→ transmit/receive one hop → sleep. An example of one of the most commonly used WSN architecture is shown in Figure 1.3.

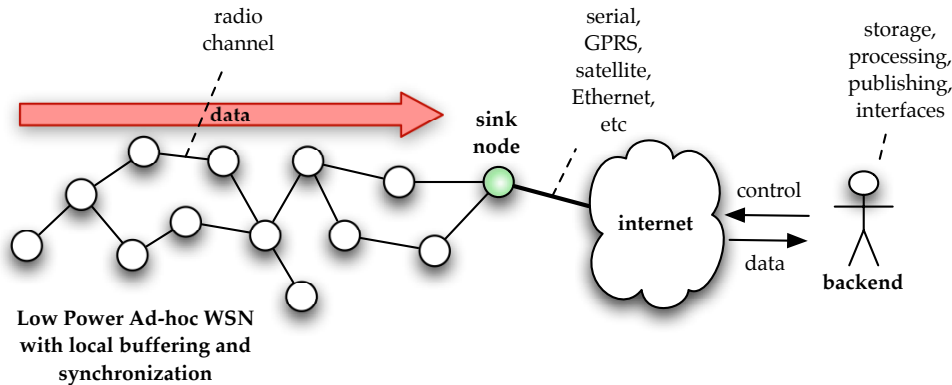


Figure 1.3: Typical WSN System Architecture

As Figure 1.3 shows, some advanced WSN feature the feedback channel providing control over the ad-hoc part of the system. Depending on the context it is used in, other unique characteristics of the WSN may include:

- the ability to withstand harsh environmental conditions,
- the ability to cope with communication and node failures,
- unattended operation,
- mobility of nodes and dynamic network topology,
- heterogeneity of nodes, and
- large and variable deployment scale.

WSN find application in environmental and habitat monitoring, industrial processes, machine health monitoring, health-care control, home automation, and traffic control, and others.

1.3 State-Of-The-Art in WSN

For nearly a decade WSN has been in active research. Many concepts in hardware and software design have been introduced, although no official standards exist yet. Here, we give an overview of the most notable representatives of hardware/software solutions for WSN domain of today.

1. Introduction

1.3.1 Hardware Platforms

Many prototype hardware WSN platforms have been proposed so far. Some of them are aimed at general research, others offer more specific features. What makes all those prototypes look similar is that their main architectural principle follows the one shown earlier in Figure 1.1. Hardware components used in those prototypes are normally cheap, mass production units. This keeps the price of the entire module low. A retrospective list of WSN hardware prototypes along with their characteristics is presented in Table 1.1.

The main trends in WSN platform design over time are: 1) form-factor and overall power consumption reduction, and 2) addition of new communication interfaces. In the past two years, more and more manufactures have focused on offering commercial OEM solutions: *ArchRock* (former *EPIC*), *Crossbow* (former *Mica* family, *TelosB*). These out-of-the-box solutions ease installation and deployment of WSN systems, functional principles remain the same.

1.3.2 Software Platforms

WSN is a relatively complex system, which requires proper software support in a form of an Operating System (OS), in order to be able to build distributed end-user applications for it. OS for WSN nodes are typically much simpler than general-purpose OS. This is caused by the specific requirements of WSN applications and the resource constraints in WSN hardware stated above in Section 1.3.1.

WSN hardware is very similar to traditional ES. The reason why the existing embedded OS, like *eCos* and *uC/OS*, are not commonly used on sensor nodes is that such OS are often designed with real-time properties in mind that are normally not required by WSN applications.

TinyOS [LMP⁺04] was the first OS specifically designed for WSN. It uses an event-driven programming model instead of multi-threading. *TinyOS* programs are constructed of event handlers and tasks. When an external event (e.g., incoming data packet, sensor reading, etc.) occurs, *TinyOS* calls the suitable handler to process it. Event handlers can post tasks that are scheduled by the *TinyOS* kernel for later execution. The *TinyOS* kernel, hardware drivers, and user programs for *TinyOS* are written in a special programming language called *nesC* [GLvB⁺03]. *NesC* is very likely an extension (clone) of the C programming language.

1.3 State-Of-The-Art in WSN

Platform	Microcontroller	Radio	Program + Data Mem	Ext Mem	Year
René	AT90LS8535	TR1000, 10 kbps, OOK, 916 MHz	0.5 kB RAM + 8 kB Flash	32 kB 24LC256, I ² C	1999
René2	ATmega163	same	1 kB RAM + 16 kB Flash	same	2000
Mica	ATmega128 (4 MHz 8 MIPS)	TR1000, 916 Mhz, ASK, 40 kbps, UART	4 kB RAM + 128 kB Flash	AT45DB041B, 512 kB, SPI	2001
Mica2	ATmega128L (8 MHz 8 MIPS)	Chipcon CC1000, 868/916 Mhz, FSK, 38.4 kbps, SPI	same	same	2002
MicaZ	ATmega128L	Chipcon CC2420, 2.4 Ghz, O-QPSK, 250 kbps, SPI, 802.15.4/ZigBee	same	same	2004
Telos	TI MSP430 (8 MHz)	same	10 kB RAM + 48 kB Flash	1024 kB ST M25P80, SPI	2004
iMote	ARM 7TDMI 12–48 Mhz	ZV4002 Bluetooth	64 kB SRAM	512 kB Flash	2004
BTnode rev3	ATmega128L	Chipcon CC1000, 433–915 Mhz + ZV4002 Bluetooth 2.4 Ghz	128 kB Flash ROM + 4 kB EEPROM	64+180 kB SRAM	2004
TelosB (Tmote Sky)	TI MSP430	Chipcon CC2420, 250 kbps, 2.4 GHz, IEEE 802.15.4	10 kB RAM + 48 kB Flash	1024 kB STM25P	2004
eyesIFXv2	TI MSP430	Infineon TDA5250, ASK/FSK, 868 Mhz, 64 kbps	same	4 Mb AT45DB041B	2005
Fleck	Atmega128L	Nordic nRF903, 433 Mhz, GFSK, 76 kbps		512 kB	2005
TinyNode-584	TI MSP430	Xemics XE1205, 868 Mhz, 153 kbps		512 kB Flash	2006
iMote2	Intel PXA271 + ARM 11–400 MHz	Chipcon CC2420 802.15.4/ZigBee	32 MB SDRAM	32 MB Flash	2006
SunSPOT Java	180 MHz 32-bit ARM920T	Chipcon CC2420, 2.4 GHz, IEEE 802.15.4	512 kB RAM + 4 Mb Flash		2006
IRIS	ATmega1281	Atmel AT86RF230 802.15.4/ZigBee	8 kB RAM	128 kB Flash	2008
EPIC	TI MSP430	Chipcon CC2420, 250 kbps, 2.4 GHz IEEE 802.15.4	10 kB RAM + 48 kB Flash		2008
Jcreate Sentilla	TI MSP430	Chipcon CC2420	10 kB RAM + 48 kB Flash		2008

Table 1.1: List of WSN Hardware Platforms

1. Introduction

NesC was designed to be able to detect race conditions between tasks, event handlers, and device drivers. Programs for *TinyOS* are compiled along with the OS kernel to form a single executable image.

There are also OS for WSN that allow programming in pure C. Examples of such operating systems include *ContikiOS*, *MantisOS*, *BTnut*, and *Nano-RK*.

ContikiOS [DGV04] was designed to support loading of modules over the network and, therefore, it offers run-time linking of standard ELF files [DFEV06b]. The *ContikiOS* kernel is event-driven, like *TinyOS*, but the system supports multi-threading on a per-application basis. Furthermore, *ContikiOS* includes **protothreads** that provide a thread-like programming abstraction but with a very small memory overhead.

Unlike the event-driven *ContikiOS* kernel, the *MantisOS* [BCD⁺05] and *Nano-RK* [ERR05] kernels are based on preemptive multi-threading. With preemptive multi-threading, applications do not need to explicitly yield the microprocessor to other processes. Instead, the kernel divides the time between the active processes.

Nano-RK is a real-time kernel that allows detailed control of how tasks get CPU time, networking and sensors.

Similar to *TinyOS* and *ContikiOS*, *SOS* [HRS⁺05] is an event-driven OS. Like *ContikiOS*, it supports for loadable modules. An entire program image is constructed of smaller modules, possibly at run-time. Thus, *SOS* provides support for dynamic memory management, as well.

BTnut [Beu06] is based on cooperative multi-threading and plain C code. It was specifically designed to support *BTnode* hardware platform.

LiteOS offers UNIX-like interface and support for C programming language. For instance, it allows end users to issue commands in the following manner: **ls** to ping the sensor network and to display the nodes, or **cp** to transfer data to or from sensor nodes.

Many other, less popular OS designs for WSN exist including *MagnetOS*, *AmbientRT*, *EYES/PEEROS* and others.

Above the OS level, a WSN software bundle is traditionally composed using the modular approach as shown in Figure 1.4. The MAC-layer on WSN platforms is implemented in software. The configuration of system services may differ and highly depends on the application's needs. An optional middleware provides the high level programming

abstractions for user applications. Those are created according to the underlying architecture using OS system calls or middleware abstractions.

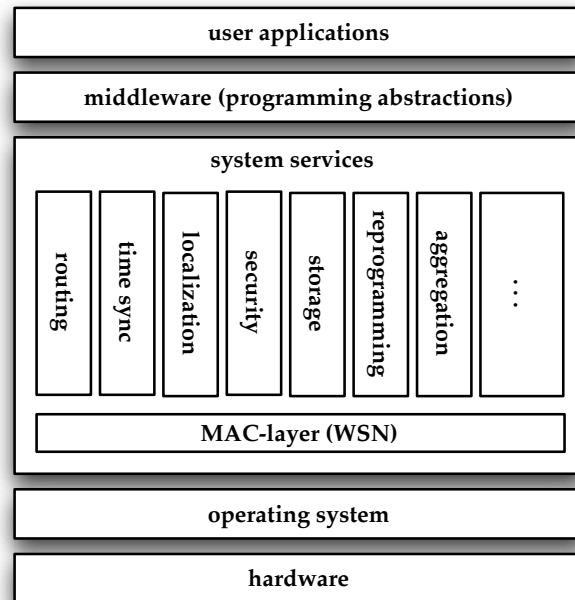


Figure 1.4: Software Suite on a Typical WSN Mote

Generally, there are several reasons, which have driven the software development for WSN beyond the OS level: limited energy budget, short communication range, large scale, and various deployment scenarios. Hence, algorithms and protocols for WSN are traditionally designed in a distributed fashion to address the following issues:

- multi-hop communication,
- lifetime maximization (data aggregation, power cycling, topology control),
- robustness and fault tolerance,
- self-configuration,
- security of data, and
- mobility of nodes.

1. Introduction

Which of the above or totally new issues to address is defined by the end-user application's needs and the context that application is used in. Addressing all of them is normally not possible because of the lack of enough available resources. As an example, in Section 1.4 we discuss a real-world environment monitoring project which has posed many of previously undiscovered challenges.

1.4 Example: PermaSense Project

The author of the present work has been involved in the environment monitoring project called *PermaSense* [THGT07] in which he was responsible for basic architecture design and performed several engineering tasks as well. Here, we would like to take this project as an example and show how such systems are built and how they work. This will help to understand what has motivated the research described in this thesis.

The main objective of the *PermaSense* project was to build and customize a set of wireless measurement units for use in remote areas with harsh environmental conditions. The second goal was the gathering of environmental data that help to understand the processes that connect climate change and rock fall in permafrost areas. To this end, several sensor fields are deployed and operated in the Swiss Alps over several years. Although our main contribution was made during the prototyping stage, we know the current state of the system quite well to discuss it here.

Wireless sensors enable monitoring of large and remote permafrost areas with spatially and temporally distributed measurements, leading to better predictions on the consequences of global warming for alpine regions. Beyond helping with the modeling of permafrost processes, this research is also applicable to natural hazard surveillance. The idea is to fill the niche of easy to deploy, stand-alone geo-monitoring and warning systems that are low-cost, cheap in maintenance and easily re-configurable when deployed. With better wireless sensor solutions, larger hazard areas can permanently be monitored and linked to warning systems that help to protect human lives.

The selected environment dictated the following unusual conditions:

- unattended operation for most of the time,

1.4 Example: PermaSense Project

- long-term deployment with extended node's lifetime (several years on the same power supply),
- extremely harsh weather accompanied by severe climatic variations, and
- highly variable link quality with a high chance of disconnected operation.

All these needed to be taken into account when choosing system architecture and hardware components, while designing protocols, and during the deployment stage. Technically, the system has all the features of a typical WSN (see Figure 1.3):

- periodical data sampling (a set of sensors attached to every node),
- to save power the operation is based on duty cycling,
- in-network time synchronization and correlation with UTC post-hoc,
- multi-hop communication between nodes,
- connected to the Internet via wireless GSM/GPRS channel,
- back-end with web-interface, visualization, database, logs and configuration scripts, and
- ability to send control commands from the back-end to the network.

Where does the project stand now? The project began in 2006 with the first prototypes deployed in the Jungfraujoch region of the Swiss Alps and field experience gathered in the summer of 2006 (see Figures 1.5a and 1.5c). Subsequently, a second deployment site was set up on the Matterhorn and a second generation wireless system deployed to both sides (see Figures 1.5b and 1.5d). From mid summer 2008, the *PermaSense* project has been actively collecting data. In the meantime, the engineering team from ETH Zürich has introduced many improvements to the initial setup. These include a modern system architecture¹ based on new hardware platforms with extended features [BGH⁺09], extra equipment such as weather stations, high resolution imaging systems [KYB09] to remotely monitor on-site operation, solar panels for renewable power supply, WiFi connection to the base station,

¹Please refer to <http://www.permasense.ch/technology/overview.html> for more up-to-date information.

1. Introduction

and installations on new sites. Recently, the project has evolved even further, changing its profile to provide support for similar projects in the area of sensing in extreme environments [BBF⁺11] and currently is known under the name of *X-Sense*.

To the best of our knowledge and based on the analysis of the two example installation sites in Figure 1.5 we can say that, besides many similarities in the underlying technology, these two setups have very different profiles. We can name the following:

- network size,
- used sensor types, and
- geo-topological location.

The first two parameters define the amount and type of data the system produces. The last one is responsible for shaping the data flow. Moreover, even within a single profile we can distinguish a big number of independent tasks. These tasks may include:

Sensor configuration: Multiple sensors can be attached to the same node. A set of sensors differs from node to node even at the same site (weather stations, visual cameras, geophysical sensors, GPS).

Node configuration: Multiple heterogeneous nodes co-exist in the network (sensor nodes, base station, core stations, video units). Additionally, node re-configuration may be caused by different environmental conditions within network partitions. For example, in case of Jungfraujoch site where two clouds are installed on the south and north slopes we might need to correct the node's clock drifts using different coefficients. And, this correction may have to change over a period of one day.

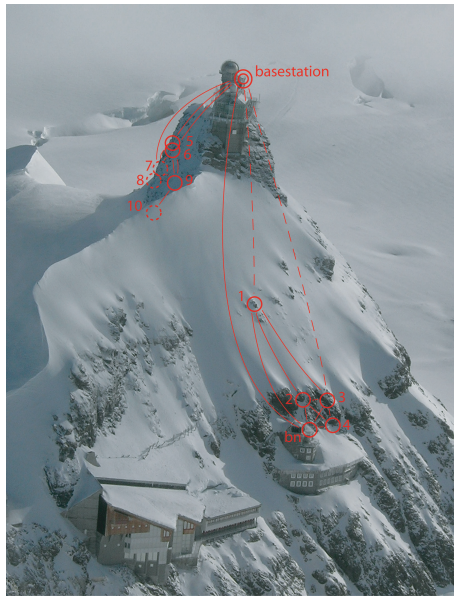
Data flow management: Generated data vary in a sample size, sampling rate, sample window. Therefore, we generate variable data volumes on different paths.¹

Interaction with a user.

These are some but not all possible tasks the system may have to perform. Depending on how carefully the system is designed the impact on data quality, data yield and network lifetime could be huge.

¹In [BBF⁺11] the authors call it **in-network data fusion** as we process data from different time and space scales.

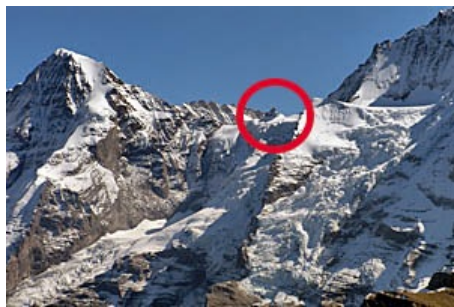
1.4 Example: PermaSense Project



(a) Jungfrauoch (1st deployment, since 2006/2007)



(b) Matterhorn (2nd deployment, since 2007)



(c) Sphinx Observatory at Jungfrauoch, 3500 m a.s.l.



(d) Hörnli ridge of Matterhorn, 3450 m a.s.l.

Figure 1.5: PermaSense Installations Sites in the Swiss Alps. Pictures and information derived from <http://www.permasense.ch/research/field-sites.html>

1.5 Motivation and Problem Statement

As it has been shown in Section 1.4 the research field of WSN is currently at the stage where multi-tasking becomes an integral part of the system's ideology. The multi-tasking is mainly caused by the need to manage the continuously growing hardware and software complexity. We believe that a new generation of WSN architectures is approaching where system operation will be viewed as a set of parallel and interacting network-wide tasks rather than a single-task process. This will eventually lead us to the point where task composition and task exchange will cause system profile to change dramatically over time, at run-time. And, the question of optimal profile representation in the context of limited WSN resources will be raised. Systems will have to find the most optimal configuration and encoding of multiple tasks in order to fit it into existing resource constraints.

In this work, we try to make an initial step towards task-specific optimization in ES at the code level. In the next few sections, we state the main research challenge of this work using the newly introduced terminology.

1.5.1 Virtual Segmentation

Virtual Segmentation (VS) is the process of dividing a physical network into so-called **profiles**.¹ Each profile describes a **task** (or a combination of tasks) in which every node in that cloud is supposed to act (see Figure 1.6a). Profiles can be rather complex; they can contain multiple independent tasks. Profiles can overlap on the physical topology, physically disconnected nodes can belong to the same profile.² In this case, a node becomes a member of two independent profiles bridging nodes between the two of them. For example, in Figure 1.6a one physical node takes a part in two profiles, *A* and *B*.

The example in Figure 1.6a is complex: multiple overlapping profiles, fully distributed peer-to-peer operations within each profile. In most experiments in this work, we consider a much simpler case where

¹Sometimes we also refer to them as **clouds**, but it has nothing to do with "cloud computing".

²Note that the concept of profiles is rather abstract. Programming the system in a way so that tasks are grouped together creates profiles. There is no "profile description language" available at the moment.

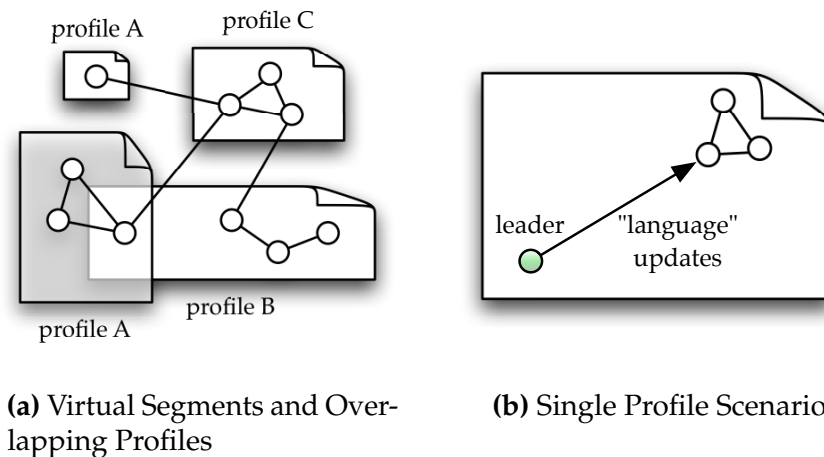


Figure 1.6: Virtual Segments and Network Profiles

we have a single profile with a dedicated leader as shown in Figure 1.6b. The **leader** is a regular node which is assigned to initiate and control all the operations and tasks within the profile. The complex scenario in Figure 1.6a can be decomposed into a set of interconnected simpler ones with each profile having the structure shown in Figure 1.6b.

Profiles can be added/removed to/from the system. They can be modified too. The system can switch between profiles at run-time or execute multiple profiles simultaneously.

1.5.2 Task-Oriented Network Morphing

The VS from Section 1.5.1 are the main building blocks of a task-oriented network tuning framework. The process of bringing a general network representation to a task-specific form we call **task-specific network morphing**. This process is continuous; it does not happen at compile-time only. The system is forced to find the most optimal representation by throwing out unnecessary elements, or adding missing ones at run-time.¹

1.5.3 Embedded Stack Composition

Although VS and network morphing can be seen from different angles including high-level programming paradigms in this work we push ourselves to the system level, the level of network stacks. WSN seems

¹Adding new functionality should normally be triggered by some sort of supervisor, or the system must know where to find the missing pieces.

1. Introduction

to be a perfect place to be in this case as all protocol layers (except for the physical and link layers) are fully re-programmable.

All network protocols share the same pieces of code between each other and between several protocol layers. For example, MAC, time-sync and routing use periodic beaconing as well as similar data structures and, therefore, similar system calls to the underlying OS. Between OS and middleware, we propose to use a so-called **netware** level which provides a set of commonly used network entities for constructing new protocol stacks. Those entities can be encoded in a very efficient way and shared by many targets. For instance, a Forward Error Correction (FEC) support could be provided as an external function, which is delivered and integrated into the protocol stack when bit error level exceeds a certain threshold. The system could even support several FEC schemes with different error detection and correction properties and switch between them depending on the current channel link quality. We call this process **stack composition**, the term originally used in [IT10]. The stack composition made for a specific task and in an on the fly manner can be seen as a sub-class of the network morphing introduced in Section 1.5.2. In Chapter 7 we refer to a special form of stack composition called **sponge protocols** the distinctive feature of which being a very fine granularity of building blocks – instead of modules, the system operates with single instructions which, however, may be quite complicated.

1.5.4 Dynamic Code Morphing

Finally, for each profile the system must be able to find an optimized (compressed) form. This is needed to make the whole design more energy and computationally efficient.

By having VS in a system we can optimize each profile (e.g., each stack) independently. We put the main focus of our research on how each profile can be brought to an optimal representation *at the code level*: mobile code implementing the profile (stack) is re-encoded at run-time in order to reduce its size and eventually the amount of transmitted code.

For example, if we use a centralized approach from Figure 1.6b to control the optimization process, a single leader is elected to run the compression and updates are sent around the network. Eventually, the system should learn and start using a better representation for the

code moving across the network; nodes agree on a new, compressed “language” on the fly. This learning process can be organized in a distributed fashion, but the main principle remains the same.

By doing so, we should be able to provide a positive effect on system’s lifetime. Additionally, the system should be able to gain some level of adaptability and autonomy. The rest of the document is dedicated to how this can be achieved.

1.6 Contributions

In this work, we make the following contributions to the area of dynamic code optimization techniques:

- We develop and implement two embedded execution environments for morphable mobile code. One of them, called *ChameleonVM*, employs traditional concepts of active networking and mobile code adapted for WSN domain; the other, called *FragletVM*, supports a new way of building network protocols using principles of artificial chemistry.
- We create and analyze a model of building task-specific configurations for WSN based on VS.
- We create a model for dynamic code optimization across task-tailored network segments.
- We analyze in every detail a part of the model concerning runtime code compression.
- We provide an experimental setting showing how our code optimization model could be applied to the code in real execution environments using two previously self-designed frameworks and code types plus we test it on a third-party code stream type.

By following the steps listed above we expect to retrieve a detailed image of what level of positive/negative impacts the run-time code optimization techniques may bring on overall performance and energy consumption characteristics of a network ES. The power consumption and wireless channel bandwidth utilization should enhance. At the same time, the run-time (de-)compression may cause some drop in performance and increase memory usage. With this work, we mainly contribute to the area of embedded network systems where the gain is expected to be the biggest. Standalone and traditional network (e.g.,

Internet) systems may not find our methodology much beneficial, as they do not feature the constraints that we address in this work.

1.7 Roadmap

This dissertation is organized into eight chapters.

Chapter 1: The chapter gives an introduction to the world of ES and its network-oriented sub-class called WSN. We discuss hardware and software design principles of WSN. We show in the example of an environment-monitoring project how WSN can be used for building a real application. We highlight the shortcomings of the current WSN architectures and explain what exactly has motivated our research. After that we state the main hypothesis.

Chapter 2: In this chapter, we try to briefly classify and analyze major current networking paradigms and draw some future directions. We locate our own work in this landscape.

Chapter 3: We start the chapter by discussing various methods of changing WSN behavior: re-configuration, re-programming, and re-tasking. We come up with a new model of creating task-specific WSN configurations using the concept of VS. We present two embedded execution environments, *ChameleonVM* and *FragletVM*, which are used to build task-tailored systems. We discuss, in detail, design and working principles of both.

Chapter 4: The ability to create task-specific setups brings us to the challenge of optimizing task representation. We explore how task representation could be changed at run-time and what implications this would have on the robustness of the system. We pay special attention to the part of the optimization process, which involves code compression.

Chapter 5: This chapter presents the online code compression framework which becomes an integral part of the design of both, *ChameleonVM* and *FragletVM*, execution environments. We explain its system architecture, working principle, and provide some performance metrics for various types of code streams and various network settings.

Chapter 6: We evaluate our online code compression scheme in several examples. We consider three code models: traditional active networking (*ChameleonVM*), chemical networking protocols (*FragletVM*), and a third-party model based on mobile agents.

Chapter 7: The chapter serves as a polygon for better understanding of what has been achieved with this work so far and to outline future possible directions.

Chapter 8: We make the final conclusions to our work and complete the picture.

In Appendices, we provide block-diagrams and source code of the algorithms and protocols used throughout the rest of the document, which are needed for better understanding of their working principles.

1.8 Summary

In the introduction, we have given an overview of ES and their special network sub-class called WSN. We have shown that characteristics of such systems pose many design issues resulting from the extremely limited resources. This we have demonstrated by giving references to a real-world project example. Having a clear understanding of the existing problems we have formulated one possible solution and outlined our research plan along with a list of major contributions. A brief introduction to the main concepts must help the reader with the rest of the document. We will operate with them throughout this work. Finally, the thesis structure must help to navigate through the document easily.

2

On Present and Future Network Models

Before unfolding the main part of our research, dynamic mobile code optimization, we discuss the state-of-the-art and several future promising paradigms in networking. This will help us to better position our contribution inside the global image and to better understand what we would like to achieve with the new method. We start this discussion with the classic packet-switching approach, which is the foundation of today's Internet technology. Following this, we give an overview of methodology known as active networking and the idea of mobile code, its main building block. We continue by referring to the concept of autonomic computing and autonomic architectures. As an example of this currently active, vast research domain we discuss Chemical Networking Protocols (CNP). To complete the picture we mention several non-traditional ways of networking like Delay-Tolerant Networks (DTN), gossip protocols, Content-Centric Networks (CCN) as well as networks with self-organized properties like WSN and MANET.

2.1 Data Packet Networks

Data packet networks came first. By saying that, we assume any type of network for which the functional principles are based on the idea of packet exchange. Packets carry two types of information: actual data and current state of the network (e.g., routing information, time, etc.). Network information is transmitted from node to node; it is analyzed and some re- or pro-active steps are taken to keep the system going.

Encapsulation of information in a packet allows for almost natural sharing of the same communication channel among multiple data flows and/or multiple data types (e.g., video, voice, text) between multiple nodes. The corresponding network infrastructure (routers, switches, etc.) decides how to regulate the packet traffic and how to deliver the information. The decision-making on how to deal with the traffic is fully decentralized. Moreover, often the data type is kept out of consideration in contrast to content-oriented approaches like the one described in Section 2.5.3. Hence, this approach might suffer from variable bit rates and long delays, although it allows to better utilizing the channel capacity. The technology responsible for this is called **packet switching**. The most well known use of packet switching is the Internet and Local Area Networks (LAN). Some other examples include Multi-Protocol Label Switching (MPLS) and, used in mobile phone technology, General Packet Radio Service (GPRS). In contrast, a second method, called **circuit switching**, sets up dedicated connections between particular nodes. This solution allows the provision of a requested quality of service (constant bit rate, constant delay) and thus is primarily used in digital telephony-related services (e.g., ISDN, GSM).

Although data packet networks still exist and are still the primary choice in many cases, they leave many questions open: maintenance, scalability, adaptability, protection against attacks, and many others. So, how can we deal with all of these?

2.2 Active Networking and Mobile Code

One of the answers to the above question was, and still is, **active networking**. It is a communication model that allows packets flowing

2.2 Active Networking and Mobile Code

through a network to dynamically modify the operation of the network.¹

Active network architecture is composed of execution environments (that can execute active packets, a combination of code and data) and of active hardware capable of routing or switching as well as executing code within active packets. This is different from the traditional network architecture, which increases robustness and stability by removing complexity and the ability to change its fundamental functions from underlying network components. The network components remain passive blocks with pre-defined functionality. **Network processors** are one means of implementing active networking concepts. Active networks have also been implemented as overlay (software-only) networks. The most famous proposals in this area are *ANTS* [WGT98] and *PAN* [NGK99].

Active networking brings the possibility of highly tailored and rapid “real-time” changes to the underlying network operation. This enables such ideas as sending code along with information packets. This allows the data to change its form (code) to match the channel characteristics. We discuss what is the smallest program that can generate a sequence of data in Section 5.1, when we speak about the definition of **Kolmogorov Complexity (KC)**. The use of real-time genetic algorithms within the network to compose network services is also enabled by active networking (see Section 7.2).

One of the biggest challenges with active networks is how to optimally allocate computation versus communication within networks. A similar problem related to the compression of code as a measure of complexity is addressed via **Algorithmic Information Theory (AIT)** which we will also discuss in Section 5.1.

Closely connected to active networking is the concept of **mobile code**. Although mobile code can support different paradigms, from Java applets (**code on demand**) to grid computing (**remote evaluation**), here we relate it to the idea of **mobile agents**.² A mobile agent is a composition of code and data, which is able to migrate (move) from one node to another autonomously keeping its state (able to continue its

¹The text below partially cites and is based on the Wikipedia article retrieved from http://en.wikipedia.org/wiki/Active_networking.

²Sometimes also referred to as **messengers**, or **active packets**, or **capsules**, or **molecules**.

2. On Present and Future Network Models

execution on the destination node). The last requirement is not mandatory (some agents do not carry their state along) but is a good practice while designing systems based on this technology. Mobile agents are capable of performing appropriately in the new environment, i.e., they are able to adapt to the changes. Sometimes the decision making functionality on when and where to move is encapsulated inside the agent, sometimes this is a function of an execution environment.

Due to higher requirements on resources active networks find very limited use in embedded systems including WSN. For instance, *Aglets* do not run on Java ME. The examples of mobile code used in WSN are: *Agilla* agents, *ChameleonVM* capsules and *Fraglets*. The first one [FRL09a] belongs to the class of classic mobile agents. *ChameleonVM* capsules (see Section 3.2) are a highly morphable form of mobile code designed specifically for use in resource-constrained systems. *Fraglets* implement the concept of chemical networking (see Section 2.4).

Hereinafter, when we refer to the mobile code we will assume the following properties:

- encapsulates code and data,
- delivered to a node and executed there,
- can generate other code pieces,
- can be merged/split/erased, and
- requires a different approach to protocol design.

As the development of active networking and mobile code progressed in different areas, the following question was raised: Do all protocols need to be implemented in an active network fashion? And, the answer would rather be no. This is especially true for systems posing high requirements on security, reliable delivery and real-time operation. Although WSN might seem to be one of those “no-go” architectures in this work we show that it can definitely benefit from implementing some of its software components using the mobile code concept (see Section 3.2). This will result in a provision of more flexible system design with high re-programmability level, the feature that is vital for WSN. Following this, optimization of message exchange mechanism will help to improve an overall performance of the system.

2.3 Autonomic Architectures

Autonomic Computing (followed by **Autonomic Networking**) was an initiative started by IBM in 2001. Its ultimate goal is to design a new generation of computer systems capable of **self-management**. Self-management allows for overcome of the rapidly increasing complexity of computing systems management. In other words, autonomic computing offers the new self-managing characteristics of distributed computing resources. Such systems can adapt to unpredictable changes; the intrinsic complexity is hidden to users. Autonomic systems make decisions on their own by using high-level policies. They constantly check and optimize their status and automatically adapt themselves to changing conditions. An autonomic computing framework is composed of **Autonomic Components (AC)** interacting with each other. An AC can be seen if a form of two main control loops (local and global) with **sensors** (for self-monitoring), **effectors** (for self-adjustment), **knowledge** and **planer/adapter** (for implementation of policies based on self- and environment awareness). Driven by such a model, a number of architectural frameworks based on “self-regulating” AC have been recently proposed (e.g., ANA project [BJT⁺10]). Most of these approaches, however, exploit centralized or cluster-based server architectures.

Normally, the following four elements of the above IBM’s autonomic computing initiative are used to describe a typical autonomic system:

- self-configuring: automatic test and installation of software releases, parametrization,
- self-healing: system can restart applications if they fail,
- self-protecting: pro-active intrusion detection and prevention, and
- self-optimizing: task-driven run-time software customization.

Below we discuss these four principles in detail.

2.3.1 Self-Organizing Networks

From Section 2.3 we have learned that **self-organization** (or **self-configuration**) is one of the key factors of the autonomic network architecture. The level at which the system can be called self-configurable is flexible

2. On Present and Future Network Models

though. It can be seen from different angles: name/address auto-resolution, topology build-up, synchronization, etc. Not all the systems named “autonomic” have all features, most have only a few. Let us consider two specialized network architectures, which feature some aspects of self-organization.

Mobile Ad hoc Networks (MANET): These are fully decentralized, infrastructure-less wireless networks. They have no administrative management, i.e., no network operator. Therefore, in contrast to Internet there is no “managed” support for system services like routing, name resolution, etc. The entire networking operation depends on incentives to collaborate from participating users (nodes). Another two key features of MANET include: 1) nodes are both hosts and routers, and 2) nodes can be mobile. The latter still remains one of the biggest challenges in MANET design and slows down their wide usage. What is more or less successfully deployed nowadays is **mesh networking**. This is a distributed radio network with a static core and relatively static or lightly mobile end users. Mesh protocols are operated only by the static core providing an Internet-like interface to end users (e.g., DHCP, DNS, SMTP, etc.).

Wireless Sensor Networks (WSN): In contrast to MANET this is normally a centralized network. Sensor nodes typically collaborate to collect/process/store the measured data and transfer it to one or many data **sink(s)**. A WSN is typically a dedicated system; it is designed for performing one particular task in some particular environment. WSN can be mobile. Even if WSN is fully static, it can show some properties of a mobile network due to unstable radio links. WSN was one of the first attempts to apply the idea of **Data-Centric Networking (DCN)** (or **Content-Centric Networking (CCN)**) in practice (see Section 2.5.3). In contrast to classical networking where routing is based on nodes and addresses (i.e., unique global identifiers), with CCN routing is based on data labeled with some attributes [IGE00].

2.3.2 Self-Healing Protocols

Hardware/software is not perfect. This is especially true for complex systems such as networks. Three steps can be outlined in the process of dealing with malfunction behavior:

- error detection (diagnosis),

- error handling (undoing the harm), and
- fault tolerance (prevent from faulting again and delivering correct service in the presence of faults), self-protection.

If the system provides a certain level of the above properties, we say that it belongs to the class of **self-healing** systems. The concept of self-healing is closely related to the idea of **self-protection** as a prevention mechanism before self-healing might kick in. Self-protection relies on several forms of redundancy for being able to recover:

- hardware redundancy: parallel execution, majority filters,
- software redundancy: software copies,
- time redundancy: sequential multiple execution, majority filters, and
- information redundancy: error correcting codes.

Since the redundancy always comes with a need to use extra system resources (CPU time, memory, etc.) special ways to approach it are used in ES where resources are extremely limited. In systems like *Maté* and *Agilla* the self-protection is implemented at the level of the execution environment. The corresponding VM controls program execution, and when an error occurs the program is suspended or removed. The new code distribution is then needed. *ChameleonVM* extends this functionality with an idea of capsules' lifetime. Capsules decay after a certain time allowing the system to update its execution context almost naturally. In this case, damaged code vanishes automatically and is replaced with a healthy copy. Fraglets use the idea of **Quines**, never-ending self-replicating pieces of code which can also be used to re-generate other code pieces. This approach might be sometimes harmful to itself if the population of ever-growing Quines is not regulated properly. Moreover, as it is shown in [Mey10] sometimes Quines can start to grow unexpectedly. The above methods provide self-healing properties at the code execution level. An additional technique, which can be used on top, is FEC, which provides code redundancy at the encoding level. We discuss code robustness further in Section 4.5.

2.3.3 Self-Optimizing Systems

Autonomic architectures must be smart enough to adapt to the changes in the environment. If a system is built using mobile code it is more

2. On Present and Future Network Models

easily achieved than with static packets as we can change code on the fly. Mobile code means late binding of functionality that in turn means communication becomes instruction-based. This provides more granularity for network operations and code deployment. That is why many mobile code platforms use some sort of byte-code. In this work, we try to occupy the niche of dynamic mobile code optimization allowing reaching a better code representation and a better utilization of system resources.

2.4 Chemical Networking and Fraglets

An **Artificial Chemistry (AC)** [MYT08a] is a bio-inspired computer model used to simulate various types of systems. AC is in some ways similar to a chemical reaction, hence the name. **Chemical Networking Protocols (CNP)** [MT09] are a follow-up field of study which uses a chemical representation to simulate and design new networking protocols.

Mentioned above, Fraglets [Tsc03], is a framework (programming language, simulator, execution environment) to design, build and evaluate networking protocols based on the principals of CNP. Formally, fraglets are tiny computation fragments (hence the name) designed to be integrated into infrastructure of an active network. Thus, fraglets can not only carry passive data along through the network, but also execute code (themselves) on routers and nodes that they pass by.

“There are two ways to look at Fraglets. First, fraglets implement a chemical reaction model where computations are carried out by having fraglets ‘react’ with each other. Alternatively, fraglets can be seen as data flow tokens that work themselves through communication media and routing tables – conceptually, the CPU is turned inside out such that the network becomes the CPU’s bus. An interesting twist (with both views) is that fraglets blend the notion of code and data, overcoming the discrepancy between ‘classic’ and ‘active’ networking.”¹

Based on fraglets, CNP addresses the problem of in-network packet processing through the prism of chemical kinetics. CNP helps to develop and analyze network protocols in a highly dynamic, reaction-oriented fashion. The dynamic nature of CNP comes with an overwhelming message complexity. It requires messages exchange of rela-

¹Borrowed from <http://www.fraglets.net/>.

tively small size but high intensity. For us, this is a perfect target, as it would allow observation of how message complexity can be reduced.

2.5 Unconventional Networking Styles

In the following section we discuss several modern styles in networking. These “styles” are not new paradigms on their own, rather they exploit the existing ones and add new functionality on top in order to adapt to special contexts. The reason why we talk about them here is that these concepts find applications at different levels in the WSN domain and because some of our examples from Chapter 6 use them too. In particular, we have a look at the following topics: Delay-Tolerant Networking (see Section 2.5.1), gossip protocols (see Section 2.5.2) and Content-Centric Networking (see Section 2.5.3).

2.5.1 Delay-Tolerant Networking

The first principle we mention is **Delay-Tolerant Networking (DTN)**. This is an approach to architecture computer networks that seeks to address the issue of lacking continuous network connectivity. Examples of such networks are those operating in harsh terrestrial environments, mobile or in-space networks (a.k.a., **inter-planetary Internet**). Disruption in connectivity may occur because of the limited range of wireless radio, mobility of nodes, energy resource constraints, as well as intentional attacks and channel noise. Classical network architectures like Internet are built on the following underlying assumptions: 1) continuous, bidirectional end-to-end path, 2) short round-trips, 3) symmetric data rates, and 4) low error rates. In contrast, the DTN [Fal03] inverts these characteristics to: 1) intermittent connectivity, 2) long or variable delay, 3) asymmetric data rates, and 4) high error rates. As can be seen WSN feature all of the above and can, therefore, be approached using the DTN model taking into account specific limitations of WSN [Lou06]. Alternative field-tailored solutions exist. For example, in [TST08] we proposed a “multi-level in-network cache” method to provide reliable data delivery in intermittently connected WSN.

2.5.2 Gossip Protocols

The second is the **Gossip** protocol (a.k.a., **epidemic** protocol) which is a network communication model inspired by the form of gossip seen in social networks and its information spread function. The model

2. On Present and Future Network Models

involves periodic, pair-wise, inter-process interactions. Reliable communication is not assumed. The frequency of the interactions is defined by the main protocol task and is normally relatively high compared to more classic counterparts. Communication peers are selected using either some form of randomness or a continuous peer sampling/ranking mechanism.

Gossip protocols are often used by modern distributed systems to solve problems that might be difficult to address using more traditional approaches. The following reasons are normally the cause why one can start using gossip-based methods: 1) the underlying network has an inconvenient structure, 2) the network is extremely large, or 3) sometimes gossip-based solutions show a better performance. The following three types of the gossip model are distinguished: dissemination, data replication and aggregation protocols.

Dissemination protocols are the most used form of gossip model in WSN (a.k.a., **flooding** or **viral** protocols). They are widely used by broadcast services to disseminate control messages. These services include: maintenance of routing tables, building hierarchies, directed diffusion, code propagation, time synchronization, etc. However, flooding produces an excessive number of unnecessary control packets, markedly increasing overhead. To address this, a more field-specific models are used sometimes. In [Tal08] the problem of time synchronization with minimal message complexity is discussed. We lately use this example in Section 6.4.6. Another example we use in our work is described in Section 6.5, a classic data aggregation gossip protocol with huge message overhead.

2.5.3 Content-Centric Networks

Content-Centric Networking (CCN) (a.k.a., **Content-Based Networking**, **Data-Oriented Networking** or **Named Data Networking**) is an alternative approach to the design of computer networks whose fundamental principle is that data is retrieved by name, not by location. In this model, there is no specific, physical location identifier like an IP-address in the Internet. Instead, CCN introduces a named content model. At the same time, CCN is built using TCP/IP design patterns. That is why it can support any current networking application in the Internet.

The CCN approach comes with a wide range of benefits including: 1) content caching to reduce congestion and latency and improve delivery speed, 2) significantly simpler configuration of network devices (since there are no actual end points), 3) improves network reliability and performance, and 4) integrates security (both authentication and ciphering) into the network and at the data level.

In the WSN domain the CCN ideology has a long history in a form of content-centric middleware and routing protocols [IGE00]. This is mainly dictated by the fact that WSN are naturally data-oriented networks. This type of middleware provides a simple abstraction through which applications can request/exchange uniquely identified content. We discuss this further in Section 3.1.2.

2.6 Summary

In this chapter, we have given an overview of the past, present and future of the networking technology. The trend to design systems with more “self”-features has greatly evolved since the first attempts to use mobile code for building network protocols and applications. Systems are gaining more autonomous, self-organization and self-configuration properties. Run-time self-optimization is becoming, more and more, an essential part of system design. It allows tuning of the system to perform a particular task. This is a specifically welcomed feature for resource-constrained systems like WSN. Finally, self-healing tries to manage an (un-)intentional system misbehavior. We have looked at one of the prospective methods, which describes network interactions through chemical reaction metaphors. As we show later, to some extent the concepts of this method can be applied to embedded network applications as well. At the end we have also listed a number of non-traditional approaches to networking which have already found applications in the embedded world.

3

Configurable VMs for Embedded Networking

First, this chapter gives an overview of the existing re-programming and re-tasking solutions for WSN. We analyze these solutions and report their basic features. We find that none of them are suitable for our further research. Thus, we propose two prototype platforms which will be used in the later chapters: ChameleonVM and FragletVM. The first is an example of a “traditional” Virtual Machine (VM) for active networking with sequential execution flow. In addition, it introduces the properties of dynamic code optimization discussed in Chapter 4. The second is an embedded implementation of the Fraglets system, which essentially models the chemical networking environment with fragmented and parallel execution. We present system architectures of both, the code and data representation they use, instruction sets and execution flows. Special attention is paid to describe one of the main building blocks of both designs responsible for dynamic code compression, which is further described in Chapter 5. The content of this chapter is essential for understanding the program examples from Chapter 6.

3.1 Existing Solutions for WSN Morphing

In the introduction (Section 1.5) we have explained why we consider morphing to be an essential feature of any network system. In this chapter, we first give an overview of the existing solutions for network morphing and then present the corresponding toolkits we have developed for two different network environments: WSN and Fraglets.

Hereinafter, by saying **morphing** we assume changes made to the network software which lead to a change in network behavior. The following approaches can be considered as different forms of a morphing process: re-configuration (re-calibration), re-programming and re-tasking.

Re-configuration (or **re-calibration**) is a process of tuning certain network parameters which allows to better reflect current traffic and topology conditions in the network. For example, re-configuration is triggered when nodes join/leave the network (address and slot re-assignment), when timings of a network protocol must be adjusted¹ and when instead of taking samples every minute we would like to do it only hourly (change of a sampling rate). This is **soft-morphing** because it does not require any changes to the code but to the data structures this code has access to. Normally soft-morphing is a functional part of the original protocol architecture meaning that the protocol can make decisions and react to those changes without commands from the outside. This comes with a challenge: the protocol must be algorithmically ready for various situations; this functionality must be pre-programmed. This results in protocol functional expansion and yet a lack of sufficient functionality for some specific cases. On the other hand, re-configuration can be established via external commands (e.g., “set threshold number 5 to value 10”). In this case, the network protocol must have a number of entry-points – the more of these points it provides the more fine-grained tuning can be done on it. The positive side of re-configuration is low intensity and small packet size of inter-node communications required as well as no issues with harmful operation.

Re-programming assumes a partial or a full replacement of the program image. This allows any level of changes to the running software;

¹Most WSN protocols do not allow doing that at run-time.

3.1 Existing Solutions for WSN Morphing

software is distributed in a pre-compiled platform-oriented (machinery, binary) form. For making this possible, the system must provide a support mechanism – the engine which will replace some parts (modules) or the entire image. This requires memory space. The other two negative aspects of re-programming are intensive use of the communication channel while the image is being replaced (requires much power) and disruption of execution (the program should be stopped and can be started again only after the new image (module) has been put in place and configured). We call re-programming a **hard-morphing**. Of all three processes described here this is the most systematic way of changing the system behavior as every aspect of the original functionality can be changed.

Re-tasking is an approach based on using a non-platform-native coding scheme. To execute such code each node in the network must be equipped with an appropriate execution environment, a VM of some type. The code can be distributed in a text (script) or pre-compiled (byte-code) form. Many middleware solutions fall into this category, some kind of on-node interpreter is normally required. This method has a very big advantage over the previous two as it solves two problems: 1) network functionality can be deeply modified, and 2) it does not require distributing a new heavyweight program image each time. Its negative side is the need for a VM instance on each node, which must be pre-installed. Additionally, as this VM is normally implemented as an application it is not given enough rights to make any changes at the OS level – even updating its own code is normally not supported. As this type of morphing does not make any changes to the actual network behavior but rather helps to describe high-level tasks, we call it **abstract morphing**.

As it has been shown above, each of the three approaches has its own advantages and disadvantages in terms of required resources and provided services. In the ideal case the line between OS, VM and middleware should be erased. All OS, discussed in Section 1.3.2, provide only very basic software support (drivers, scheduler, memory space allocation, etc.). VM is responsible for providing higher level interfaces and further protection of process execution than OS. Middleware gives an opportunity to describe tasks using abstractions, which are close to natural language commands (e.g., SQL-like queries). The higher we move through this hierarchy the less control over underlying levels we

3. Configurable VMs for Embedded Networking

normally get. It is a very debatable subject of what level of granularity each level should provide. The current situation is rather restrictive, access to the underlying layers is not offered (middleware \rightarrow VM \rightarrow OS). Our focus is put on creating a so-called **netware**¹ level where network functionality can be regulated at the level of system services or even MAC-layer (see Figure 1.4; we should have a chance to replace or modify any of the system services as desired: middleware \rightarrow VM \rightarrow OS) and yet this task should be carried out using some level of abstraction (without making OS system calls). Additionally, this level should offer a set of well-defined interfaces which could be used by higher levels independently: middleware $\xrightarrow{\text{netware}}$ OS and VM $\xrightarrow{\text{netware}}$ OS). Further integration with an OS core could be done via callbacks: netware \rightleftharpoons OS. Why would having such a layer be useful? The answer, we would like to have a tool for building **task-specific configurations on the fly**. This is currently impossible. Instead of having all functionality pre-programmed and (maybe) configurable we would like to have a system tailored to some specific task at a time (e.g., at time τ_1 we want our system to measure and deliver data, at time τ_2 the system should only aggregate accumulated data, make in-network decisions and steer actuators; system profiles at times τ_1 and τ_2 should be independent). Switching between tasks (between profiles) should also be possible. Moreover, it is not required to have all the profiles available on a node at all time; profiles can be delivered or removed from a node on-demand. Having a temporary task-oriented profile in the system will allow us to optimize communications involved in carrying it out by reducing the amount of information (packet size) needed to be exchanged between nodes and, therefore, minimize energy consumption. It has been shown in [BA03] that the trade-off between communication intensity and computation complexity might be tricky (the less we transmit the more that has to be done on encoding/decoding side). The idea would then be to create a system where the encoding/decoding stage is simple enough to be carried out by resource-constrained devices like WSN.

The attracting thing about WSN for applying such a technique to is that you can go as deep as you want in modifying and customizing the underlying layers (MAC/link layer is fully re-programmable in WSN). In contrast with, for example WiFi routers, this gives a huge level of

¹Do not confuse with the *Novell NetWare* network operating system.

3.1 Existing Solutions for WSN Morphing

freedom for a developer. However, the same technique can be used on systems with lower modifiability.

We intentionally do not position our work as middleware but as an execution environment for network protocols (netware). To show the difference we will borrow the code dissemination classification from [BS06] and [HKSS05]. As shown in Figure 3.1 the following stages in a code dissemination are distinguishable:

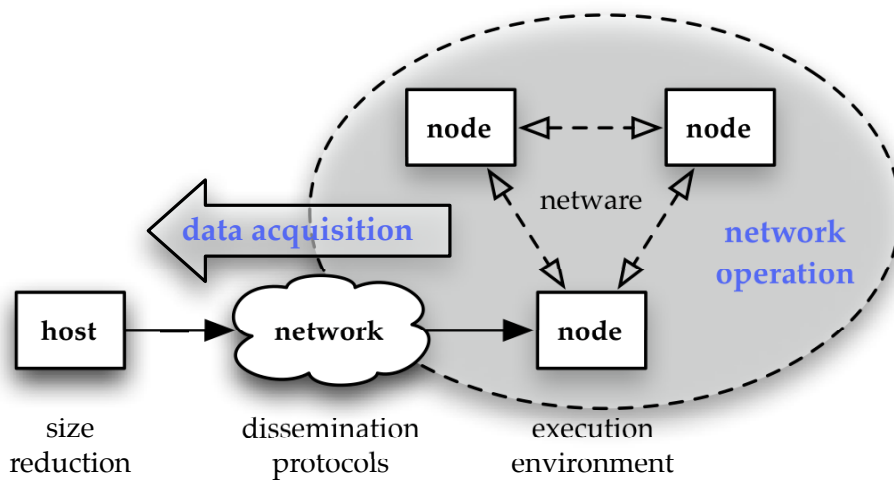


Figure 3.1: Code Dissemination (partially borrowed from [BS06])

As can be seen from Figure 3.1 middleware normally performs data acquisition and processing tasks whereas netware is involved into carrying out in-network services. Since netware is based on mobile code it borrows elements from the code dissemination shown in Figure 3.1. To understand which services the netware level must support let's have a look at each sub-type of code dissemination in detail.

Dissemination protocols: Dissemination protocols are used to deliver software updates to WSN nodes. These solutions normally concentrate on how to deliver data rather than what to deliver. Software updates are pre-compiled binary images of a relatively big size (comparable with the total amount of memory available on most WSN platforms which is 30–40 kB). Examples include: *Deluge* [HC04], *De-*

3. Configurable VMs for Embedded Networking

ployment Support Network (DSN) [DBK⁺07], *Impala/ZebraNet*, *Infuse*, *MNP*, *MOAP* [SHE03], *Trickle* [LPCS04], *XNP*.

Size reduction: In contrast to dissemination protocols this sub-type addresses an issue of bulky data transmissions by modular organization of software and incremental updates. This is normally done on a host, which can be either an access point computer or another node. However, it still assumes pre-compiled platform-oriented code similar to dissemination protocols. Examples include: *Reijers*, *Rsync*, *Remote Incremental Linking*.

Execution environments: Instead of distributing binary code these solutions introduce high-level programming schemes which allow description of network interactions using fewer instructions (e.g., using single “send” instruction instead of making a system call with many parameters). Examples include: *Agilla* [FRL09a], *COMiS*, *Contiki VM* [Dun07], *cSimplex*, *Mantis/MOS*, *Maté/Bombilla* [LC02], *REAP*, *SINA*, *SP* (*Spatial Programming*), Szumel et al’s *Mobile Agent Framework*, *Pushpin*, *ScatterWeb*, *SensorWare*, *SOS DVM* [BHR⁺06].

Most execution environments were designed to implement data acquisition, middleware tasks. This is similar to building a data collection application as we show in Section 6.4.7. That is the reason why they lack a number of fundamental features such as code manipulation, code morphing and compression, code navigation, etc., which are required for creating low-level protocols. On the other hand, they provide a big number of data processing functions, which are not needed for this type of operation. The last normally overwhelms their design.

As we have mentioned above, our focus is put on the so-called network level, which would allow us easy access to system network functions from inside the network. For example, node *A* wants to move some code chunks to node *B* which is two hops away. Node *A* has to be able to do that in an elegant and easy manner but without having an underlying propagation layer with huge message complexity. Decision-making is delegated to the nodes. The model should use a fully decentralized architecture; a need for session leaders or control signals from outside is not necessary. The last point is especially important for autonomic systems. Furthermore, the system must be relatively lightweight (memory, computations) so that WSN nodes can easily adopt it. It might seem similar to mobile agents technology like

3.1 Existing Solutions for WSN Morphing

Agilla mentioned above but there is one fundamental difference, mobile agents live in user-space and are not allowed to make any changes to the media they use. We propose a system, which in fact consists of multiple morphing-prone agents. The last statement means that agents are not static, their “filling” and, therefore, functionality can change. By prototyping such a system which features switchable task-specific profiles at a time, we will obtain a tool for our further work, namely carrying out analysis on dynamic code compression techniques (see Figure 3.2). The last element, in our vision, is an essential part of automatic construction of optimized network protocol stacks on the fly from fractions of code flooding the network (so-called **sponge protocols**). The step of automatic protocol construction we have not achieved in this work but we discuss it in Chapter 7.

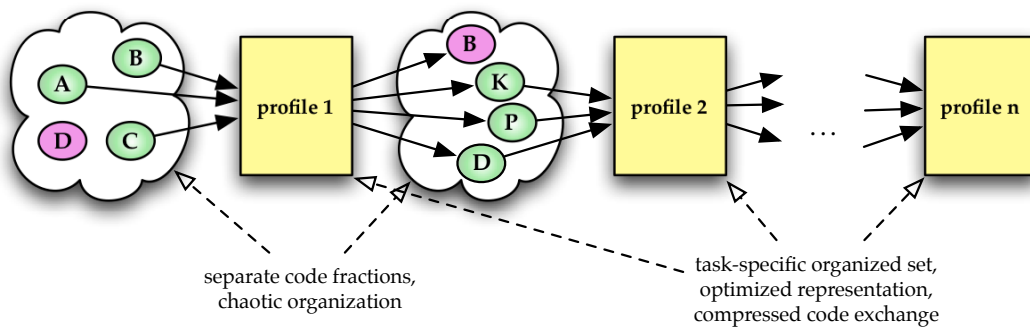


Figure 3.2: Profiles Building and Switching

Figure 3.2 can also be seen as a process of building a solution which solves a specific problem at a time from a set of available functional building blocks. Following this, the solution’s encoding is optimized. In our case the optimization criterion is the resulting code size. Other criteria such as message complexity can be used instead or in combination.

Before describing the details of the network level we would like to propose a slightly different classification of code dissemination techniques, based not on their place in the network or their functionality but the level of control over code morphing they offer.

3. Configurable VMs for Embedded Networking

3.1.1 Frameworks for Re-Programming

Re-programming requires access to low-level (OS-level) services, thus it is normally implemented as a part of OS. A classical representative of this type is *Deluge* [HC04] in *TinyOS* which pioneered a lot of principles for code dissemination in WSN. *Deluge* allows to wirelessly install a new binary program image. This is accomplished by propagating a program binary over the wireless network and having each node program itself with the new image. After all the images have been delivered the node can switch between them and run different applications on demand.

This approach is the least error-prone because the entire program image is replaced. Although it comes with a number of disadvantages, one of which is the significant power consumption due to the need of transmitting the entire program image (in case of *TinyOS* it includes the OS code as well, and might be as big as 30–40 kB). The other big disadvantage is the disruption of network services and, therefore, the lack of operation during maintenance.

Systems like *ContikiOS* [DGV04] and *SoS* [HRS⁺05] tried to improve this mechanism by utilizing their modular architecture and allowing transmission and installation of separate modules [DFEV06b] instead of full images. The issue with power consumption in this approach still remains, as modules are still too big. Moreover, each module has to carry information needed for the dynamic linking on a node.

Code remains static and immobile (it is delivered and installed); further in-network interactions are based on exchanging data packets as usual.

3.1.2 Frameworks for Re-Tasking

In contrast to re-programming, re-tasking brings the morphing task to the user level. Basically, all middleware solutions fall into this category as they hide system-level complexity and provide a set of simplified programming abstractions which should be enough to describe any task in a particular application domain. The middleware population for WSN is rather huge. Particularly, it includes: node centric (e.g., *Hood*, *Abstract Regions*, *Logical Neighborhoods*, *Virtual Nodes*) and macro-programming (e.g., *Regiment*, *Kairos*, *ATaG*) abstractions, VM (e.g., *Maté*, *ASVM*, *DAViM*), distributed databases (e.g., *Cougar*, *TinyDB*, *SINA*, *SwissQM*). Additionally, middleware solutions can be

3.1 Existing Solutions for WSN Morphing

event- (e.g., *Mires*) and application- (e.g., *MiLAN*) driven, component-based (e.g., *RUNES*, *MWSAN*); with further extensions like tuple spaces (e.g., *TinyLime*, *TeenyLime*) and tuple channels (e.g., *TCMote*) or mobile agents (e.g., *Agilla*, *MAWSN*, *actorNet*). For further information on existing middleware solutions and their comprehensive classification and analysis, refer to [MP09].

In these solutions code mobility is typically limited and pre-determined: code goes where it is told to, normally it does this only once and resides on a node until it is removed. Code cannot make decisions by itself. Many solutions do not even use the concept of code but rather commands or queries which are translated in some form of a bytecode. The only exception which goes beyond these limitations are mobile agents.

In general, middleware provides abstractions from the underlying network layers. We would like to bring interest back to the network level. Below we will consider two of the above, the most inspiring solutions for this work: VM (see Section 3.1.2.1) and mobile agents (see Section 3.1.2.2).

3.1.2.1 Virtual Machines for WSN

Maté was the first VM specifically designed for use in WSN. It is written in *nesC* [GLvB⁺03] programming language used in *TinyOS*. *Maté* uses *Trickle* protocol [LPCS04] as an underlying code propagation layer. Programs are encoded using a special assembler-like language. Code dissemination is done using small capsules of 24 bytes. If a program requires more code it can be divided into several capsules which are delivered to a node, installed and run as a single program. *Maté*'s capsules can forward themselves to other nodes. In the heart of *Maté* lies a byte-code interpreter also providing network, logging, hardware and boot/scheduler capabilities. Later *Maté* was later extended to *ASVM* [LGC04], [LGC05] which allows to specify an instruction set at compile-time.

Other VM exist for WSN. The most notable solutions and their remarkable features are shown in Table 3.1.

3.1.2.2 Mobile Agents for WSN

Mobile agents serve for re-tasking as well. A good example in WSN is *Agilla* [FRL09a] which is a middleware that provides a mobile-agent

3. Configurable VMs for Embedded Networking

	Name	Memory Footprint (code/-data)	Application Execution Method	Programming Model	Domain Specific VM	Remarkable Features
Middleware Level VM	Maté	7.5 kB / 600 bytes	Interpretive	Stack-based	No	Tiny program capsules
	Scylla	Unknown (rich nodes)	Native code	Register-based	No	Runs native code using on the fly compiler.
	SensorWare	180 kB / unknown	Interpretive	Unknown	Supports	Multiple applications (script) Native code exposed as VM instructions; dynamic, multiple applications.
	DVM	13 kB / 600 bytes	Interpretive	Stack-based	Yes	
	DAViM	Unknown (rich nodes)	Interpretive	Stack-based	Yes	Dynamic
	QM	33 kB / 3 kB	Interpretive	Stack-based	No	Choice of instruction set, compact and space efficient representation; multiple applications.
	ASVM	38 kB / 2.9 kB	Interpretive	Stack-based	Yes	Based on Maté. Adds customizable instruction sets. Class compaction; dynamic, serializable representation of objects; quasi-threading dispatch; multiple applications.
System Level VM	VM*	6 kB / 200 bytes	Interpretive	Stack-based	Supports	
	MagnetOS	1.3 MB / unknown	Interpretive	Stack-based	No	Multiple applications; network resources management.
	Squawk	660 kB / unknown	Interpretive	Stack-based	No	Compacted bytecode; multiple applications; debugging.
	S- and E-VM	1000 lines of code / unknown	Unknown	Unknown	No	Targeted for timing code, rather than functional code.
	TinyVM and leJOS	<10 kB; 17 kB / unknown	Interpretive	Stack-based	No	Compact class files

Table 3.1: Classification of VM for WSN (borrowed from [CPS07])

3.1 Existing Solutions for WSN Morphing

style of programming. Applications are constructed of mobile agents that can migrate their code and state across the network; programs can now control where they go and to maintain both their code and state across migrations. Functionally, mobile agents are able to

- provide more flexibility as they allow applications to control how they propagate and self-distribute in the network,
- bring themselves to the most optimal locations within the current configuration to perform application-specific tasks,
- save energy by bringing computation to the big amount of data rather than requiring the data be sent over unreliable wireless links to the location where the computation sits,
- re-distribute the functionality inside a WSN field by “attaching” themselves to specific locations that better meet the current application’s requirements (instead of spreading the repetitive code throughout an entire network), and
- share the resources of a single node, i.e., multiple mobile agents can reside on each WSN node.

The following advantages of using mobile agents can be thought of:

- network re-tasking (since new agents can be injected into a pre-existing network),
- multiple applications can co-exist (since each agent executes autonomously and multiple agents can simultaneously be executed on a node), and
- quick adaptation to changes in an environment (since mobile agents can move and clone).

The fact that *Agilla* is essentially middleware limits its use to user-level applications. Agents can clone themselves and migrate from node to node but these actions use pre-determined algorithms (e.g., moving through a grid in case of *Agilla*). Moreover, mobile agents do not have control over underlying layers (transport, MAC). Another disadvantage is that mobile agents are static in terms of their own structure, they are pre-programmed pieces of code which cannot be modified.¹

¹The description is borrowed from <http://mobilab.cse.wustl.edu/projects/agilla/>.

3. Configurable VMs for Embedded Networking

Servilla [FRL09b] brings the mobile-agent concept to the next level. It is a highly flexible service-provisioning framework for enabling applications to execute within heterogeneous WSN. It features a service-oriented programming model and a modular *Agilla*-like middleware layer. This combination enables to construct platform-independent applications over a set of devices with diverse computational resources and sensors (e.g., PDA, WSN nodes, laptops, etc.). The *Servilla*'s middleware architecture can be customized to devices with a wide range of resources and the application's functionality can be distributed according to the available resources on each node. In fact, *Servilla* provides a set of mobile agents spread over the network. On top of that *Servilla* uses binding configuration scripts which enable it to build and deliver a service to a specific position in the network. A distinctive feature of *Servilla* is its support for dynamic discovery and binding to local and remote services. All these allow building computationally- and energy-efficient in-network processing applications.

The main shortcoming remains the same as with *Agilla*, code itself is immorphable and the abstraction level is too high to be able to build truly task-specific network configurations. Another disadvantage is that both *Agilla* and *Servilla* use an underlying code propagation layer with a huge message complexity.

Another examples of mobile agent systems for WSN include: *Wave* [GVVL06], *Impala* [LM03], *SensorWare* [BHS04], *SmartMessages* [KBX⁺04], etc.

3.1.3 Netware

None of the above solutions described in Section 3.1.2 are suitable for our further research since the level of task description is too high and certain design limitations exist (e.g., pre-defined code dissemination layer, fixed execution flow, etc.). In our work we want to bring focus on having what we call netware, the level helping to build task-specific protocol stacks using OS-level network abstractions. It is supposed to become a foundation on which network stacks can be designed in a mobile code manner, deployed, executed and steered. Additionally, we would like to pay an attention to autonomic "self-*" features (deployment, organization, maintenance) of network applications. Besides all that applications must stay resource- and energy-aware.

3.1 Existing Solutions for WSN Morphing

The principal difference between classic network protocols and those based on mobile code is the following: traditional network protocols use data packets to exchange information about their current state between nodes. Upon receiving a packet each node makes a local decision on what it should do next. The mobile code approach assumes that no external decision making is needed. Parts of the program continuously travel through the network and are executed on each node. By doing so, they carry along all necessary information and further algorithmic steps to be made. We propose the netware, the level lying between OS and user application or middleware and substituting the current static network protocol stack. Although we develop the system for use in WSN domain, netware can also serve as a mobile code plane across heterogeneous networks and nodes (e.g., WSN nodes using different hardware platforms, or WSN nodes and WiFi routers). We illustrate this in Figure 3.3.

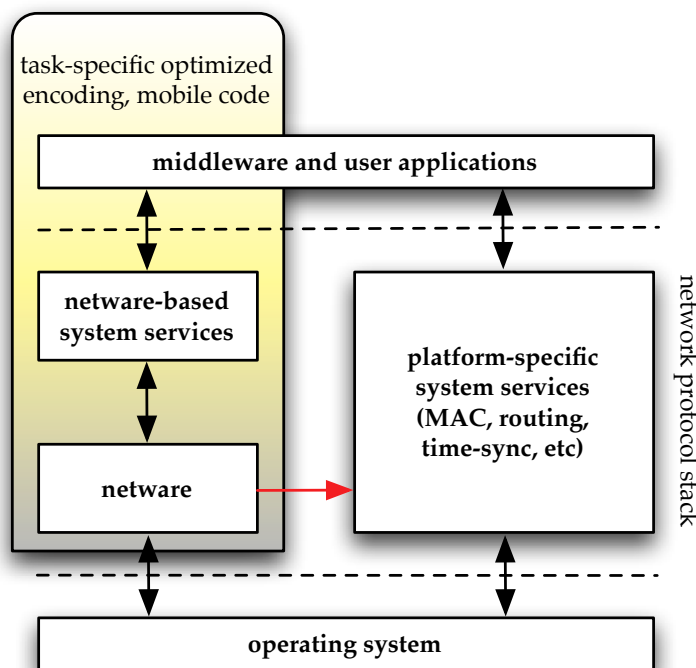


Figure 3.3: Netware Level within Node's Software Structure

3. Configurable VMs for Embedded Networking

Netware isolates custom network protocols from OS platform-specific complicated primitives (*Active Messages (AM)* in *TinyOS* or sockets on Linux-based routers). Its sole task is to be used as a building plane for task-specific network protocols. Network protocols are supposed to be created in a mobile code fashion. Netware allows them to be platform-independent. For instance, the same code will be understood by WSN and WiFi routers and executed using low-level platform primitives. In this work, we present netware for a WSN domain, further ports are pending. Netware is designed to live side-by-side with existing network protocols and to use their functionality if required. At the same time it provides tools for building custom protocols.

Lets consider an example of a simple network protocol (e.g., time sync protocol presented in Section 6.4.6 and Listings A.1 and A.2). In Figure 4.1a we show how this protocol was implemented using traditional principles (using static code and exchange of data packets) and one of many possible implementations using mobile code (Figure 4.1b).

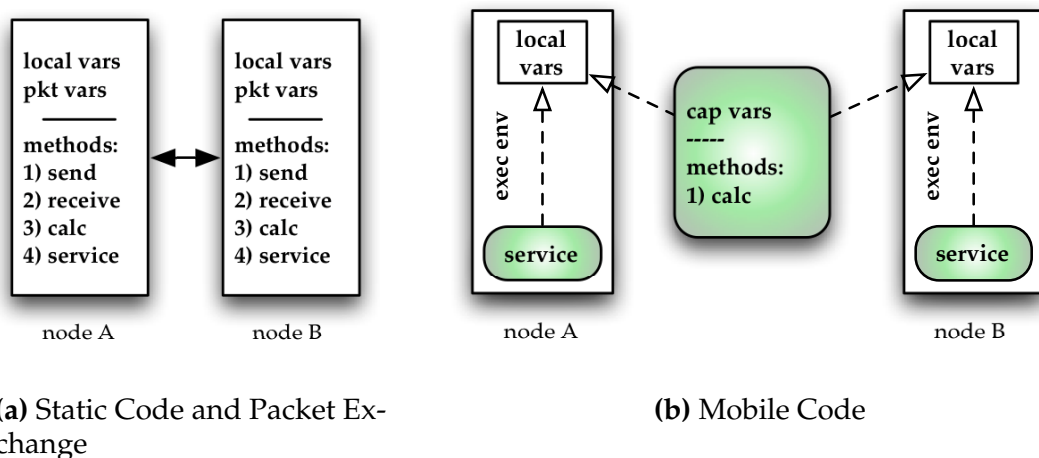


Figure 3.4: A Network Protocol Implementation: Static Code and Packet Exchange vs Mobile Code

Traditional implementation is set up as follows: local variables (reside on a node), packet (state) variables (carried with a packet) and a set of methods (“send” (when we need to send a packet), “receive” (is called upon arrival of a packet), “calc” (makes calculations using local

3.1 Existing Solutions for WSN Morphing

and state variables) and “service” (is called when additional actions are to be done, e.g., initialization)) on each node. In case of our simple time synchronization protocol from Listing A.1 those variables and methods would be translated as follows:

- “state variables”: `myNodeId`, `validSkewFlag`, `currentSkew`,
- “local variables”: all others,
- “send”: `at beacon send time`,
- “receive”: `at beacon reception time`,
- “calc”: `at end-of-wake time`, and
- “service”: `at wake-up time`.

In the mobile code version the picture changes dramatically: we have locally installed “service” functions and local variables, the rest of the functionality is encapsulated in the migrating capsule in the middle (see Figure 4.1b). Upon arrival to node *A* or *B* this capsule makes some calculations, updates in-capsule and local variables correspondingly and schedules its own departure. We do not need dedicated methods for incoming and outgoing packet processing as it is the capsule itself. Please note, that this is not the only configuration of mobile code possible. For instance, “service” functions can also become mobile, or local variables can be carried along as well.

The biggest challenge in designing network protocols in mobile code fashion is to find an optimal configuration which allows to perform a task in the most efficient way. The biggest advantage of this approach is an opportunity to have no code pre-installed in the system. Code resides in the media rather than permanently on a node. A node becomes just an execution engine. Code can be dynamically injected into the system and removed from it. This is true for individual parts of the original code set too. For instance, if we want to change the “calc” method we just replace the corresponding code section.

Compared to existing middleware and other approaches for WSN which provide the following features only partially or do not provide them at all, netware incorporates them all using very simple principles:

In-network decision making: This is the most important feature. Network protocols can be designed in a way that they become able to make decisions according to the current conditions.

3. Configurable VMs for Embedded Networking

For example, in the *PermaSense* project (see Section 1.4) we had a problem with quartz oscillators drifting away at different speeds on the nodes in the north and south clusters due to huge temperature variations between two sides of the slope (see Figure 1.5a). The temperature difference between the sunny and shady sides could be as big as 20 °C and change as the day goes by. A quartz oscillator is the source for time sub-system and time-related services like time-synchronization on WSN nodes. Using previous generation hardware platform and static protocols we had to introduce a drift correction table on each node, otherwise nodes got desynchronized and lost connection to each other. The drift correction was measured for certain temperature levels. Each node then had to compensate its drift in the software. In our mobile example above this task becomes much easier to carry as we can just extend the “calc” method with a new “drift compensation” code.

The most interesting part of in-network decision making is that new code can be generated inside the network by mutating the existing code set. Code injection from outside is needed only when brand new functionality is requested.

Optimal representation: As described earlier in 3.1, we use the concept of network profiles, optimal network stack configurations to perform a specific task. The process of finding an optimal representation runs constantly (see Chapter 5), adapting to the current software configuration.

Distribution of computations: By doing task-clustering of a network (moving computations to specific locations; see Section 1.5.1) we can achieve much better figures on performance compared to “each-node-does-everything” approach. We can call it **task delegation**.

Remote deployment: No pre-installed code is required on a node, except for netware execution environment (see Section 3.2.6), which in turn can be implemented as OS module or part of OS image.

No overlay network: Since many existing middleware solutions create an overlay network, we allow applications to use the existing infrastructure (existing protocol stacks including dissemination layer) as much as they can. Although in order to make this

possible, some changes to the existing code are required (see Section 3.2.12).

Since we do not intend to develop an underlying code propagation/migration layer, nor to preserve the state of the execution (in contrast to mobile agents discussed in Section 3.1.2.2), we can implement the features which are missing in the existing systems. Namely they are: 1) code manipulation, and 2) code optimization.

Obviously, a system like netware highly relies on network connectivity: code can be lost or corrupted (which is more probable while transmitting rather than while locally executing it – see Section 4.5). Therefore, special protection methods can be used to keep the system alive like code replication and code preservation. We use a simple MAC-layer acknowledgment scheme for code exchange (see Section 3.2.6). This simplifies design and keeps power consumption low.

In general, netware requires a total rethinking of how network protocols are designed today. Below we present two types of netware. The first one, called *ChameleonVM* (see Section 3.2), is WSN-oriented. It was created for use on sensor nodes, however, its design principles can be easily adopted for other network devices, e.g., routers. The second example is *FragletVM* (see Section 3.3), the netware execution environment for chemical protocols. Later, in Chapter 5, we use both systems to analyze the code optimization methods we propose in Chapter 4.

3.2 ChameleonVM

ChameleonVM is a netware level for WSN. However, some design ideas in *ChameleonVM* are similar to those found in the existing execution environments for WSN like *Maté* (e.g., scheduler, memory organization). The reasons for that are simple design and the need to meet specific constraints of WSN-nodes. Nevertheless, some parts of the design proposed here are debatable. For example, we use Round-Robin scheduling which is proved not be the best for real-time applications; preemptive, prioritized multi-tasking would be more preferable in this case. We use simple solutions where it was acceptable in order to achieve fast prototyping and directly go to the core of our design. Moreover, we try to position our system for use in a wide variety of network ES including and for example, Linux-based network routers – in this case a more appropriate module configuration should be chosen.

3. Configurable VMs for Embedded Networking

Although we sometimes borrow the existing principles there is a big number of new architectural principles discussed below.

The main task of *ChameleonVM* is to allow building, execution and providing support for network protocols. These can be protocols designed using the tools provided by *ChameleonVM* or external protocols (e.g., created using *Rime* [DOH07]) which can be made compatible with *ChameleonVM* (e.g., external protocols can be tweaked to exchange information with the running *ChameleonVM* programs; see Section 3.2.12).

Like *Maté*, *ChameleonVM* uses the concept of capsules (see Section 3.2.2) for code propagation. **Capsules** are small self-sufficient fragments of code which flood the network. Each capsule contains data and code. In contrast to *Maté*, *ChameleonVM* does not have an underlying code propagation layer; *ChameleonVM* programs either rely on existing layers or should use in-built features to propagate themselves through the network. The way propagation is done will have effect on how efficient and reliable the final protocol behaves. *ChameleonVM* is build using the functional blocks shown in Figure 3.5:

In the incoming path there are: packet multiplexer (MUX), code verification (CVM), dictionary (DICT) and updater (DICT UPDATER), decompressor (DECOMP). Multiplexer redirects packets according to their type. Data packets are forwarded to the local data storage (RAM, Flash) or go directly out if they are meant to (e.g., forwarding packets of the underlying layers). Dictionary updates are sent further down to the dictionary where they are applied. The third type of packets are capsules. First, capsules go through code verification module which is responsible to make sure that corrupted or malicious code does not go any further. Then capsules are moved through the decompressing stage after which they are finally ready for execution; they are put in a waiting queue from where they are picked according to the scheduling.

The outgoing path is much simpler and consists of compressor (COMP) and demultiplexer (DEMUX) only. Compressor accepts capsules scheduled for sending them out. Demultiplexer is used to move three sources (packets from RAM or Flash, capsules and dictionary events (rules)) into the network.

The central execution module (CEM) fetches capsules from the queue according to the scheduling and executes them. After the capsule has been executed once there are three possible destinations for it: it can be put back into the queue (as a whole), gets combined with

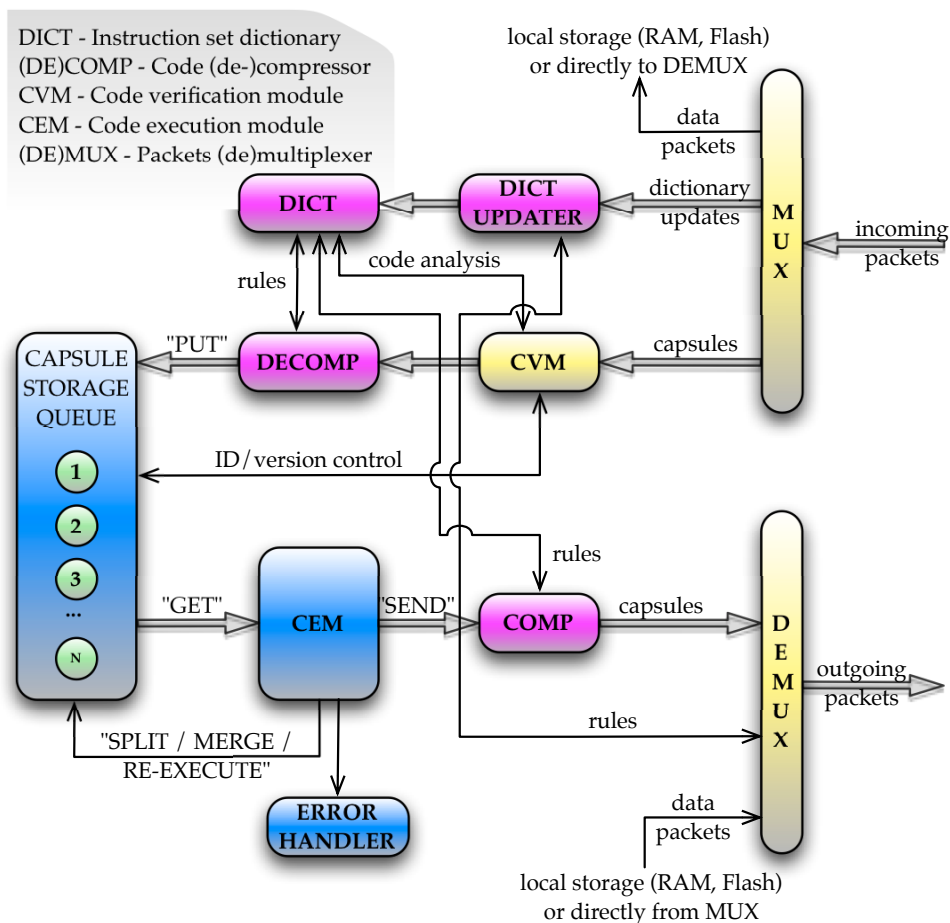


Figure 3.5: ChameleonVM: System Architecture

other capsules or split and put back into the queue, or leaves the node through the compressor and demultiplexer.

Capsules can take up various functions: management (initialization, reset), communication (send, receive), computation. Capsules can be mobile (can navigate themselves through the network) or static (can be sent out by others but do not encapsulate this functionality in their own code).

According to block structure in Figure 3.5 we can highlight three main functional layers the VM consists of: Code Control Plane (CCP), Code Execution Plane (CEP) and Code Optimization Plane (COP) as shown on the right side of Figure 3.6. CCP (see Section 3.2.6) is used to deploy and control code propagation. As there is no specialized propagation layer the way a capsule would like to move through the

3. Configurable VMs for Embedded Networking

network must be pre-programmed (e.g., “go to all neighbors” or “go one level up in the spanning tree”). Therefore, there is no particular module responsible for disseminating code over the network.¹ CCP is also responsible for installing and versioning code updates. CEP allows first to verify code and then to execute it on a node. CVM and CEM modules (see Sections 3.2.3 and 3.2.5) are the main functional blocks in this plane. COP brings an optimal representation to the code by constantly observing the code structure and re-encoding instructions to achieve the smallest code size. DICT, COMP and DECOMP modules are responsible for that (see Sections 3.2.9, 3.2.10 and 3.2.11).

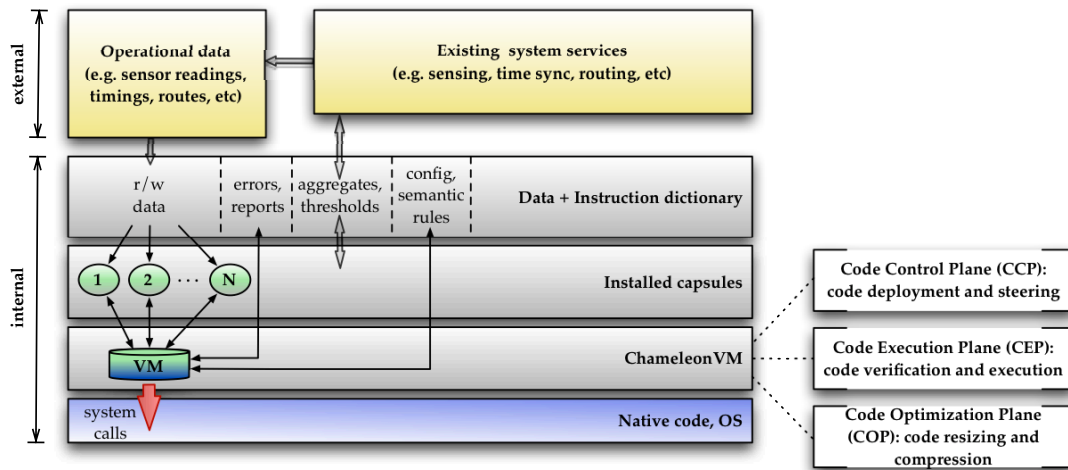


Figure 3.6: ChameleonVM: Functional Planes and Data Flows

In Figure 3.6 we also show the relationship between code and data in a system which uses *ChameleonVM*. As integration at the code level between *ChameleonVM* and existing protocols may be a real challenge (different coding and execution styles) we decided to provide interaction between them via data. The existing protocols can be easily instrumented to enable regularly reading/writing of individual fields or whole structures into memory. Rather than creating a whole new complicated interface between them we allow existing protocols to access the data memory space of *ChameleonVM* through a number of very

¹*ChameleonVM* does not support update of its own code. This must be done through the underlying OS if it is possible.

simple methods (see Section 3.2.12). When data is there, *ChameleonVM* can start using it to provide on-top functionality. This mechanism also allows for external protocols to read information provided by *ChameleonVM*. Such approach is very universal as we do not know which protocol might want to integrate with our VM: it might be time sync, routing or something else. Additionally, we allow the existing (platform-specific) software to access our dictionary engine – instruction can be added/changed/removed. In particular, this allows to assign platform-oriented methods a unique code in our instruction set and after that can be used as an alias whenever we need to call upon it (see Section 3.2.12).

Before we discuss the more complex aspects of the system architecture let us have a look at the very simple code example which will explain some basic principles of *ChameleonVM*. The program in Listing 3.1 is a trivial alarm sensor. It is continuously, with a 1 second interval, taking temperature measurements on a node and if the temperature drops below 30 °C then the red LED goes off otherwise it is lit on.

```
1 # Example: switch red led ON if temp raise beyond 30 C, switch
2 # OFF otherwise
3
4 .sys                # SYSTEM segment
5     Autoupdate On    # capsule's parameters
6     Lifetime 10s
7     Id 0x10
8
9 .code.timer0        # CODE segment "timer"
10    sense TEMP       # sense temperature
11    jimple 30,L1     # > or < 30 C?
12    led RED,ON       # switch the red led on
13    jmp L2
14 L1: led RED,OFF     # switch the red led off
15 L2: delay 1000      # sleep for 1s and do it again
```

Listing 3.1: Simple Alarm Sensor in ChameleonVM

ChameleonVM uses an assembler-like language for programming. The assembler code is then pre-compiled by the user into byte-code form which can be interpreted by on-node instances of *ChameleonVM*.

3. Configurable VMs for Embedded Networking

Sometimes in this work we use reduced programming notations in the listings. These notations can be easily expanded into real code stream.

3.2.1 System Architecture

ChameleonVM uses **stack-based** architecture. All operations use stack to grab arguments from, and put results back, on the stack. This allows for extreme simplification of the VM kernel as it does not require complex address resolution and long data transfers (e.g., “fetching data from the main memory at the address specified by a register given by other register”). Another thing the stack provides is easy control over its state and easy protection of program execution. Also, the stack normally allows to use less program space to perform certain operations as they use the stack as explicit operand.

We allow the stack to have **variable resolutions**. It can operate as 8-, 16-, 32-bit word storage depending on which operation is being executed. Non-traditional word-widths are also possible, although nothing will be gained on the memory side from using them on existing hardware platforms: 5-bit value will still be stored a 8- or 16-bit word on most platforms. This starts to make sense when we switch from local operations to communications. Although word-wrapping is also used at the link-layer saving can be made from truncating operations and operands before we transmit them. For instance, if we use 5-bit words, then over transmitting 8 such words we can save 3 bytes ($8 \text{ words} * 8 \text{ bits} - 8 \text{ words} * 5 \text{ bits} = 24 \text{ bits}$). To this end, a resolution of each operation (a resolution of each argument) can be changed individually by specifying this in the on-board dictionary (see Section 3.2.9). By sending dictionary updates to other nodes this change can be committed for the entire network. Additionally, *ChameleonVM* can natively learn from the execution context. For example, if we sense some value and it never goes beyond 255 *ChameleonVM* can reduce this field to 8 bits instead of default 16. We refer to this as **resolution-variable instructions**. In theory, any operation can have variable resolution if it operates with primitive (numerical) arguments.

ChameleonVM is a single stack-based machinery, the result of each operation is put back on the same stack which arguments are taken from. This dramatically simplifies the design. Therefore, function calls (subroutines) are not supported. Each capsule can be considered as an individual function which can be locally and remotely addressed

by its ID. Stack's maximum capacity can be adjusted at compile-time, dynamic change of stack's size is not supported.

There are several types of memory regions in *ChameleonVM*: heap (dedicated memory pool for each capsule), shared memory (shared between capsules) and capsule store. The last is used for storing capsules and cannot be directly accessed from an application, it is managed by VM.

Heap: Each capsule has its own random access memory pool called a heap. This memory can be used for aggregating intermediate results, status info, etc. Two subtypes are available: on-node (uses on-node memory resources; the information stored in here stays on the node and does not migrate along with the capsule when it moves; denoted as **BUFS**) and in-capsule (a part of the capsule's memory space which resides in the data segment of each capsule; information stored there travels along with the capsule; denoted as **BUFC**). The first subtype can be used to accommodate intermediate results or program states which can be preserved on the node between capsule's trips; can be implicitly released when capsule leaves a node using the **die** command. In contrast to *Maté* our VM does not provide automatic garbage collection. The second subtype is an integral part of a capsule, it is released when capsule "dies". Both heaps offer similar operations to access and process the data (read, write, append). By default the heaps in *ChameleonVM* behave like a circular buffer. The **append** operation will add a new item to a heap in a circular fashion. Extra service functions over the heaps (min/max/average calculations, sort, etc.) can be implemented using aliases (see Section 3.2.4).

Shared memory: Shared memory in *ChameleonVM* represents a class of random access memory which can be used by any capsule installed in the system. It is a responsibility of each capsule to maintain the integrity of the shared memory. Supported operations are similar to heaps: read, write, append, etc.

Capsule store: Capsule Store contains all the capsules installed in the system (node). Although there is no direct access to this memory from an application, some operations over capsules (e.g., **split** or **merge**) can have an effect on the memory usage.

3. Configurable VMs for Embedded Networking

The memory organization of *ChameleonVM* is shown in Figure 3.7. Common capsule structure is shown on the right; it is discussed in detail in Section 3.2.2. On the node's side each capsule is represented by **context** as shown in Figure 3.7 on the left. Instruction pointer keeps track of the current instruction being executed. Error register accumulates various flags related to the execution: stack overflow, invalid instruction, etc. Error flags are further discussed in Section 3.2.5. Dictionary selector indicates the dictionary currently in use.

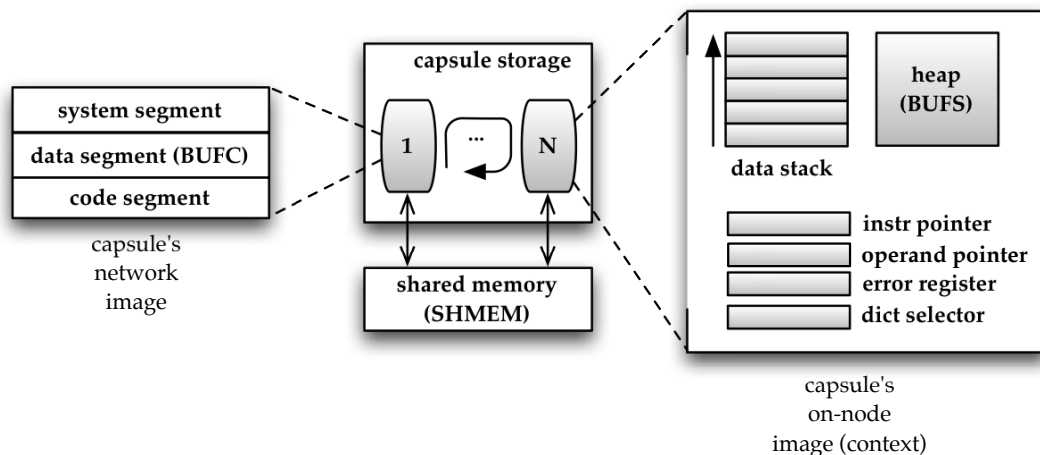


Figure 3.7: ChameleonVM: Memory Structure

ChameleonVM provides a **safe execution** environment which ensures that programs will not damage anything and will be executed in the best error-free manner. Each capsule is executed in its own name and memory spaces, they do not have access to the name space of each other. However, capsules can perform management actions on other capsules. For example, a capsule can request a merger with another capsule. Deletion is not possible though. We discuss **merge**, **split** and other capsule-related operations in Section 3.2.2.

ChameleonVM prevents occurrence of race conditions by applying **resource arbitration** to all operations. Resource management implicitly locks all shared components (send/receive buffers, sensor readings, Flash memory, etc.) and releases them when they are not in use anymore. Resource locks cannot be manipulated explicitly. The only exception is that the data in the shared memory (**SHMEM**) is not locked.

In order to plan tasks *ChameleonVM* implements a simple Round-Robin **scheduling scheme**. There are no priorities. Task (context) switching is done every 1 *ms* and is not affected by long operations. All operations in *ChameleonVM* are blocking which means that regardless how complex an operation is (either it is a simple **pop** (take something from the stack) or a **send N** (send a packet to node **N**), the execution will not continue until the operation is fully completed (all buffers are empty, all status information and return codes have been received, etc.). This does not affect context switching and scheduling.

Nevertheless, *ChameleonVM* is a concurrent engine which allows several capsules be executed simultaneously. This means if execution of the current capsule is postponed due to another event (e.g., waiting to finish transmitting a packet or sensing some value) then another capsule can be given a chance to execute at least a part of its code. Installed capsules are in charge of any possible execution dead-locks.

3.2.2 Programming Model

Code processed by *ChameleonVM* is broken in so-called capsules. The max size of a capsule is limited by the execution environment and, therefore, is adjustable (e.g., for WSN the upper payload limit is approximately 30 bytes). Each capsule contains data and program segments as shown in Figure 3.8. Flags include, for instance, **AUTOUPDATE** (enable/disable autoupdate of capsules), **TIMEUNIT** (switching between *s* and *ms* units for the **LIFETIME** field), dictionary selector (2 bits), **REASM** (used to re-assemble capsules for long programs, this is different to capsule merger discussed below; see Section 7.3), dictionary to use and others. Flags, ID/version and lifetime fields are further discussed in Section 4.4. Data segment size defines the size of the in-capsule data buffer (**BUFC**). All the above is a part of the system segment. The system segment is followed by a data segment. The last also contains immediate volatile (non-fixed) stack operands (not addresses or references to system constants like **ALL**, **NOW**, **ME**, etc.). The rest of the capsule's memory space is occupied by the code segment which extends to the end of the capsule.

ChameleonVM partially separates data and code. Volatile (immediate) operands are removed from the original program stream and are stored separately within data segment. Static variables stay with the code. This allows to reduce code entropy and improve compression.

3. Configurable VMs for Embedded Networking

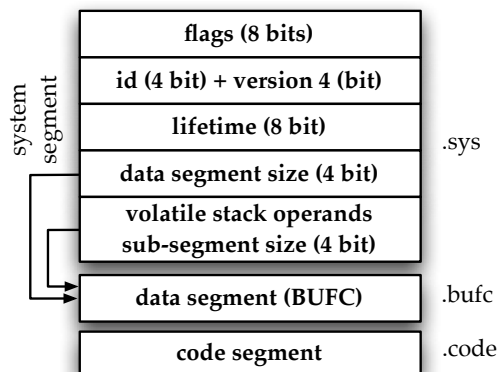


Figure 3.8: ChameleonVM: Capsule Structure

Similar to *Maté*, *ChameleonVM* programs can be composed (deployed) of multiple capsules. Although only initial multi-capsule propagation is allowed. After re-assembling is done no partitioning of code is possible anymore. The reason for choosing such architecture is to make capsules autonomic units without any linking and dependency resolution needed. This simplifies design, makes it more robust and on the other hand it requires non-traditional thinking of how network protocols are built. This is achieved by advanced communication capabilities possible between capsules. Capsules can merge and split forming various contexts depending on the situation. As we show later in Chapter 6 some protocols will have to wait for the compression scheme to reduce them down to the point where they are able to fit their mobile parts in a packet and transmit it.

Merge: Capsules merger allows two or more capsules to combine their computation abilities in order to provide more efficiency to the programs they are used in. The main constraint is that the resulting capsule's size should not exceed the platform-specific limit (in case of WSN it is about 30 bytes). If the resulting capsule exceeds the limit the operation is canceled and capsules stay untouched. Merger is done using `merge` instruction which takes the following arguments: capsule's ID (ID of the capsule the merger should be carried out with), reactant (the part in the original capsule which should be used during merger; the value can be: `ABOVE` (code above the current position), `BELOW` (code

below the current position), **ALL** (entire handler)), reaction type (reactions can be prefix-like or appendix-like; the value can be: **BEFORE** (prefix the code before the corresponding handler in the second capsule), **AFTER** (appendix code after the corresponding handler in the second handler)).¹ The way **merge** operation works is shown in Figure 3.9.

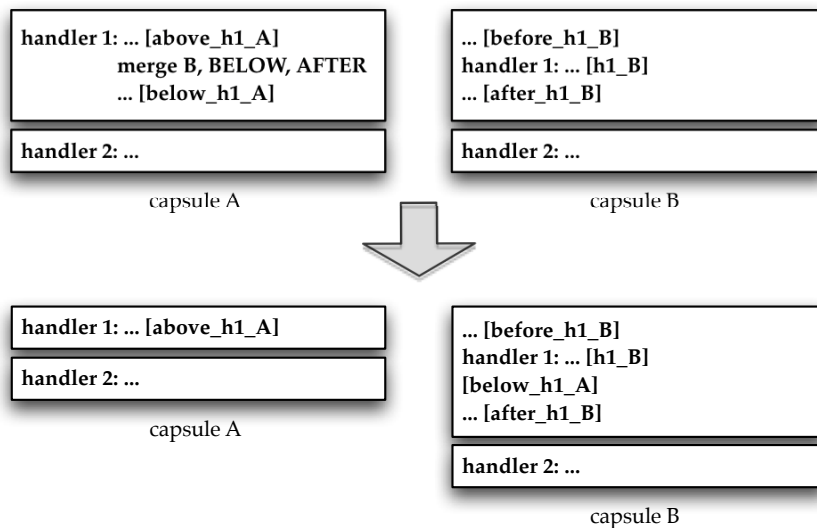


Figure 3.9: ChameleonVM: Merge Operation

There are several constraints with **merge** operation:

- Merger can attach code only to the beginning or to the end of another code piece. Insert is not possible.
- Merger fails if the resulting capsule's size exceeds the capsule's size limit.
- Only handlers of the same type can be merged.
- Flags and other parameters within system segment are inherited from the second capsule (in Figure 3.9 it is B). The same for data segment. This brings some limitations on using in-capsule variables.
- Currently there is no "merger marker". It might be introduced in the future. In this case, the merger range will be

¹Handlers are explained in Section 3.2.3.

3. Configurable VMs for Embedded Networking

limited by those markers rather than segment borders (see `split` operation below).

Clone: Clone is a specific case of merger which preserves the original capsule. Basically, clone operation causes copying some code and removing only the `clone` operator itself from the original capsule. This is shown in Figure 3.10.

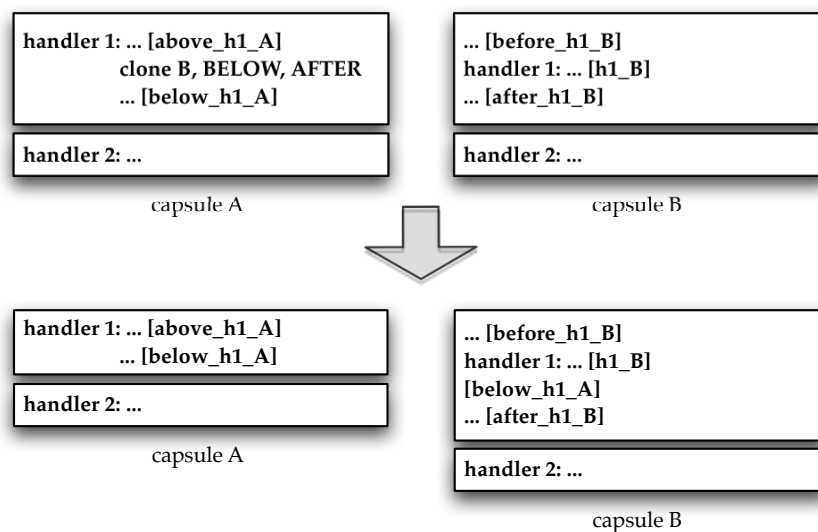


Figure 3.10: ChameleonVM: Clone Operation

All limitation of the `merge` instruction remain true for the `clone` too.

Although there is no obvious need for that but a capsule's copy can be created by sending the capsule to the current node, receiving and installing it.¹ This can be done from the same capsule or from an other capsule.

Split: Split operation allows capsules to break down into smaller pieces. Later, capsules can be reassembled using merger operation. Split breaks capsule into two parts at the position where it appears. If the original capsule has more than one handler it preserves them, the splitting segment is removed, otherwise capsule

¹ChameleonVM will not use radio channel in this case.

is erased as it becomes empty. As a result, two new capsules are created with the following features (also see Figure 3.11):

- System segment is copied from the original capsule.
- Data segment is copied from the original capsule.

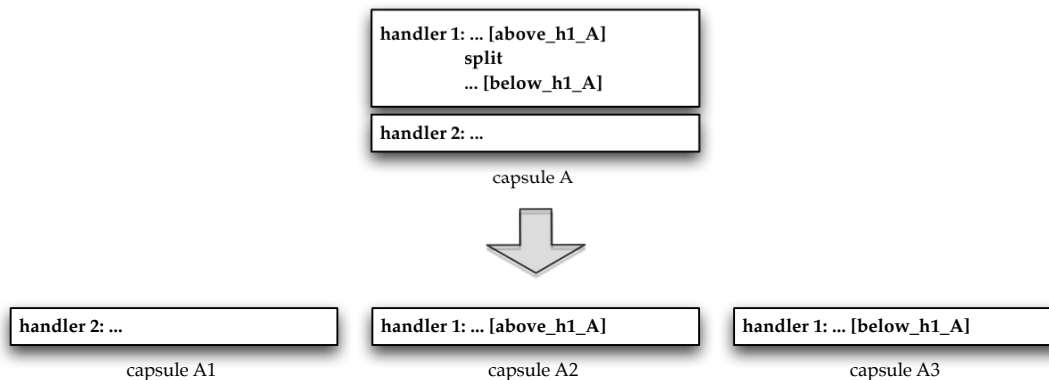


Figure 3.11: ChameleonVM: Split Operation

By using **merge** and **split** the optimal software configuration can be found. We designed **merge** and **split** operations to make re-configuration process efficient. Hence, merger is supposed to enhance functionality by bringing and embedding new computational blocks into the existing framework. Split allows to easily extract unwanted parts.

Capsules cannot remove other capsules from the system. Instead, *ChameleonVM* uses the mechanism of **self-decaying code**. This is achieved by using a lifetime field for each capsule (see Section 4.4).

ChameleonVM is a type-free machinery. Data type (resolution) is defined for each instruction. Type checks are not performed; no type casting is available. Memory is not typed either; each memory region can hold various data types at the same time. Argument type is defined by the instruction accessing it. Stack architecture eliminates the need for variables. This provides better code integration and distribution facilities as no dependency resolution is needed.

Operation polymorphism is natively supported by *ChameleonVM*. This means that operation execution flow depends on the dictionary currently being used (e.g., **add** can mean “add two 8-bit values from

3. Configurable VMs for Embedded Networking

the top of the stack and push the result back” or “add three 16-bit immediate operands and push the result onto the stack”). Instruction polymorphism is further explained in Section 4.3.

3.2.3 Execution Model

Capsules in *ChameleonVM* cannot trigger each others execution. Also, there are no sub-routine (function) calls within a capsule. This limitation is overcome by using on-board dictionary and instruction assignments (see Section 3.2.9) which allows to introduce aliases.

ChameleonVM does not require to reboot each time it receives a new code. Capsules are scheduled for execution as soon as they have been installed onto a node, all needed resources have been allocated and verification has been accomplished successfully. At its time, control jumps to the first instruction of the capsule and executes until it reaches the end of the capsule (if there are multiple capsules, execution may be interrupted by context switching). Merger/split operations stop execution of the existing capsule, newly created capsules are scheduled for execution again. In case if the capsule would like to be executed again this functionality must be programmed in the capsule. `execute` method puts the capsule back into the waiting queue at the point of call. This is needed only for the code within handlers of common type only; packet handlers and timers are executed on-event and do not need to be rescheduled each time.

As it was mentioned above, capsules address- and name-spaces are fully isolated from other capsules in the system. Within each capsule’s code segment one or more handlers can be placed. Handlers are event-driven code sections which are called in response to some event in the system. The type of handlers a capsule has defines to which events it can react.

The following handlers are currently supported:

- Initialization (denoted by `.code.init`): Is executed only once for newly installed capsules. Can be used to initialize some values or setup timers.
- Receive message (denoted by `.code.pack`): Is executed each time a message (everything which is not a capsule or dictionary update) arrives.

- Receive capsule (denoted by `.code.cap`): Is executed each time a capsule arrives to a node.
- Timer (denoted by `.code.timerN`): Currently there are only two timers available (denoted as `timer0` and `timer1`) which should be sufficient for most tasks.
- Common (denoted by `.code.common`): Can be used to perform some auxiliary actions. Re-execution of this handler should be explicitly requested using `execute` instruction from inside itself or any other handler.

Under the hood everything in *ChameleonVM* is done in response to reception of a packet/capsule. This means that `execute` instruction or timer expiry event are signaled by actually sending a capsule of special type to itself. The only difference from a real capsule reception is that *ChameleonVM* does not use radio channel in this case to save energy.

Handlers allow isolation of specific information processing. Moreover, handlers can be borrowed from other capsules using `merge` or `clone` operations. If some handlers are not needed anymore they can be removed using `split` method.

3.2.4 Instruction Set

ChameleonVM instruction set is dictionary based which means that the definition of each instruction is given through one or more on-board dictionaries. It is extensible and changeable. Some primitive instructions (e.g., `push`, `pop`) are mandatory and cannot be removed, but can be modified. Others are compound instructions. For example, `push A B` (pushing 2 values on the stack; it is compound) can be specified via primitive `push` instruction:

$$\text{push A B} = \text{push A} + \text{push B}$$

Instruction polymorphism is naturally supported through overloading. New instructions can be defined. Two categories of newly defined instructions exist: 1) defined via existing instructions, and 2) aliases (calling these instructions causes execution of native pre-compiled functions). As it was briefly discussed in Section 3.2.1 instructions with variable resolution are supported via re-defining them in the dictionary. Note that, for example, for having simultaneously `push` 8-bit and 16-bit versions the system must define two separate instructions.

3. Configurable VMs for Embedded Networking

All instructions are stack-based. However, immediate arguments can be used by instructions as well. Moreover, both types can be mixed within one instruction.¹ The result is always pushed back on the stack. The configuration of arguments is done through the dictionary. Using only stack operand is preferable as it makes program design more straightforward and robust. Figure 3.12 shows all incoming data and code sources for an instruction. This also demonstrates various types of addressing used by *ChameleonVM*.

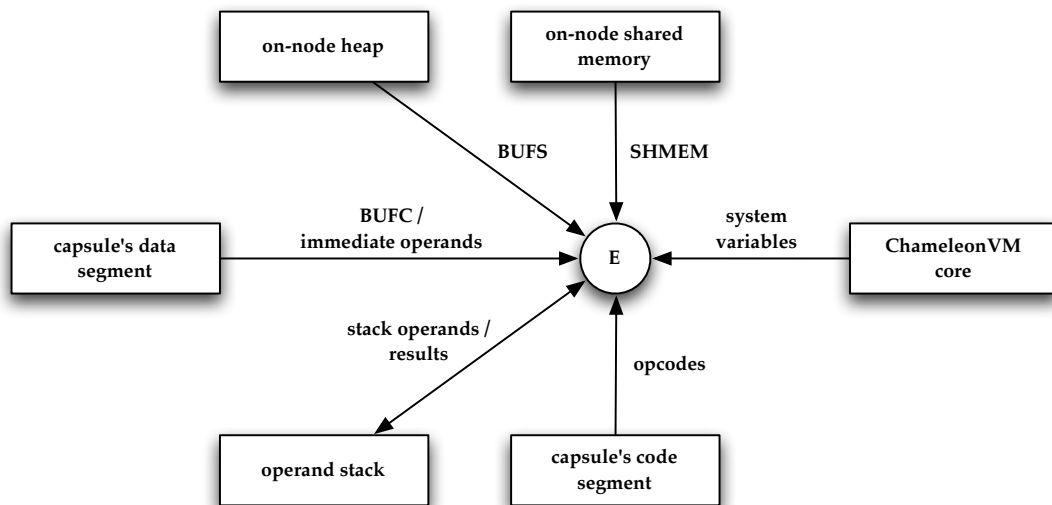


Figure 3.12: ChameleonVM: Instruction Data and Code Sources

The following instruction classes are currently defined:

- Stack: Stack operations manipulate with data stack. These include: `push`, `pop`, `swap`, `dup`, etc.
- Arithmetic: Perform arithmetic computations on the values on the stack. The result appears on top of the stack. These include: `add`, `sub`, `mult`, `div`, etc.
- Binary/Logic: Perform binary/logic operations on the values on the stack. The result appears on top of the stack. These include: `and`, `or`, `not`, `xor`, etc.

¹In this document we usually use notation `instr A B C` to show how many arguments the instruction takes up. They can be fetched either from the stack or given as immediate parameters.

- Control: Execution flow can be managed using `jmpeq`, `jmpplt`, `jmp`, etc.
- Capsule manipulation: Operates with capsules and handlers inside them. These include: `merge`, `split`, `clone` discussed above in Section 3.2.2.
- Communication: Are used to provide communication facilities with other nodes. These include: `send` and `sendd`, `get` (child, parent, nodeid).
- Memory access (heap and shared memory): `read`, `write`, etc.
- Miscellaneous: `set` (nodeid), `led`, etc.
- Aliases: In fact, these instructions are just references to the pre-compiled native functions provided by other layers or other *ChameleonVM* instructions. For example, `sort` implements sort of an array which resides in the shared memory, or `rand` which generates a random number. The first makes a call to the user-defined function, the second is a callback to the OS library function.

Most of the instructions shown above are primitive – they cannot be specified using other instructions. The full *ChameleonVM* instruction set is presented in Appendix C.

Many support instructions like `set` and `get` are specific to WSN implementation. For example, on other platforms `get`'s functionality can be achieved using in-memory inter-protocol interaction as described in Section 3.2.12.

3.2.5 Code Verification

Code verification is an essential part of a system based on mobile code. The fact that code comes from various sources which are not always identifiable, requires treating it with a precaution. *ChameleonVM* verifies incoming code twice: upon arrival and at run-time.

Code verification module (CVM) in Figure 3.5 from Section 3.2 is responsible for the first type of check – upon arrival. The following characteristics of an incoming capsule are checked: 1) integrity (capsule's structure, segment borders), and 2) ID/version pair. Capsules having corrupted internal structure are not passed through, they are simply dumped. In order to check if ID/version numbers are correct, CVM sends a request to the capsule storage. *ChameleonVM* defines a range of allowed capsule IDs. If the received value falls out of this

3. Configurable VMs for Embedded Networking

range, the capsule is rejected. If there is already a capsule with a newer version number in the system, the capsule is rejected too.

Run-time code verification ensures that the system stays operating at all time. *ChameleonVM* keeps track of the following possible collisions which can happen while executing capsules:

- Stack under-/overflow: An attempt to read from the empty stack or write more values than its maximum size.
- Hang-up: Each instruction is served by a watch-dog timer. Timeout is triggered.
- Invalid instruction: If invalid instruction code is met.
- Invalid instructions pattern: If instruction does not fit any pattern in the dictionary map (e.g., wrong number of argument, arguments of wrong type, etc.).
- Address out of range (heap or shared memory): If while accessing memory the address falls out of range.
- Lack of memory: Memory resources are fully occupied.

When one of the situations above occurs the capsule execution is interrupted and the corresponding flag is flipped in the Error Register (see Figure 3.7). This register can be read onto the stack using `push ERREG` call. In the current implementation the infected capsule is just removed from the system. This is not the best possible handling. Alternatively, a “damaged capsule” notification could be sent to the neighboring nodes in response to which those nodes who have a copy of this capsule would send back a healthy replica.

ChameleonVM cannot not detect dead-locks between capsules. Therefore, it is a responsibility of capsules to be sure they do not get blocked on each other. Concurrent access to resources by blocking a particular resource while it is in use by some capsule. It is released when the capsule does not need it anymore.

Re-definition of instructions in the presence of running code may cause errors in code integrity and generate mismatching code versions. It must be done with high precaution.

3.2.6 Code Propagation and Deployment

ChameleonVM does NOT provide a self-propagation feature like for instance *Trickle* in *Maté*. There are two ways to propagate code using

ChameleonVM: 1) a capsule should navigate itself from source to destination (this functionality must be placed inside the capsule), or 2) a capsule can rely on the underlying protocols (MAC, routing) and the status information provided by them.

The following instructions can be useful for building an ad-hoc self-routing in *ChameleonVM*:

- **send W,N**: It is the main instruction to support communication between nodes. It has two arguments: what to send (**W**) and where to (**N**). The following values can be used as **W** parameter (what to send): **ME** (this capsule), **PACK** (the packet being processed if called from `.code.pack` handler – see Section 3.2.3), another capsule's ID given as a numerical value. Parameter **N** (destination) can be: another node's ID, **ALL** (broadcast), **UP** (one hop up the spanning tree; works if only the spanning tree exists), **DOWN** (one hop down the spanning tree; works if only the spanning tree exists), **ANY** (random neighbor, or any random address), **GROUP** (group members).¹
- **sendd W,N,DN**: In contrast to **send** which uses an absolute, next-hop address (**PACK.TO**) **sendd** also allows to specify an address of the final destination as a third parameter (e.g., top of a spanning tree; **PACK.DST**). Otherwise, two instructions are identical.
- **get W**:² Is used to get the communication data from other layers. The only parameter **W** denotes what exactly we need. It can be: **NUMN** (number of neighbors), **CHILS** (node IDs of all children), **CHIL** (node ID of a random child), **PAR** (node ID of the parent), **NID** (host node ID), **ME.ID** (this capsule's ID), **GROUP** (group's ID this capsule belongs to).

As can be seen communication primitives can use many parameters provided by the underlying layers. The presented set is specific for WSN but can be defined for other platforms as well. Additionally, this requires a very close integration between capsules and the existing protocols (see Section 3.2.12).

Every capsule in *ChameleonVM* has two flags: **AUTOUPDATE** and **AUTOEXEC**. **AUTOUPDATE** (default is on) allows to control capsule prop-

¹This is specific to *TinyOS*.

²**get W** is equivalent to **push W**.

3. Configurable VMs for Embedded Networking

agation and installation. If on then new capsules are installed automatically. `AUTOEXEC` (default is on) specifies if the capsule should also be executed automatically upon installation. Execution of postponed capsules can be triggered by other capsules. This feature is useful for propagation of the initial code set, especially in case of long programs which are propagated using multiple capsules.

In cooperation with code versioning (see Section 4.4) the presented communication primitives are sufficient to embed a self-propagation feature into a capsule.

In contrast to *Trickle* used in *Maté* which propagates, installs and executes the same code on each node in the network *ChameleonVM* follows different ideology. Capsules can decide themselves where in the network to go to execute their code. No dissemination support is provided. A simple acknowledgment scheme is used for code exchange to provide some degree of communication reliability. Further enhancements can be done at the application level.

3.2.7 Packet Processing

One of the main *ChameleonVM*'s functions is packet processing. In mind we have the scenario when a set of capsules reside on each node and by switching between them a various types of packet processing could be accomplished. Therefore, for example, there is no need to have pre-defined routing rules and tables, or even a specialized scripting engine [Gra10].

If a capsule wants to be capable of doing packet processing, it should implement `.code.pack` handler. After that the capsule will be notified about all incoming packets (which are not capsules or dictionary updates). If multiple capsules provide `.code.pack` handler a packet is processed by all capsules sequentially. For example, if capsules A and B want to process packet P, then first P is processed by A (or B), then it is passed to B (or A). The order of processing (A,B or B,A) is not pre-determined and cannot be fixed. While the packet is being processed, an access to it for other capsules is blocked to avoid race conditions. If the packet is erased or sent out before some capsule gets access to it the notification request is annulled. This avoid packet duplication which may occur. Locally generated packets can be first processed by the locally installed code. For this we would need to send

packets first to ourselves. To do it from outside, this trick requires to use a separate system call provided by *ChameleonVM*.

In a similar manner `.code.cap` can be used to process incoming capsules.

3.2.8 Node-To-Node Communication

We can distinguish several communication “threads” between nodes using netware and equipped with *ChameleonVM*. Those “threads” include:

- Exchange of capsules.
- Exchange of data packets: Any type of packet which is not a capsule or a dictionary update. This includes, for example, MAC-layer and routing packets.
- Announcement of dictionary updates: Dictionary updates are sent in order to keep nodes’ dictionaries synchronized. More on dictionary synchronization see Section 3.2.10.

These threads run in parallel. The collisions are managed using either a pre-deployed MAC-layer or at the application level. Two nodes communication model can be seen as shown in Figure 3.13.

Multi-node communication model is just a scaled version of the picture above.

3.2.9 On-Board Dictionary

An on-board dictionary is the central part of our design. It serves as a semantics holder for Instruction Set Architecture (ISA). At the same time, it is a data bank on which constant instruction re-encoding is performed. Dictionary records can be updated through so-called dictionary updates sent by other nodes. Alternatively, they can be changed from outside, by external applications using *ChameleonVM* system calls (see Section 3.2.12). In the current architecture *ChameleonVM* requires a single session leader. This session leader carries out compression and sends out updates to the rest of the network. This is different from *FragletVM* discussed in Section 3.3 where updates can be initiated by any node (see Sections 3.3.8 – 3.3.10); in other words *FragletVM* employs a fully distributed architecture. The session leader is must be elected (pre-configured) before the compression process starts. Currently, this is done manually.

3. Configurable VMs for Embedded Networking

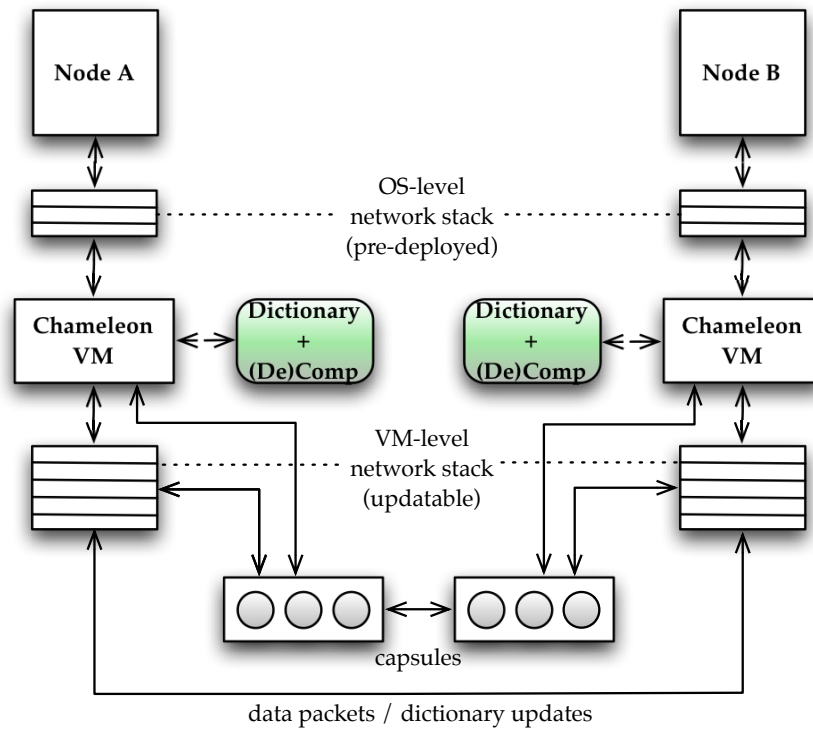


Figure 3.13: ChameleonVM: Communication Model Between Nodes

Each node can have more than one dictionary. Switching between dictionaries is done via command sent by the leader. Each capsule has a dictionary pointer which reflects this command (see Section 3.2.1).

Each record in the dictionary has the structure shown in Figure 3.14. In fact, *ChameleonVM* dictionary is a mapping allowing to form new instructions out of those which are already in the dictionary. Dictionary start growing from a set of primitive instructions (see Section 3.2.4), like stack, arithmetic, etc.

Each opcode in the dictionary is a linked list of one or more instructions, a pool of operands and an opcode specification. The list ends with an **end marker**. In turn, each member of the list can be an entry on itself (with the same structure): in Figure 3.14 $opcode_{i-1}$ is a sub-entry of $opcode_i$ entry. The dictionary ensures to not have mutually nested entries.

The operands pool is a set of parameters passed while call $opcode_i$ is made. When $opcode_i$ is called, the execution is passed to $opcode_{i1}$. The operands for $opcode_{i1}$, which is essentially $opcode_{i-1}$, are picked from

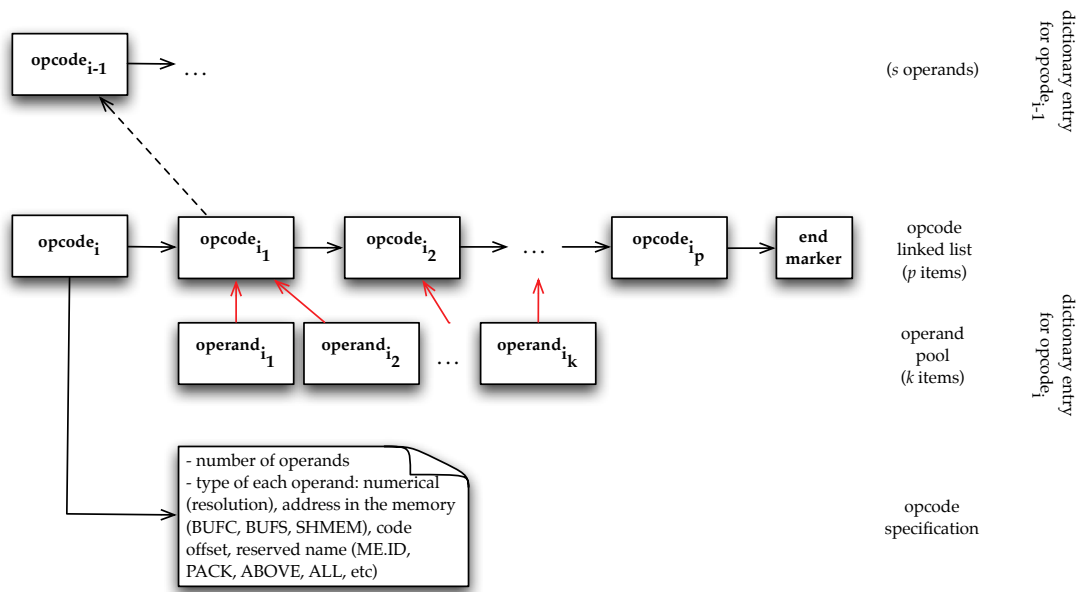


Figure 3.14: ChameleonVM: Dictionary Structure

the operands pool according to the specification of `opcodei-1`. Then `opcodei2` is called and the next set of operands are picked. The process continues until the end marker is reached. The number of operands in the operand pool of `opcodei` must be equal to the sum of operands of all opcodes in the list:

$$\sum_{j=0}^k operand_{ij} = \sum_{l=0}^p opcode_{il} \sum_{t=0}^s operand_{ilt}$$

If the condition above fails the run-time execution error is indicated. The error is triggered too if the argument type is incorrect (see Section 3.2.5). The type for each argument is given by opcode specification table. It includes the number of operands, and a type for each operand. The last can be a numerical value (in this case its resolution is specified), as address in one of the memory regions (BUFC, BUFS or SHMEM), code offset (for flow control instructions like `jmp`) or a reserved name which is translated by *ChameleonVM* in the actual object at run-time (for example, `ME.ID` is converted into numerical value representing capsule's ID).

to have up-to-date copies the update process relies on distributing changes only, the delta from the current copy. The following changes can be made to a local copy of the dictionary:

- Add a new instruction: A new instruction (with a new opcode) is added to the dictionary. In the example for `jmpeq L1 5` instruction from Section 3.2.9 the update command might look as shown in Figure 3.16.

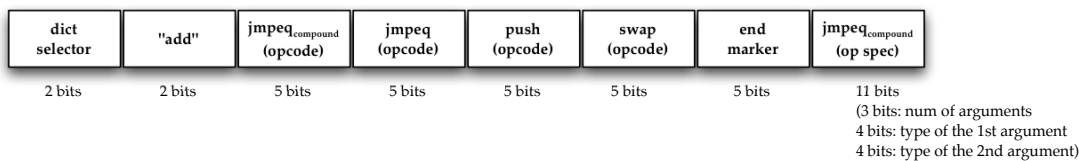


Figure 3.16: ChameleonVM: “add new instruction” command

In this example, we have 5-bit encoding scheme (each instruction is encoded using 5 bits). Therefore, the total size of the message is 40 bits.

- Remove an instruction: An existing instruction is erased from the dictionary. For `jmpeq L1 5` instruction the update command would look as shown in Figure 3.17.



Figure 3.17: ChameleonVM: “delete existing instruction” command

The total size of the update message is 9 bits.

- Update an existing instruction: An existing instruction is changed. For `jmpeq L1 5` instruction the update command would look as shown in Figure 3.18. “Update” instruction is similar to “add”, the new semantic rule has to be forwarded.
- Protection enable/disable: This allows to set/remove the flag which protects each instruction from changes. For primitive

3. Configurable VMs for Embedded Networking

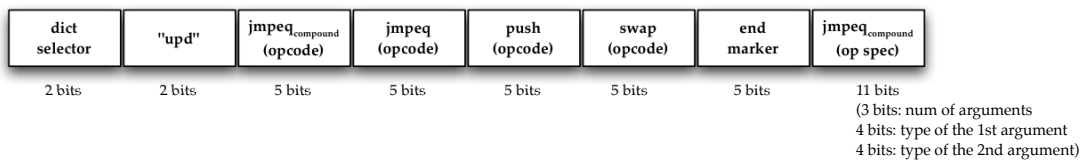


Figure 3.18: ChameleonVM: “update existing instruction” command

instructions (stack, arithmetic, etc.) this flag is enabled by default. If some instructions are not needed they can be removed. The functionality for primitive instructions cannot be removed from a node, though, as it is a part of the *ChameleonVM* core. The command format is shown in Figure 3.19. Command encoding takes 10 bits.

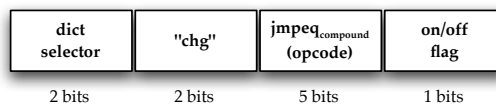


Figure 3.19: ChameleonVM: “enable/disable protection from changes” command

As has been said before, *ChameleonVM* employs the centric-oriented dictionary update/synchronization scheme. This is the simplest possible configuration which allows us to explore the main properties of the synchronization process. The network has a leader as shown in Figure 3.20. It should not necessarily be an access (sink) node. The leader constantly performs code analysis and assignment of new instruction codes. Afterwards it sends out an “add new instruction” message to neighboring nodes which propagate it further. Each node has to acknowledge that it has received an update and agreed on it. Only after acknowledgments from the entire network has been received, the change is applied. Local dictionary copies of non-leading nodes are not synchronized between each other but with the leader’s copy only. This architecture can be extended to multiple scattered profiles as discussed in Section 1.5.1 and shown in Figure 1.6a.

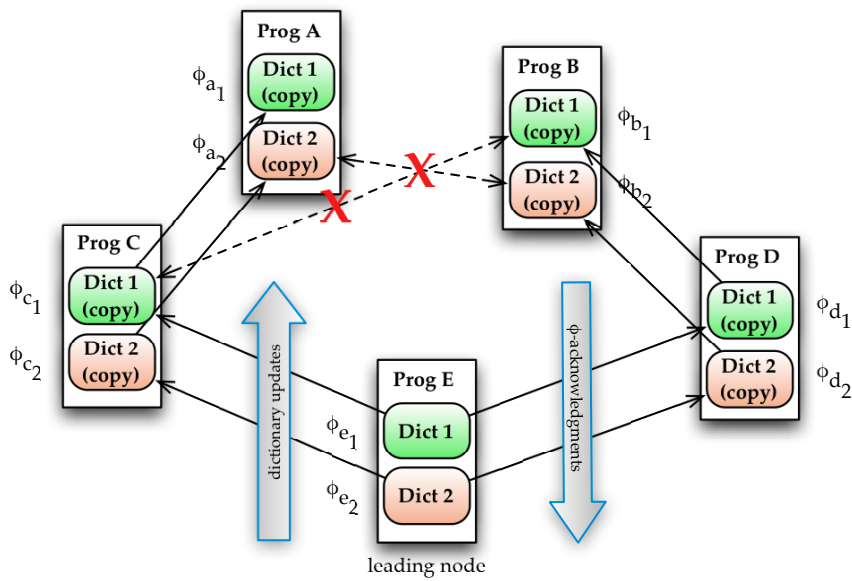


Figure 3.20: ChameleonVM: Propagation of Dictionary Updates

Also, as shown in Figure 3.20 a node can hold more than one dictionary at a time. Each capsule in the system segment has 2-bit dictionary selector (see Figure 3.8) which is copied to the dictionary selector register when capsule is installed on a node (see Figure 3.7). Switching between dictionaries can be initiated from code using `dict N` instruction which takes dictionary number to switch to as a parameter. If the node does not possess a certain dictionary it ignores the corresponding updates and commands.

In order to improve dictionary synchronization robustness, a sort of periodical consistency check can be used as well. The acknowledgment for updates sent back by nodes is calculated as an **integrity function** ϕ over the current dictionary copy. We propose to use a hash-function like CRC over the set of opcodes. The reason for that is that CRC is simple and fast to compute and validate, and it is easily adaptable to data resolution via changing its polynomial length. On top of per-update acknowledgments a periodical integrity check cycle could take place: nodes periodically sent out its integrity function, other nodes check and confirm. In the example in Figure 3.20 all ϕ_{i1} must be identical. The same for all ϕ_{i2} . If integrity check fails, the node could claim a new copy of the dictionary from the leader. Moreover, integrity check could

3. Configurable VMs for Embedded Networking

be organized in a distributed fashion, neighbors watching might be sufficient and no centric approach is needed.

3.2.11 Compression and Decompression

Compression is performed on outgoing capsules only. They are marked upon arrival, later only those capsules participate in the compression process. Decompression is done when the capsule arrives to a node. In the capsule storage (see Figure 3.5) all capsules have uncompressed representation in order to speed up their execution. Although it can be done in real time as well – dictionary-based architecture allows this.

Compression is a continuous process. It takes time for the system to converge to the optimal instruction encoding. In some cases as discussed in Chapter 5 the encoding can diverge from the optimum temporarily but eventually the system will find the way. Optimization of the instruction encoding belongs to the class of NP-problems. Obviously, ES do not have enough resources to completely solve it. Moreover, the task must be carried out at run-time yet keeping a memory footprint as small as possible. All these restrictions were taken into account while developing the compression scheme architecture we present in Chapter 5.

3.2.12 Support for Existing (External) Software

Before we have mentioned that *ChameleonVM* provides the support for existing software (network protocols). This support is given through a set of interfaces which external protocols can use. The interaction is made via memory – external protocols can write/read to *ChameleonVM*'s stack, shared memory (**SHMEM**), on-node heap (**BUFS**) and individual capsule's memory buffer (**BUFC**). By sharing memory resources, capsules and existing protocols can exchange necessary data. For example, the existing routing protocol can be configured to always write the current "next hop" address to a corresponding capsule's internal buffer, then the capsule can use this value to forward itself. The capsule does not have to make any extra write/read calls; it should not even know where this value comes from. Other exchange values might include: an upstream hop in a spanning tree, an aggregated value, a number of neighbors, etc. Alternatively, a capsule can leave some value in the buffer which can be later used by some external protocol.

We give a description of support calls in a form of C-function prototypes as shown in Table 3.2. The calls in Table 3.2 have a number of restrictions. Currently, aliases cannot have a return value and can only accept numerical (integer) parameters (example with `led` instruction). All calls for memory access do not take effect if memory boundaries are not respected (if copying buffers exceed the maximum allowed size or out-of-range addresses are used). Capsules cannot be manipulated through those calls; *ChameleonVM* keeps an exclusive access to capsule storage, they cannot be called or sent via aliases. A special care must be taken with using aliases. As an alias contains an address of the local function it should NOT be used in migrating capsules.

Call	Description
<code>void push(int capid, char *buf, int length)</code>	Push on the stack of the capsule with ID <code>capid</code> . The data of <code>length</code> is taken from <code>buf</code> .
<code>char *pop(int length)</code>	Pop from the stack of the capsule with ID <code>capid</code> . The data of <code>length</code> is copied to the main memory. The pointer is returned.
<code>void write_bufc(int capid, int offset, char *buf, int length)</code>	Write the internal buffer of the capsule with ID <code>capid</code> starting from the position <code>offset</code> . The data of <code>length</code> is taken from <code>buf</code> .
<code>char *read_bufc(int capid, int offset, int length)</code>	Read the internal buffer of the capsule with ID <code>capid</code> starting from the position <code>offset</code> . The data of <code>length</code> is copied to the main memory. The pointer is returned.
<code>void write_bufs(int capid, int offset, char *buf, int length)</code>	Write the heap of the capsule with ID <code>capid</code> starting from the position <code>offset</code> . The data of <code>length</code> is taken from <code>buf</code> .
<code>char *read_bufs(int capid, int offset, int length)</code>	Read the heap of the capsule with ID <code>capid</code> starting from the position <code>offset</code> . The data of <code>length</code> is copied to the main memory. The pointer is returned.
<code>void add_alias(void (*func_ptr)(...))</code>	Create an alias for the function <code>func_ptr</code> .
<code>void del_alias(void (*func_ptr)(...))</code>	Delete an alias for the function <code>func_ptr</code> .

Table 3.2: ChameleonVM: Support Calls for Existing Protocols

As Table 3.2 external software can also directly access and change dictionary records. Only access to aliases is provided. By using these

3. Configurable VMs for Embedded Networking

methods an alias to an external function call can be created in the dictionary.

One of the further improvements of the mechanism above is called **exported functions**. A capsule can export its functionality (for example the `.code.common` segment, or a pre-compiled library function) as a function which will permanently reside on a node and could be assigned an instruction to make a call to. Export functions provide more uniform access to the external functionality.

3.2.13 Exported Functions

An **export function** is a special instruction in ISA which refers to a pre-installed code. In our case, when a dictionary basically describes each instruction as a sequence of simpler operations there is not so much difference between standard instruction and exported functions. The only difference is that exported code is not stored inside the dictionary. Calling such an instruction is similar to making a function call with limited parameters. Hereinafter, aliases are referred to as export functions and vice versa.

We use the export function concept in one of our examples later in Section 6.4.6. The pre-compiled function `getSkewAdjust()` (Listing A.6) is distributed as a module, registered in *ChameleonVM* namespace, assigned an opcode in ISA and called using this opcode from other capsules (see Listing A.4).

3.3 FragletVM

FragletVM is the first attempt to create an execution environment for chemical approach of networking and its descriptive language called Fraglets [Tsc03]. The original work on Fraglets [Mey10] has been primarily based on using the Fraglets simulator. Our sole task is to design a prototype implementation for ES showing basic functionality of the Fraglets; hence we do not implement the entire instruction set and some extra advanced features. More focus is put on how to enable easy embeddability and meet specific requirements of resource-limited devices. We take WSN as a target platform.

FragletVM operates with so-called **vessels**, a non-ordered collection (a.k.a., **soup**) of small code pieces, **fraglets**. A typical vessel is shown in Figure 3.21.

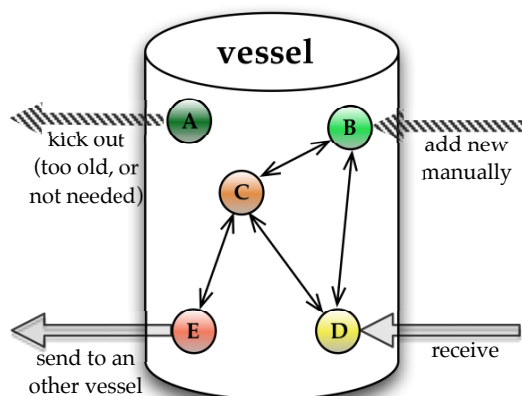


Figure 3.21: FragletVM: Vessels

Fraglets in a vessel interact with each other using `match*` instructions and probabilistic approach. During this interaction some fraglets may become obsolete,¹ or naturally disappear, or sent out. New fraglets may be created as a result, or received from other nodes, or added manually. That is how fraglets population is maintained.

Normally, a node has one vessel. A vessel can contain **sub-vessels** (code sub-group with isolated name space; fraglets can be moved in/out of such a group) but for the reason of memory-saving these are not available in the *FragletVM* implementation (see Section 3.3.2).

3.3.1 Fraglets Language

Fraglets is a prefix-like (a.k.a., tag- or header-rewriting) language used to model networks in a form of chemical reactions. The basic idea consists in using some fundamental properties from chemistry to represent in-network activities. To this end, the following notions are introduced:

- **Molecule**,² an independent program task similar to capsules; see Section 3.2.
- **Reaction**, an interaction of two code pieces, i.e., molecules, resulting in producing a new one, consuming an existing one and/or mutating the reacting molecules.

¹Although there is no lifetime concept in *FragletVM* as the one in *ChameleonVM* (see Section 4.4) fraglets which have not been touched for quite a while are removed from the system by the engine. This reflects the self-decay process.

²Also referred to as **fraglet** but **molecule** more likely represents a functional entity.

3. Configurable VMs for Embedded Networking

- **Mutation**, the ability of code to mutate over time.
- **Concentration** is used as a numerical estimate of some parameter or a value which is indicated through the multiplicity of some molecule.
- **Vessel**, an execution context.

The distinguishable feature of the Fraglets language, as of any prefix-like language, is that a type of the next item in the code stream is defined by the result of the previously executed items. The type can be: named operand, value or instruction.

Fraglets are composed by concatenation of symbols. Fraglets can be of different lengths. In the original version there are three different symbol classes:

- **Instruction**, a finite set of symbols denoting a transformation or a reaction, e.g., `dup`, `match`, `ssum`, etc.
- **Identifier**, a string that tags a fraglet (passive fraglet; see below), a variable name, e.g., `x`, `count`, etc.
- **Value** of various possible types (string, integer, float, etc.), e.g., `"x"`, `5`, etc.

For our implementation we had to make the following changes and apply certain restrictions to the above definitions:

- **Instruction**, an opcode denoting a transformation or a reaction, e.g., `dup`, `match`, `ssum`, etc. (encoded in a certain way).
- **Tag**, a synchronization symbol halting an execution of the current fraglet till the next match is met, e.g., `x`, `count`, etc. (encoded in a certain way).
- **Argument**, a binary value (8-, 16- or 32-bits). Variable-length arguments are not supported at the moment.

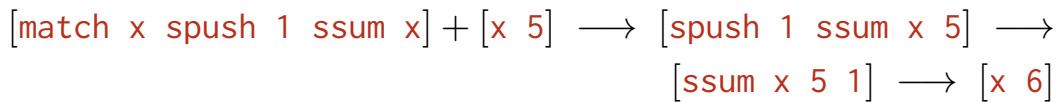
At a time, each fraglet can belong to one of the two main classes:¹

- **active**, a fraglet which starts with a `match`-like instruction, or
- **passive**, a fraglet which starts with a synchronization symbol.

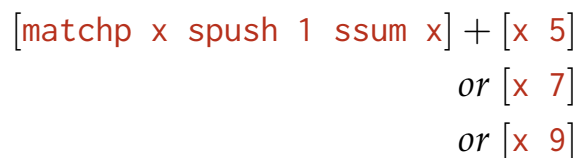
¹Obviously, a fraglet can move classes during execution.

The third class, let call it **immediate**, starts with a constant value or an immediately consuming instruction (e.g., **sum**). More on how different classes of fraglets interact see Section 3.3.4.

In order to demonstrate how fraglets are composed and interact with each other let us consider a trivial example:



In the example above we perform an increment of the variable x with an initial value of 5. The order of execution is not guaranteed. Lets consider the following setting:



Execution model does not guarantee which value will be incremented first. Execution order follows mass action and randomization laws (the more the concentration the higher the probability of execution is, and as a consequence, if two reactions are equally probable then the next reaction to execute is picked randomly; see Section 3.3.4). This is similar to execution of desynchronized program threads. If we would like to have those reactions synchronized we would need to use some type of markers (similar to the idea of UNIX semaphores).

The following change to both fraglets (active and passive) makes only one pair to react:



Alternatively it can be done this way (only active fraglet is changed):



Before we discuss more complex aspects of the system architecture let us consider the example from Section 3.2, an alarm sensor, and show how it can be implemented using fraglets. The program in Listing 3.2 is similar to the one from Listing 3.1. It is continuously taking temperature

3. Configurable VMs for Embedded Networking

measurements on a node and if the temperature drops below 30 °C then the red LED goes off otherwise it is lit on. The only difference is that the fraglets implementation does not have a timer. Otherwise, both implementations are identical.

```
1 # Example: switch red led ON if temp raise beyond 30 C, switch
2 # OFF otherwise
3
4 [sense T TEMP] --> [T X]
5
6 [matchp T gt G L 30] + [T X] --> [gt G L 30 X] --> [G 30 X] or
7                                     [L 30 X]
8
9 [match G led 1 1] + [G 30 X] --> red led ON
10
11 [match L led 0 1] + [L 30 X] --> red led OFF
```

Listing 3.2: Simple Alarm Sensor in FragletVM

3.3.2 System Architecture

FragletVM is built on top of an OS and uses system calls to perform low-level operations (e.g., sending a fraglet).

Run-time environment: *FragletVM* must be pre-installed on each node in the network. One node plays a role of an “access point” (or a “base station”) to send control commands and upload code to the network and to collect various run-time information. Programs can be injected into the running network via access node in either of two ways: “all together” (the front-end software reads a source file, injects molecules (fraglets) on each node and triggers their execution at the end), or “one-by-one” (fraglets are injected manually, code execution is initiated manually as well). Additionally, *FragletVM* offers an option to execute molecules one-by-one inspecting the result of each operation. This may be useful for debugging purposes. Currently, there is no way too remove installed molecules (fraglets). This somehow follows one of the main principles of CNP [MT09] which stimulates developing of self-regulating protocols: we can inject another molecule that will react with and destroy the undesired one.

Program and data memory: There is no program or data memory in *FragletVM*, all logic and data is encapsulated in fraglets. Therefore, we

use a specific form of storing fraglets in our system. In fact, it is a big hash table with tags used as keys and fraglets tagged with those keys as values. All programs are executed in the same program space. Logically all fraglets can be grouped according to their tag. Those groups we will call **classes**. Classes can interact by using the same tag at some point. Each fraglet can be seen as a small stack-oriented program of one of two types: “left-to-right” vanishing (fraglet is consumed sequentially from the head to the tail) or “dual” vanishing (fraglet is consumed sequentially from the head to the tail; the tail is used as a data stack for some instructions). This is shown in Figure 3.22.

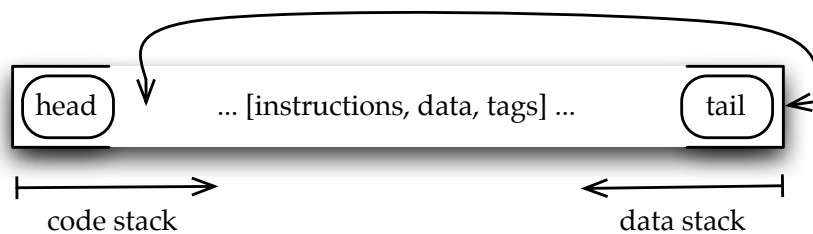


Figure 3.22: FragletVM: Fraglet Structure

Data types: There is no distinguish between code and data. Data fields can be occupied by instructions, and, therefore, data can become executable at some point. Each instruction has a certain structure and number of arguments (on code or data stack); in turn, each argument has a certain type. From the *FragletVM*'s perspective everything is seen as a bit stream where data can be 8-, 16- or 32-bit long chunks. Extensible data types may be introduced later.

Program Execution Table (PET): As it was mentioned above, fraglets are stored inside *FragletVM* as a big hash table with the structure shown in Figure 3.23.¹

The PET is being continuously re-written as reactions are executed according to the next reaction selection algorithm described in Section 3.3.4. The algorithm operates on PET.

¹[`match x`] and [`x`] prefixes are actually not stored in this table for memory saving purposes, only tails are stored.

3. Configurable VMs for Embedded Networking

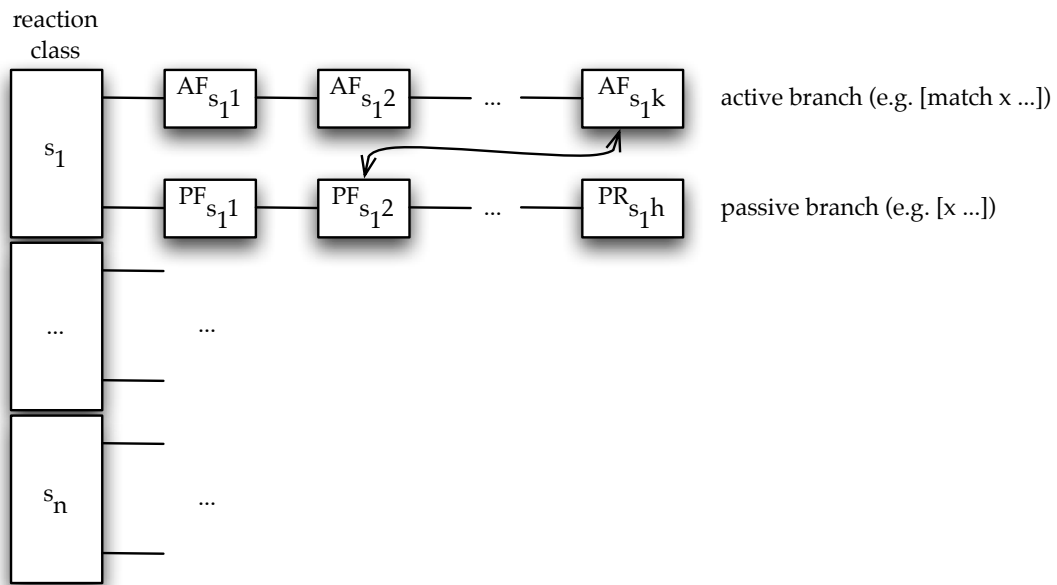


Figure 3.23: FragletVM: Program Execution Look-Up Table

Variables, namespaces and shared memory: *FragletVM* does not offer data memory. Therefore, there are no variables. However, tags can be seen as some sort of variables. All fraglets are executed within one single namespace called *soup*, meaning that all those fraglets can interact which match. All memory resources of the *soup* are shared.

Vessels and sub-vessels: In the original implementation there is the notion of vessels and sub-vessels. Although the latter is not available in *FragletVM* we briefly explain it here. Fraglets can be incorporated into sub-vessels to separate their execution from those in the rest of the *soup*. This means that only fraglets within the sub-vessel will interact. Fraglets can be moved in/out a sub-vessel using *inject/expel* instructions. Sub-vessels can be nested (see Figure 3.24).

Resource management: In order to avoid problems with sharing of resources all operations are blocking. This means the next instructions can be executed if and only if the previous one has been completed and all resources have been released. This should be taken into account while using operation like *send* which may take significant amount of time to complete.

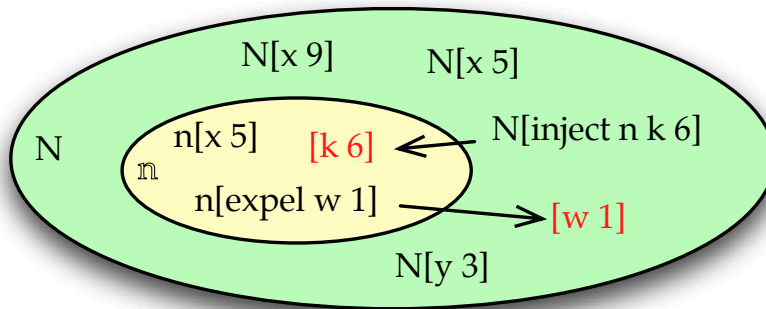


Figure 3.24: FragletVM: Sub-Vessels

Instruction dictionary: *FragletVM* uses similar dictionary-based approach to specify its instruction set as *ChameleonVM* (see Section 3.2.9). Structurally they are absolutely identical. Moreover, the update mechanism is the same too (see Section 3.3.9).

Error handling: *FragletVM* controls the following execution errors: 1) invalid opcode (invalid opcodes are consumed immediately), 2) memory overflow, 3) fraglet oversize, etc. All current errors can be observed using `erreg` instruction which extracts the content of the system error register.

Trying to generally compare the *FragletVM* and the original design from [Mey10] we can say that the former lacks some features making the system fit into memory-constrained devices. For example, *FragletVM* does not support **sub-vessels** (similar to functions) and **interfaces** (similar to sockets). Many original instructions are excluded or simplified.

3.3.3 Programming Model

Fraglets programming model differs from the conventional, sequential programming style. However, nearly the same functionality can be achieved using Fraglets as with traditional programming paradigms.¹

¹One of the exceptions would be “if-no” branching which is not possible using current *FragletVM* instruction set.

3. Configurable VMs for Embedded Networking

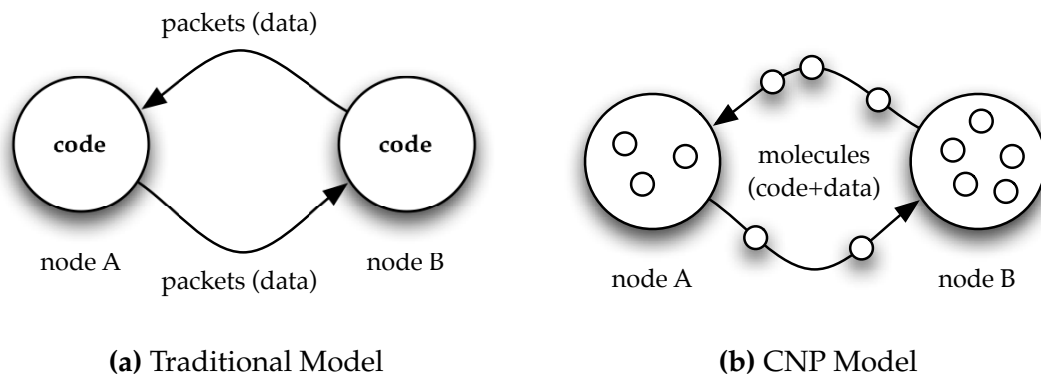


Figure 3.25: Traditional vs CNP Programming Models

In contrast to sequential execution Fraglets programs normally do not give time guarantees.

Programs can be built in either of two ways: as a one single fraglet or as a set of interacting fraglets. A fraglet, in fact, can be seen as an encapsulated program having two stacks: code stack at the head and data stack at the tail. Which stack is addressed at the moment depends on the instruction type used. Fraglets react according to the matching tags at a certain moment in time.

Using only symbolic features of Fraglets is enough to create fully functional programs. However, in this case those programs would be similar to their more traditional mobile-code counterparts, i.e., programs based on *ChameleonVM*'s capsules (see Section 3.2). In CNP the focus is put on designing network protocols using dynamic nature of chemical reactions. Protocol operations are controlled using the concept of **concentration**. This is easier to understand if we start thinking about network traffic as of packet flows each having a certain number of molecules of some type, i.e., concentration, as shown in Figure 3.25.¹ For this case Fraglets offer all necessary tools as well. In Section 6.5.1 we discuss a typical example of a protocol built on the principles of CNP.

¹Concentration of molecules is used to represent some parameter (value) instead of representing it with a data field.

3.3.4 Execution Model

Fraglets execution model incorporates various concepts from chemistry: reactions, mutations, etc. Fraglets reactions are modeled using a probabilistic method rather than a deterministic traditional program execution. The next reaction from the soup is picked according to its probability to happen. In the artificial chemistry the most famous method to model such a system is called *Law of Mass Action (LoMA)* [Gil77], [MT11]. It has been shown that this method gives the best fairness regarding the selection of a next reaction. In our implementation we used an adapted version of LoMA for embedded devices which required much less memory than the original design.

The original LoMA-based look-up algorithm over the PET (see Section 3.3.2) which allows to choose the next reaction to execute looks like as follows:

1. Calculate the weight w_{s_i} of each reaction class.
2. Pick the next reaction class according to its weight and a selective algorithm based on randomization (one possible selective algorithm is shown below): S_1 . The original implementation also schedules the reaction for the time point $\tau = \frac{1}{W}$ (see how W is calculated below). Here we assume that instructions are executed sequentially – without time gaps and overlapping. Delays are implemented with timers. However, *FragletVM* also provides the way to execute programs on “ticks”. In this case, execution flow is clocked at some frequency (currently, granularity is seconds); blocking operations are not interrupted and molecules are not thrown out if timings cannot be met.
3. Pick randomly¹ two molecules (fraglets) to react: AF_{s_1i} and PF_{s_1j} (always one active and one passive are involved in a reaction).
4. Concatenate two fraglets (active with a corresponding passive).
5. Execute all the instructions till we meet the next passive tag (a.k.a., variable name or synchronization symbol).
6. Insert the new fraglet into the look-up table to the appropriate class according to the new tag.

¹This better represents chemical reactions model but we are also thinking about introducing prioritized queues instead, which might bring even more fairness to the model.

3. Configurable VMs for Embedded Networking

7. Loop. Goto step 1.

A version of the selective algorithm based on randomization to choose the next reaction class, i.e., a tag, to execute is presented below (the example is based on Figure 3.26):

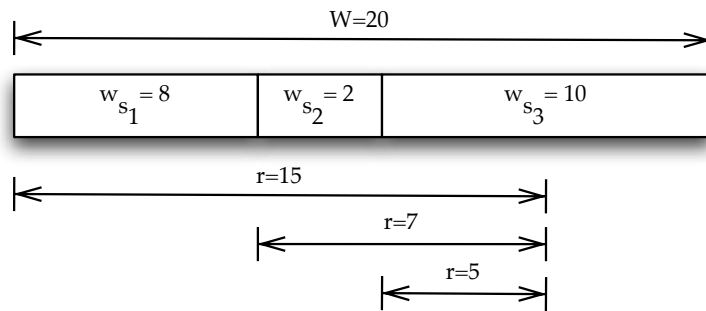


Figure 3.26: FragletVM: Original Selective Algorithm

1. Calculate the weight for each reaction class: $w_{s_i} = |AF_{s_i}| * |PF_{s_i}|$
2. Calculate the total weight: $W = \sum_{i=0}^n w_{s_i}$
3. Generate a random number on the interval $[0, W]$: $r = 15$
4. Keep checking all sub-intervals before the first match:

$$\begin{aligned}
 15 < 8 &\longrightarrow \text{FALSE} \\
 r - 8 = 15 - 8 = 7 < 2 &\longrightarrow \text{FALSE} \\
 r - 2 = 7 - 2 = 5 < 10 &\longrightarrow \text{TRUE}
 \end{aligned}$$

5. Pick S_3 as the next reaction class.

In our implementation we employ the ideas from the algorithms shown above but in order to save memory space and computation time we avoid keeping track of every single weight. Instead each node has an array-based PET of equal size. Next reaction selection is made by choosing two random reactions from PET as shown in Figure 3.27.

The free space provides the necessary probabilistic effect. A reaction happens if the following requirements are met: 1) one passive and

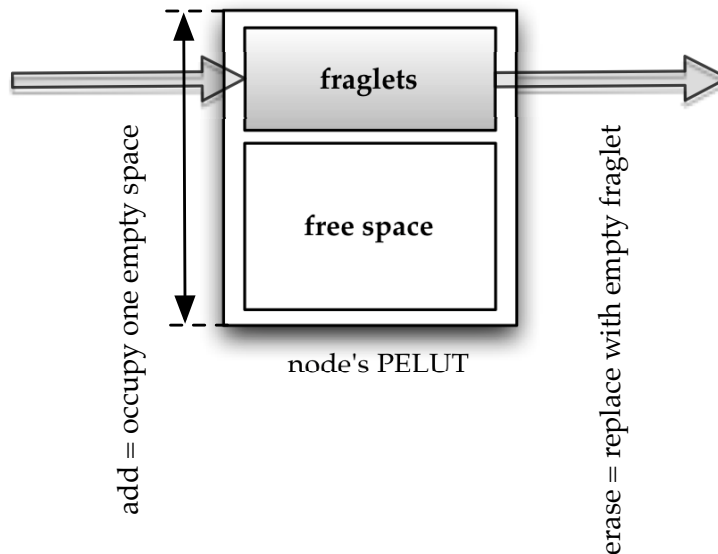


Figure 3.27: FragletVM: Customized Selective Algorithm

one active fraglets have been selected, 2) synchronization tags in two fraglets match, and 3) reactions are valid. If at least one of these requirements fails (i.e., an empty fraglet has been picked) nothing happens and the system goes to the next iteration of the selective algorithm.

As soon as the reaction has been picked and executed, the program execution process continues until the next synchronization tag is reached or the reaction is completely consumed. This means that all computational operations are carried out immediately for the entire fraglet.

3.3.5 Instruction Set

The full Fraglets ISA specification can be found in [Mey10]. For the embeddable version we have picked only necessary instructions. Not the entire original set is supported. Moreover, some of the operations have been modified to simplify their execution.

There are two variants of most of the instructions, one that consumes arguments from the head (immediate) and another that consumes arguments from the tail of a fraglet (stack). Not all the instructions are present in both versions. One of the fundamental concepts is that each operation must result in consuming at least one argument.

3. Configurable VMs for Embedded Networking

We enforce the product to be smaller; hence the instruction format. This is done to avoid uncontrolled growing of fraglets which are constructed or interact improperly.¹ The currently supported instructions are presented in Appendix D.

Encoding of fraglets is done in a straightforward fashion. Each fraglet is encoded in a bitstream as shown in Figure 3.22. For packing and unpacking the bit-shift is used to get a more compact representation. Hence, the numerical parameter's resolution is limited to the instruction's resolution. This is due to fraglets' data/code intermix nature.

3.3.6 Code Propagation and Deployment

In principle, there are two totally different ways of delivering code to a node: using a third party code dissemination method (shipped with an OS) or a fully implemented propagation layer in fraglets (e.g., "viral" propagation). In this work, we used the former method; the code is spread as loadable modules. Currently, the Fraglets philosophy makes code deployment a non-trivial task. The reason for that is there is no "no-match" primitive (compare to `match`-like instructions from Table D.7). Moreover, there is no easy way to erase the existing code either. That is why code exchange process may require some tricks.

3.3.7 Node-To-Node Communication

Nodes can communicate to each other using `send` instruction¹ (see Table D.8). For unicast operation a node address must be specified as an argument (see Figure 3.28). `ALL` denotes broadcast transmission. *FragletVM* does not support anycast transmissions. Nor it supports `interfaces` between nodes (dedicated communication channels similar to sockets).² The original implementation does.

Forwarding can be easily done with fraglets as shown in Figure 3.29.

¹This is somehow similar to the situation in sequential programming when dynamic memory is allocated and not released which leads to memory leakage.

¹When `send` instruction appears in a fraglet it refers to the rest of the fraglet only, not the entire soup.

²We might introduce them in the future since by having interfaces and using operation polymorphism (see Section 4.3) we could just do the `send usb ...` instead of `print`, `send radio ...` instead of `send N`, `send led ...` instead of `blink`, etc.

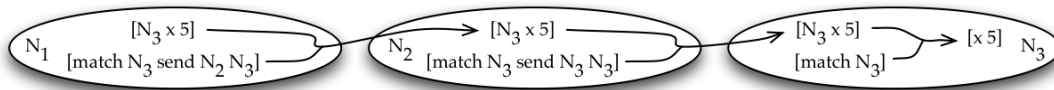


Figure 3.28: FragletVM: Sending Fraglets Between Nodes

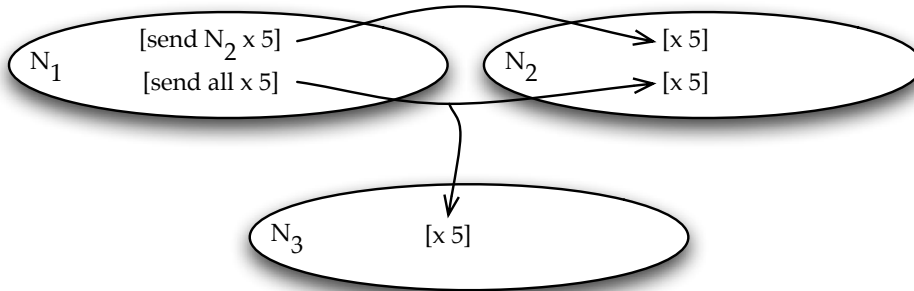


Figure 3.29: FragletVM: Forwarding Fraglets

As with *ChameleonVM* (see Section 3.2) one of the restrictions of the *FragletVM* implementation is that fraglets sent over network cannot exceed a certain size. This is a limitation of the WSN link layer (about 30 bytes). In Section 7.3 we set it as an open question. Locally executed fraglets can be of any size permitted by the memory capacity of a node.

Compared to the node-to-node communication model for *ChameleonVM* from Section 3.2.8, *FragletVM* does not offer data packet exchange. Data must be integrated into fraglets before it can be transmitted. A detailed two nodes communication model can be seen as shown in Figure 3.30. Again, a multi-node communication model is just a scaled version of the picture above.

3.3.8 On-Board Dictionary

The structure of the on-board dictionary in *FragletVM* is identical to the one found in *ChameleonVM*. It was previously discussed in detail in Section 3.2.9.

The main difference is that *FragletVM* employs a distributed architecture where dictionary updates can be initiated by any node in the network. In practice, the architecture is not fully distributed. It rather allows to elect a session leader and assign this role to any node.

3. Configurable VMs for Embedded Networking

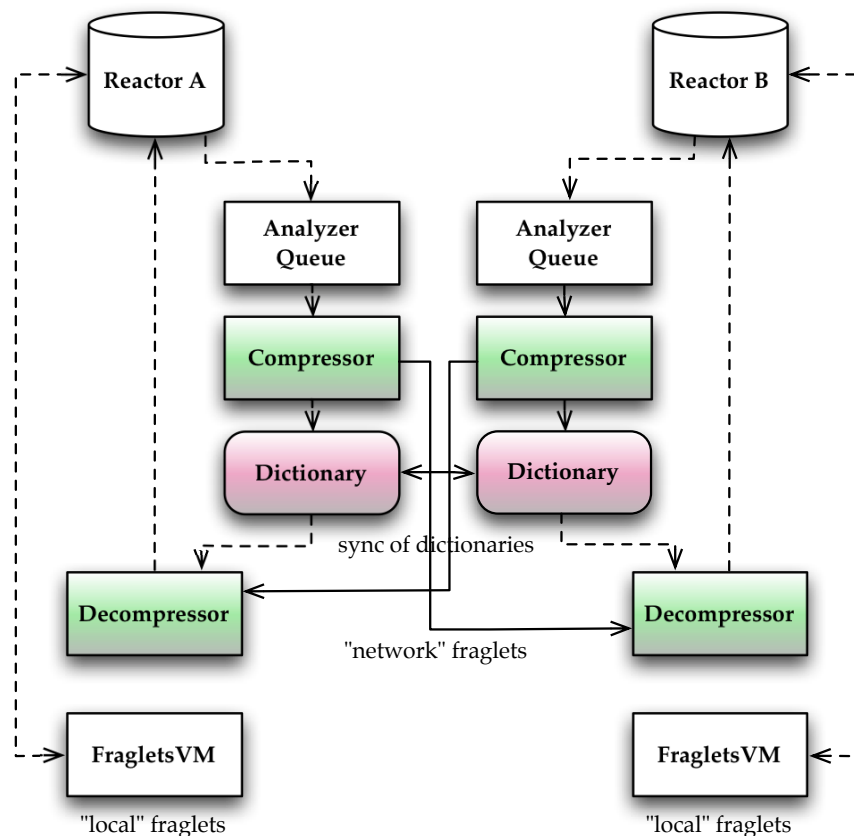


Figure 3.30: FragletVM: Communication Model Between Nodes

This session leader carries out compression and sent out updates to the rest of the network. In *ChameleonVM* the leader role is fixed to the access node. We explain this mechanism in Sections 5.6 and 5.7. A truly distributed compression is further discussed in Section 5.7.

As with *ChameleonVM* each node can have more than one dictionary. But in *FragletVM* dictionaries are created on a per-link basis (between every two communicating nodes). Switching between dictionaries is done automatically depending on which neighbor a node is talking to at the moment.

3.3.9 Dictionary Updates and Synchronization

In each pair of nodes a leader is elected. After that the dictionary synchronization process turns into the group compression model described in Section 5.6. Dictionary updating mechanism is identical to

the one used by *ChameleonVM* and described in Section 3.2.10. The only difference that it runs in a distributed fashion. The process is shown in Figure 3.31.

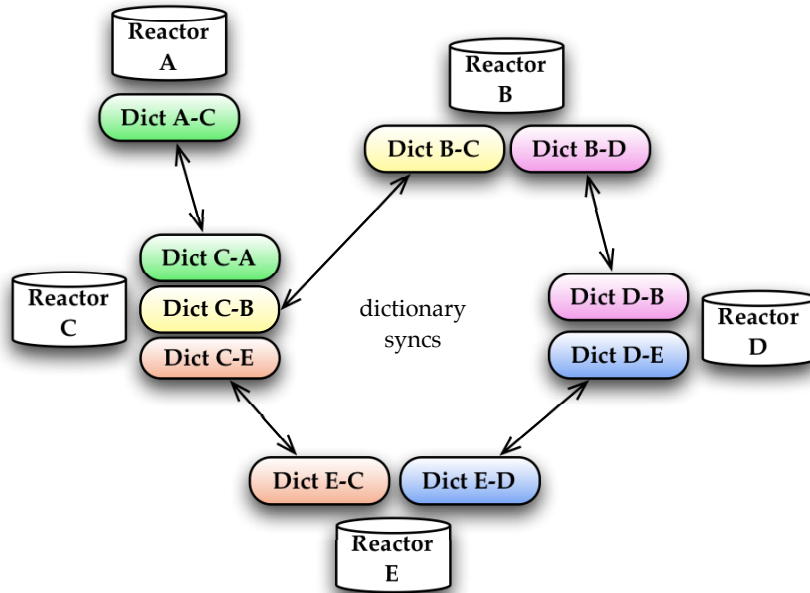


Figure 3.31: FragletVM: Exchange of Dictionary Updates

In addition to automatic dictionary update mechanism *FragletVM* has an extra instruction `newinstr` which allows to create new instructions manually. However, the use of this instruction is limited at the moment. Only instruction with no parameters can be combined together. As a result a new instruction opcode is created in the on-board dictionary. The idea of using the `newinstr` is simple but the transformation function is not trivial as it requires to combine instruction semantics (this is not implemented yet):

$$\begin{aligned}
 & [\text{ssum send } N \ x \ y] \longrightarrow N[(x+y)] \\
 [\text{newinstr ssum send } N \ x \ y] & \longrightarrow [\text{ssum\#send } N \ x \ y] \longrightarrow N[(x+y)]
 \end{aligned}$$

As with *ChameleonVM* a periodical dictionary consistency check is done in order to improve dictionary robustness (see Section 3.2.10). A

3. Configurable VMs for Embedded Networking

simple CRC-based integrity function φ is used for this purpose. This is shown in Figure 3.31.

3.3.10 Compression and Decompression

The (de-)compression engine in *ChameleonVM* and *FragletVM* is the same. Hence, (de-)compression principles remain the same too (see Section 3.2.11). Compression is performed on outgoing fraglets only; locally executed code is kept untouched.

3.4 Implementation Remarks

In Sections 3.2 and 3.3 we have presented two execution environments for network embedded systems. Besides having these frameworks simulated we also partially prototyped both systems on *TelosB* hardware platform with *ContikiOS* support. Initially *ChameleonVM* was also prototyped under *TinyOS*. Later we switched to *ContikiOS* because of its modular, flexible structure, support for loadable modules and plain C programming environment. We also made an attempt to implement *FragletVM* on *Sentilla Perk* with *Java ME CLDC 1.1*. The last platform allows easy Java programming and fast prototyping as all needed system level protocols (e.g., MAC) are automatically provided. Although we did not implement the entire functionality “in hardware” two prototypes have shown that the proposed designs can be fit in resource-limited devices like WSN.

3.5 Summary

To this point we have analyzed the existing work in the area of network morphing for WSN. We have proposed our own extended and generalized model of WSN re-tasking. The model is based on using spacial and time separation of tasks (profiles) on different network VS. After that we have presented two example implementations for two different network models, classical active network with mobile agents and chemical reaction network. Both examples are based on using a small embedded re-configurable VM. In our designs we use properties of the existing systems but at the same time add a number of unique features, especially in the area of mobile code run-time processing. We have described in detail main architectural aspects and building blocks

of these two systems giving when it was necessary a comparison to the existing solutions.

4

Overview of Code Optimization Techniques

In Chapter 1 we briefly discussed why the focus of this work was put on code optimization techniques. So far, in the literature mainly compile-time code optimization have been researched. We believe that by “tweaking” code in run-time (i.e., “online”) further improvements in performance and energy consumption can be archived. Although every improvement comes with a certain overhead; hence, a trade-off must be found. In this chapter, we discuss the following optimization steps in details: code shrinking (code size reduction), compression, polymorphism, versioning and robustness. The last two methods are not actually a part of the optimization process, rather they play a complementary roles of controlling code distribution and execution, and making code more reliable and resilient to corruption.

4.1 Code Shrinking

The first optimization technique we going to explore is **code shrinking**. Code shrinking basically assumes getting rid of those parts of the program which are not needed anymore. This process takes place continuously while executing a program. The removed parts are not necessarily removed from the system (they can be kept for future use), rather from the execution process. We will demonstrate how code shrinking works on two variations of one simple examples for *ChameleonVM* which is discussed in Section 3.2.¹

For this example we take a program which does the following:

- turn on LED² number 1 on node 1,
- turn on LED number 2 on node 2,
- turn on LED number 3 on node 3, and
- turn on all LEDs on all other nodes.

This algorithm demonstrates a very common programming language concept called **branching**. We assume unique nodes numbering. The aim we are pursuing is the following: as soon as the original program reaches node 1, 2 or 3, we erase the corresponding chunk of code and propagate it further, as those nodes are unique and it is not needed to address them anymore. On the other hand, the code section targeted at the rest of the nodes must be kept till the entire set is processed.

The first problem we face is structural. The efficiency of the method strictly depends on how the original code was designed. Lets consider two variations. The first variation belongs to the class of *case*-branching (see Listing 4.1).³

```
1 ADDR || CODE
2 =====
3 0003 ||      send ME,ALL      # broadcast itself
```

¹Code compression is used by *ChameleonVM* and *FragletVM*. Additionally, *ChameleonVM* also uses other optimization steps described in Chapter 4.

²LED (*Light Emission Diode*) – indicator scheme available on most WSN platforms, normally used for debugging purposes.

³For simplification, the presented code assumes the von Neumann principle of intermixed code and data. In reality, *ChameleonVM* uses the Harvard architecture (see Section 3.2.2).

```

4 0005 ||      push NID          # put node id on the stack
5 0008 ||      jmpeq 1,L1
6 0011 ||      jmpeq 2,L2
7 0014 ||      jmpeq 3,L3
8 0016 ||      led 0x7          # nodes!=1,2,3: all leds
9 0017 ||      exit
10 0018 ||     ---            # erase marker
11 0020 || L1: led 0x1        # node == 1: led #1
12 0021 ||      exit          # stop execution
13 0023 ||      erase UP      # erase up to the nearest marker
14 0024 ||     ---
15 0026 || L2: led 0x2        # node == 2: led #2
16 0027 ||      exit
17 0029 ||      erase UP
18 0030 ||     ---
19 0032 || L3: led 0x4        # node == 3: led #3
20 0033 ||      exit
21 0035 ||      erase UP

```

Listing 4.1: Example of “Case”-Branching

The second variation refers to the class of *if*-branching (see Listing 4.2). Compared to the previous case this one is characterized by a bigger overhead the original code gets.

```

1 ADDR || CODE
2 =====
3 0003 ||      send ME,ALL      # broadcast itself
4 0005 ||      push NID          # put node id on the stack
5 0006 ||      ---
6 0009 ||      jmpeq 1,L1
7 0011 ||      jmp L4
8 0013 || L1: led 0x1
9 0014 ||      exit
10 0016 ||      erase UP
11 0017 ||      ---
12 0020 || L4: jmpeq 2,L2
13 0022 ||      jmp L5
14 0024 || L2: led 0x2
15 0025 ||      exit
16 0027 ||      erase UP
17 0028 ||      ---
18 0031 || L5: jmpeq 3,L3
19 0033 ||      jmp L6
20 0035 || L3: led 0x4
21 0036 ||      exit

```

4. Overview of Code Optimization Techniques

```
22 0038 ||      erase UP
23 0040 || L6: led 0x7
```

Listing 4.2: Example of “If”-Branching

In order to make the difference between the above examples clear lets look at the algorithmic representation. From Figure 4.1 it is becomes obvious that these two cases are not isomorphic.

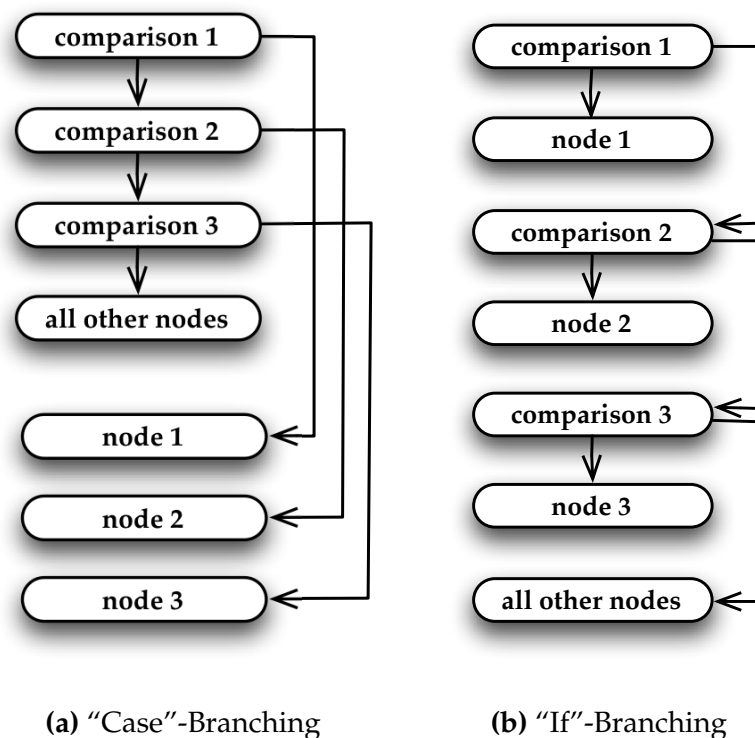


Figure 4.1: “Case”-Branching vs “If”-Branching

Now lets have a look at how code shrinking works. Operator `---` stands for an erase marker which, if it is referred later by `erase UP` operator initiates deleting of the code in between those two. Operator `erase` can take one of the following arguments:

What the impact does it have to introduce that overhead in your code? It highly depends on the context, i.e., the current configuration and the propagation process (the sequence of nodes a capsule passes

Argument	Meaning
UP	up to the nearest --- operator
DOWN	down to the nearest --- operator
TOP	up to the beginning of the capsule (or the current segment)
BOTTOM	down to the end of the capsule (or the current segment)

through). For the *case*-version and various propagation scenarios we will have:

Propagation Scenario	Code Reduction (symbols)
1 → 2 → 3 → ...	35 → 29 → 23 → 17 → the same

The sequence in which nodes 1, 2 and 3 are passed by is not important for this example (e.g., 1 → 2 → 3 → to 3 → 1 → 2 →) because each of those steps initiates removal of 6 bytes. In case of different processing on each node it will matter. Passing by a node being not from the set (1, 2, 3) does not change the code as soon as we keep the corresponding slice for later. Similarly, for the *if*-version we have:

Propagation Scenario	Code Reduction (symbols)
1 → 2 → 3 → ...	41 → 30 → 19 → 8 → the same

Again, the order of processing does not matter as the size of each chunk is the same.

This comparison shows one of the fundamental features of code shrinking: the more overhead the original code has the better shrinking rates can be archived.

4.2 Code Compression

The next technique we are going to give a try is code compression. In Chapter 5 we discuss in detail the scheme we propose called **online code compression**. Here, we will briefly discuss our view of the compression world, various methods used to compress the code, what they feature and lack, and what eventually motivated us to develop a new method.

4. Overview of Code Optimization Techniques

Data compression (or **source coding**) is the process of encoding information using fewer information symbols (particularly bits) than an unencoded representation would use, through use of specific encoding methods. The basic goal of every data compression scheme is to get rid of redundancy in an original data stream, and keep it still decodable. Compression is traditionally used in two areas: communications (to improve channel utilization) and data storage (to improve storage media utilization). Both, transmitter and receiver (writer and reader), must be familiar with the encoding scheme being used in order to understand each other. One of the biggest challenges with designing an appropriate compression scheme is to make it meet the resource- and time-constraints of the system. That is, an encoding/decoding engine should fit into system's resource space (e.g., in case of limited memory) and still allow the main system's task(s) to be executed within a certain time range (time-critical applications). For instance, a compression scheme for VoIP/video may require extra memory and processing cycles for the sound/video stream to be decompressed fast enough to be heard/viewed as it is being decoded (full decoding before use might be unacceptable because it requires much time and more storage space than the system can normally provide). The design of data compression schemes, therefore, involves trade-offs among many factors, including the degree of compression, the amount of distortion introduced (if using a compression scheme with losses), and the computational resources required to compress and uncompress the data.

Compression methods can be divided into two main groups: **lossy** and **lossless** compression methods. Lossy data compression is used if some loss of fidelity is acceptable. Generally, this method allows some data to be removed from an original source without losing the sense of what this data represents. This means that input of an encoder and output of a decoder are never the same. Lossy schemes accept some loss of data in order to achieve higher compression, thus it is mainly used in image- (e.g., JPEG), video- (e.g., MPEG) and sound- (e.g., VoIP) related applications which do require only a certain level of precision distinguishable by human's receptors. Lossless compression schemes are reversible so that the original data can be reconstructed. Lossless compression algorithms usually exploit statistical redundancy in such a way as to represent the sender's data more concisely without error. Lossless compression is possible because most real-world data has

4. Overview of Code Optimization Techniques

$$\text{compression factor (CF)} = \frac{\text{compressed code size}}{\text{original code size}} * 100\% \quad (4.1)$$

In the example above it is $5/50 \approx 10\%$ for lossless compression, $5/50 \approx 10\%$ for lossy compression (the same as lossless but precision is lost), and $1/52 \approx 2\%$ for our hypothetical technique scheme.

In a similar way, the **space saving** can be defined as follows:

$$\text{space saving (CF)} = \left(1 - \frac{\text{compressed code size}}{\text{original code size}}\right) * 100\% \quad (4.2)$$

Here we come to a very important point of our discussion: data and code data sources are naturally different. Code stream has much higher entropy than data, thus much less statistical redundancy mentioned above (which is needed for lossless compression schemes to be efficient). There are two main reasons for that: 1) code streams have less patterns, and 2) code streams are normally finite and relatively short. Most solutions proposed so far have focused on applying data compression techniques to the code world, and in our opinion they all have failed to some extent because of ignoring the different nature of the code. The embedded world makes it even more obvious. Although some relatively good CF may be archived, a compressing/decompressing engine cannot be normally fitted in the system profile because it requires too many computational resources. Let us take an example from [TDV08a], where applying various compression algorithms (including *gzip*) to pre-compiled ELF software modules a CF of approximately 50% can be archived. This comes with a memory footprint of 7.7 kB out of 10 kB totally available in the system and decompression time on the order of several seconds depending on the module's size.

From now on we will focus on lossless compression methods only. Our choice is dictated by the fact that if some pieces of code are removed from an original stream, therefore, decoder will not be able to reproduce it later, then the code needs to have some kind of "self-healing" feature to restore the missing parts. This might be a good research question for systems like Fraglets which can potentially heal themselves. Some insights can be taken from the works on resilience to knock-out attacks and software self-healing properties [MYT08b]. The

lossless compression methods can be grouped into several main categories we discuss in the upcoming sections: very basic encoding forms (Section 4.2.1), *transform-based* coding (Section 4.2.2), *prediction-based* coding (Section 4.2.3), *dictionary-based* coding (Section 4.2.4) and *probability (entropy)* coding (Section 4.2.5). We will also pay some attention to special coding schemes such as building coders with infinite inputs and coders for the distributed environment.¹

4.2.1 Primitive Methods

4.2.1.1 Run-Length Encoding

Run-Length Encoding (RLE) encodes repetitive symbolic or binary sequences as a single symbol followed by a number of repetitions. The example of this scheme was shown in Section 4.2. This method is very effective if there are many such sequences in the incoming stream. Decoding is very simple as it requires stream expansion only. As we show in Chapter 5 this method cannot cope well with code streams having naturally high entropy on which it could greatly increase the original stream size. Sometimes it gets combined with *Huffman* coding (see Section 4.2.5.1) to improve the CF.

4.2.1.2 Delta Encoding

Delta encoding (or *delta differencing*) is a method based on using relative entropy.² The resulting streams contains not the actual data but differences between sequential data. *Delta* encoding can greatly reduce redundancy if the stream evolves gradually, i.e., differences are small.

The best known example of *delta* encoding is *VCDIFF* format. Also can be used in HTTP [32202]. Sometimes the results get better if *delta* encoding is followed by *RLE* (see Section 4.2.1.1).

¹The descriptions in Sections 4.2.1–4.2.6 discuss the existing methods at the time of writing; they partially cite and are based on [Ble10] and the Wikipedia articles rooted at http://en.wikipedia.org/wiki/Data_compression (these pages contain proper referencing to the original sources). Copyrights are held by the corresponding authors.

²The difference between two data values is the information required to obtain one value from the other.

4. Overview of Code Optimization Techniques

4.2.2 Transform-based Encoding

4.2.2.1 Move-To-Front Transform

The *Move-To-Front (MTF)* transform [Rya80] is an improvement over entropy encoding discussed in Section 4.2.5. The main idea is that each symbol in the stream is replaced by its index in the stack of “recently used symbols”. For example, long sequences of identical symbols are replaced with as many zeroes, whereas when a symbol that has not been used recently appears, it is replaced with a large code. This is achieved by moving an encoded value to the front of the list before continuing to the next value. At the end the stream is transformed into a sequence of codes; if the data exhibits a lot of local correlations of frequencies, then these integers tend to be small. However, not all data exhibits this type of local correlation, and for some streams, the *MTF* transform may actually increase the entropy.

Normally, *MTF* is used in cooperation with *Burrows-Wheeler Transform (BWT)* discussed in Section 4.2.2.2. The *BWT* is very good at producing a sequence that exhibits local frequency correlation from certain classes of data, mainly text. However, it does not decrease the entropy, it does reordering only. Compression becomes more effective if *BWT* is followed by *MTF* before the final *entropy-based* encoding step (see Section 4.2.5).

4.2.2.2 Burrows-Wheeler Transform

Burrows-Wheeler Transform (BWT) [BW94] permutes the order of the symbols in the stream, it does not change their values. If the original stream has several patterns that occur often, then the transformed stream will have several places where a single symbol is repeated multiple times in a row. This is useful for further *entropy-based* compression step using techniques such as *MTF* (see Section 4.2.2.1) or *RLE* (see Section 4.2.1.1).

There are two important aspects of *BWT* to mention: 1) it is a reversible sort, allowing the original stream to be re-generated, and 2) it expands its input slightly.

4.2.3 Prediction-based Encoding

Prediction by Partial Matching: *Prediction by Partial Matching (PPM)* [CW84] is an adaptive statistical data compression technique based on

context modeling and prediction. *PPM* models use a set of previous symbols in the uncompressed symbol stream to predict the next symbol in the stream. Predictions are usually reduced to symbol rankings. *PPM* is normally followed by *Dynamic Markov Compression (DMC)*. The actual symbol selection is usually made using *arithmetic* coding (see Section 4.2.5.2), though it is also possible to use *Huffman* encoding (see Section 4.2.5.1) or even some type of *dictionary* coding technique (see Section 4.2.4). Markov modeling can be replaced as well. Multiple symbols prediction is possible. Normally, *PPM*-oriented algorithms require a significant amount of RAM. *PPM* is used by many popular compression formats such as *ZIP*, *RAR*, *7z*. The best performance metrics have been shown for the natural language text.

Context mixing: *Context mixing* [Mah05] is a type of compression in which the next-symbol predictions of two or more statistical models are combined to yield a prediction that is often more accurate than any of the individual predictions. Some methods average the probabilities assigned by each model, the others outputs the prediction that is the **mode**¹ of the predictions output by individual models. The *PAQ* series of data compression programs use context mixing to assign probabilities to individual bits of the input. This model is used in the *bzip2* compression format too.

Dynamic Markov Compression: *Dynamic Markov Compression (DMC)* [CH87] algorithm uses *predictive arithmetic* coding similar to *PPM*, except that the input is predicted one bit at a time (rather than one byte at a time). *DMC* has also a lot of similarities with context mixing algorithms except that *DMC* uses only one context per prediction. The predicted bit is then coded using *arithmetic* coding (see Section 4.2.5.2). *DMC* has a good compression ratio and moderate speed, similar to *PPM*, but requires more RAM than *PPM*.

4.2.4 Dictionary Encoders

A *dictionary* coder operates by searching for matches between the stream to be compressed and a set of strings contained in a data structure called the **dictionary** and maintained by the encoder. When the encoder finds such a match, it substitutes a reference to the string's

¹In statistics, the **mode** is the value that occurs most frequently in a data set or a probability distribution.

4. Overview of Code Optimization Techniques

position in the data structure. The resulting dictionary has to be carried along with the encoded data.

Some coders use a static dictionary. In this case, a full set of strings is determined before coding begins and does not change during the coding process. This approach is most often used when the message or set of messages to be encoded is fixed and large. More common are methods where the dictionary starts in some predetermined state but the contents change during the encoding process, based on the data that has already been encoded. Both the *LZ77* and *LZ78* algorithms discussed below work on this principle.

Lempel-Ziv-1 (LZ77) and Lempel-Ziv-2 (LZ78): In *LZ77*, a data structure called the **sliding window** is used to hold the last N symbols of data processed; this window serves as the dictionary, effectively storing every substring that has appeared in the past N symbols as dictionary entries. Instead of a single index identifying a dictionary entry, two values are needed: the length, indicating the length of the matched sequence, and the offset (also called the distance), indicating that the match is found in the sliding window starting offset symbols before the current stream.

In contrast, *LZ78* uses a more explicit dictionary structure; at the beginning of the encoding process, the dictionary only needs to contain entries for the symbols of the alphabet used in the stream to be compressed, but the indexes are numbered so as to leave spaces for many more entries. At each step of the encoding process, the longest entry in the dictionary that matches the sequence is found, and its index is written to the output; the combination of that entry and the symbol that followed it in the stream is then added to the dictionary as a new entry.

The most popular *LZ77*-based compression method is *DEFLATE* [19596]. It combines *LZ77* with *Huffman* coding. The *LZ78* was populated by the *LZW* algorithm.

Lempel-Ziv-Welch (LZW): *LZW* [ZL77] is an improved version of *LZ78* algorithm with various refinements for different applications. That is why it is important that the encoder and decoder agree on which type of *LZW* is being used: the size of the alphabet, the maximum code width, whether variable-width encoding is being used, the initial code size, whether to use the clear and stop codes (and what values they

have). Most formats that employ *LZW* embed this information into the format specification or provide explicit fields for them in a compression header for the data.

The algorithm works best on streams with repeated patterns, so the initial parts of the stream will see little compression; longer input streams are typically necessary before the compression builds up efficiency. As the message grows, however, the compression ratio tends asymptotically to the maximum.

Statistical LZW: *Statistical LZW* [KH01] may be viewed as a variant of the general *Lempel-Ziv (LZ)* method. The contribution of this concept is to include the statistical properties of the source information to identify the useful data that should be put into the dictionary (search window) while most of the *LZ*-based compression methods, such as *LZ78* and *LZW* do not take this property into consideration. *Statistical LZ* improves the compression ratio compared with *LZ77* because more useful data can be kept in the dictionary. The dictionary can be smaller in size for keeping the useful data and hence less memory required in the decompressor. Since not all the data has to be shifted into the window, less processing power is required on the decompressor.

Many derivatives from *LZ77*, *LZ78* and *LZW* exist [Sal97] which are meant to improve the CF: *LZMW*, *LZAP*, *LZWL*, *LZMA*, *LZSS*, *LZJB*, and others.

Byte-Pair Encoding (BPE): *Byte-Pair Encoding* [Gag94] is another simple *dictionary-based* coding scheme, where a symbol that does not appear in the source stream is assigned to represent the most commonly appearing consecutive two-symbol combination. This can be done repeatedly as long as there are symbols that do not appear in the source stream, and symbols that are already representing combinations of other streams can themselves appear in combinations. A table of the replacements is required to rebuild the original stream.

Our *online compression* scheme discussed in Chapter 5 borrows the main principle from *BPE* because of its simplicity. One significant advantage of the both algorithms is that compression never increases the stream size. In contrast, *LZW* can greatly inflate the size of certain data sets, such as randomized data or pre-compressed fields. *LZW* compression adapts linearly to frequently occurring patterns, building up strings one character at a time. The *BPE* and *online compression*

4. Overview of Code Optimization Techniques

algorithms adapt exponentially to patterns, since both symbol in a pair can represent previously defined pair codes. The previously defined pair codes can themselves contain nested codes and can expand into long strings.

4.2.5 Entropy Encoding

Increased CF can often be achieved with an *adaptive entropy* encoder. Such a coder estimates the probability distribution for the value of the next symbol, based on the observed frequencies of values so far. A standard entropy encoding such as *Huffman* coding or *arithmetic* coding then uses shorter codes for values with higher probabilities.

4.2.5.1 Huffman Coding

The method [Huf52] uses a *variable-length* code (a.k.a., *prefix* code) table for encoding a source symbol. The *variable-length* code table is derived in a particular way based on the estimated probability of occurrence for each possible value of the source symbol. Although *Huffman's* original algorithm is optimal for a symbol-by-symbol coding (i.e., a stream of unrelated symbols) with a known input probability distribution, it is not optimal when the symbol-by-symbol restriction is dropped, or when the probability mass functions are unknown, not identically distributed, or not independent. Other methods such as *arithmetic* coding and *LZW* coding often have better compression capability: both of these methods can combine an arbitrary number of symbols for more efficient coding, and generally adapt to the actual input statistics, the latter of which is useful when input probabilities are not precisely known or vary significantly within the stream. However, *Huffman*-encoding can be used adaptively, accommodating unknown, changing, or context-dependent probabilities. In the case of known independent and identically-distributed random variables, combining symbols together reduces inefficiency in a way that approaches optimality as the number of symbols combined increases. *Prefix* codes including *Huffman* tend to have inefficiency on small alphabets.

Many variations of *Huffman* coding exist including *adaptive Huffman* encoding and *Shannon-Fano* coding.

Adaptive Huffman coding [Vit87] (a.k.a *Dynamic Huffman* coding) allows building the code as the symbols are being transmitted, having no initial knowledge of source distribution, that allows one-pass

encoding and adaptation to changing conditions in data. The benefit of one-pass procedure is that the source can be encoded in real time, though it becomes more sensitive to transmission errors, since just a single loss ruins the whole code.

Shannon-Fano coding [Sha48] [Fan49] constructs a *prefix* code based on a set of symbols and their probabilities (estimated or measured). It is suboptimal in the sense that it does not achieve the lowest possible expected code word length like *Huffman*-coding; however, unlike *Huffman*-coding, it does guarantee that all code word lengths are within one bit of their theoretical ideal. The algorithm works, it produces fairly efficient but not always optimal *prefix* codes. In contrast, *Huffman*-coding is almost as computationally simple and produces *prefix* codes that always achieve the lowest expected code word length. That is why *Shannon-Fano* coding is rarely used in practice.

4.2.5.2 Arithmetic Coding

It is a form of *variable-length entropy* encoding [Ris76] [RL79] [WNC87]: frequently used characters are stored with fewer bits and not-so-frequently occurring characters are stored with more bits, resulting in fewer bits used in total. *Arithmetic* coding differs from other forms of *entropy* encoding such as *Huffman* coding in that rather than separating the input into component symbols and replacing each with a code, *arithmetic* coding encodes the entire message into a single number, a fraction n where $0 \leq n < 1$.

In general, *arithmetic* coders can produce near-optimal output for any given set of symbols and probabilities. Compression algorithms that use *arithmetic* coding start by determining a model of the data – basically a prediction of what patterns will be found in the symbols of the message. The more accurate this prediction is, the closer to optimality the output will be. *Arithmetic* coding has higher computational complexity than *Huffman* or *Shannon-Fano* but can produce greater overall compression.

Range encoding [NM79] is sometimes considered as a different form of *arithmetic* coding. When processing is applied as one step per symbol, it is *range* coding, and when one step is required per every bit it is *arithmetic* coding. *Range* encoding conceptually encodes all the symbols of the stream into one number, unlike *Huffman* coding which

4. Overview of Code Optimization Techniques

assigns each symbol a bit-pattern and concatenates all the bit-patterns together.

4.2.5.3 Static Codes

In order to be able to use this type of coding the approximate entropy characteristics of a data stream must be known in advance. Moreover, these models are normally very general and assume infinite input streams. They can be divided into two main sub-groups: *universal* and *non-universal* codes.

Non-Universal Codes: *Non-Universal* codes are specifically designed for the streams with certain properties (distributions).

Golomb coding [Gol66] require alphabets to follow a *geometric* distribution to achieve an optimal *prefix*-like *Golomb* code. This makes *Golomb* coding highly suitable for situations in which the occurrence of small values in the input stream is significantly more likely than large values.

Rice coding [RP71] denotes using a subset of the family of *Golomb* codes to produce a simpler (but possibly suboptimal) *prefix* code. It is especially suitable for use with binary arithmetics.

Unary coding is an *entropy* encoding that represents a natural number, n , with n ones followed by a zero (if natural number is understood as non-negative integer) or with $n - 1$ ones followed by a zero (if natural number is understood as strictly positive integer).

Universal Codes: Universal codes for integers are *prefix* codes that map the positive integers onto binary codewords, with the additional property that whatever the true probability distribution on integers, as long as the distribution is monotonic (i.e., $p(i) \geq p(i + 1)$ for all positive i), the expected lengths of the codewords are within a constant factor of the expected lengths that the optimal code for that probability distribution would have assigned. *Universal* codes are generally not used for precisely known probability distributions, and no *universal* code is known to be optimal for any distribution used in practice.

These are some *universal* codes for integers [Mac03]: *Elias* codes, *Exp-Golomb* coding, *Fibonacci* coding, *Levenstein* coding.

Huffman coding and *arithmetic* encoding when they can be used give at least as good, and often better compression than any *universal* code. However, *universal* codes are useful when *Huffman* coding cannot be

used – for example, when the exact probability of each message are no known, but only the rankings of their probabilities. *Universal* codes are also useful when *Huffman* codes are inconvenient. For example, when the transmitter but not the receiver knows the probabilities of the messages, *Huffman* coding requires an overhead of transmitting those probabilities to the receiver. Using a *universal* code does not have that overhead.

4.2.6 Distributed Source Coding

Distributed Source Coding (DSC) (a.k.a., *Slepian-Wolf* coding [SW73]) describes the compression of multiple correlated information sources that do not communicate with each other. By modeling the correlation between multiple sources at the decoder side together with channel codes, *DSC* is able to shift the computational complexity from encoder side to decoder side, therefore, provide appropriate frameworks for applications with complexity-constrained sender, such as WSN [XLC04] and video/multimedia compression. One of the main properties of *DSC* is that the computational burden in encoders is shifted to the joint decoder. Two sub-models exist: *asymmetric DSC* and *symmetric DSC*. *Asymmetric DSC* means that, different bitrates are used in coding the input sources, while same bitrate is used in *symmetric DSC*. Although most research is focused on *DSC* with two dependent sources, the more than two input sources cases have been studied as well [XK08].

4.2.7 Compression in Embedded Systems

As it is stated in [Phi05], compression/decompression may require a large amount of CPU processing and can be especially difficult on microcontrollers with small word sizes and limited memory. However, the savings achieved in data propagation are normally not over-shadowed by the CPU cycles for decompression in an ES such as WSN since communication is the biggest energy spending part on such platforms. So far, the focus has been put on applying data compression methods to pre-compiled code modules. With no surprise, most of the solutions described above show bad figures, especially in embedded applications [TDV08a].

Use of a high level language which is interpreted by a VM is another approach to reduce the amount of data that is transmitted. The language's instructions "resemble common unit operations that are

4. Overview of Code Optimization Techniques

carried out by a typical sensor network application, such as ‘sense the temperature and transmit it to the base station’. Thus updating the application merely requires transmitting a new script to the network. This approach does not support updating the VM or the underlying OS, however.” Most of the solutions mentioned above in Section 3.1 somehow follow this idea by introducing new instruction sets which are better suited for a particular application group. *ASVM* [LGC05], build on top of *Maté* VM [LC02], goes a bit further and allows to configure the instruction set at compile-time to better fit it into application’s context. Our belief is that in order to build a truly flexible (adaptable) system, the system itself must be able to change its instruction set at run-time, that is the system must possess some features of autonomic computing. By doing so, the system must be able to find its own way to an optimal instruction set in the particular context, at some particular point in time.

Ideally, our hypothetical system must be able to learn communication patterns from scratch, by analyzing incoming active packets of different formats and building a new compression rules according to the current code landscape (**code adaptation process**). In this work, we simplified our task and started from a higher level, i.e., a level of VM instructions (we do not research here the translation step from executable (native, machine) to interpretable code – see Figure 4.2). Having a relatively short and transparent instruction set makes compression an easier task and eliminates all the problems with compatibility.

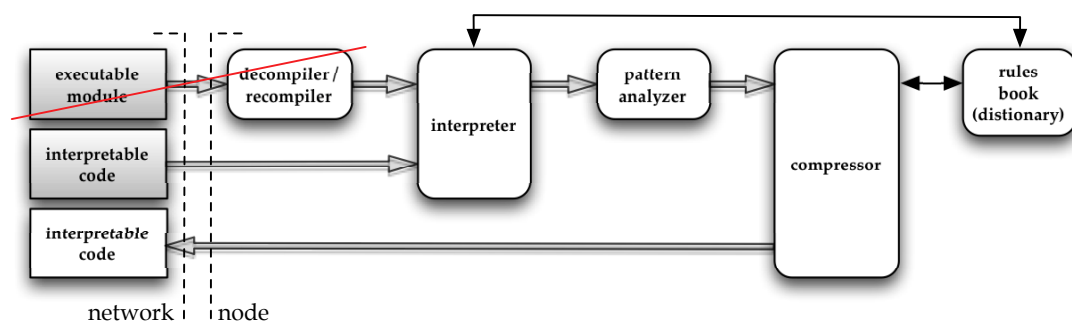


Figure 4.2: Code Adaptation Process

In Chapter 5 we propose an *online compression* scheme and analyze various scenario of its use. The basic principle of our solution is that there is no compression in the first place. Code floods the network, is installed on nodes and put into operation. Constantly, while being executed, this code is put through a compressor chain which does not take an effect on execution flow straight away. Rather, over time, the network nodes, by analyzing current execution patterns, pick a potentially optimal encoding and apply it without interrupting the running software. As soon as the new software pieces are injected and the old ones are replaced the system gradually adapts to those changes and modifies coding on the fly. This allows to archive nearly optimal coding (compressed representation) for the program. As it has been said, program execution is not interrupted. The implementation is simple and does not require much computations or memory usage, thus it can be easily implemented in most types of ES which exist nowadays. In some situations communication between nodes is needed in order to agree on the next steps in compressing process to be made; our method was designed to minimize communication activities and thus prolong the nodes' lifetime. In Chapter 5 we also analyze various scenario of using this scheme: local, group, cloud and distributed compression variants.

4.3 Code Polymorphism

The next code reduction technique we will consider refers to the concept of **code polymorphism**.¹ Polymorphism can be of different nature. **Code polymorphism** is the ability to create a variable, a function, or an object that has more than one form with the same name. When and which form is put into operation by calling it by the common name depends on the context. **Type polymorphism** allows program code to work with various types. We will distinguish between two types of polymorphism here: **space-** and **time-** types of code polymorphism.

Lets consider the following example which essentially just takes a measurement, puts it in a buffer (stack), then after having 10 measure-

¹Do not confuse with “**polymorphic code** which is code that uses a polymorphic engine to mutate while keeping the original algorithm intact. That is, the code changes itself each time it runs, but the function of the code (its semantics) does not change at all.”

4. Overview of Code Optimization Techniques

ments in the buffer it calculates a mean. The code doing this is shown in Listing 4.3 below.

```
1  sense          # sense the very first value
2 L1: sense      # sense some value
3  add           # do "add" with two top values on the stack
4  add 2,1       # increment counter by 1 (position 2)
5  jmpneq 2,10,L1 # keep doing so until 10 measurements have
6                # been taken
7  div           # do "div" with two top values on the stack
```

Listing 4.3: Code Space-Polymorphism Example

Let us illustrate it with the following picture which shows what is happening on the stack (see Figure 4.3).

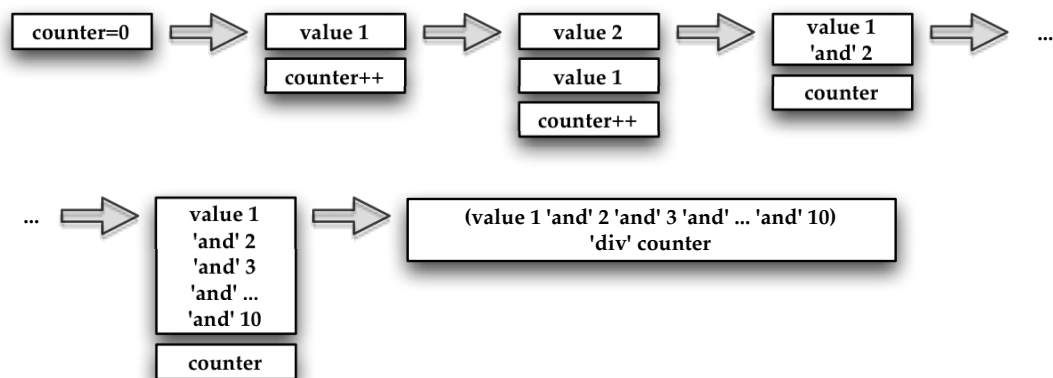


Figure 4.3: Stack Operations for the “Sense-Calculate-Mean” Example

Obviously, what this piece of code does highly depends on the semantics of `add` and `div` operators. Lets consider two variations:

1. `add` means “take two top elements from the stack, do arithmetic addition on them, and put the result back on top of the stack” and `div` means “take the top value from the stack, do arithmetic division of that value by the next value on the stack (position ‘top-1’), and put the result back on top of the stack”.¹

¹To keep it simple, we do not take into account possible overflow or division by 0 exceptions.

2. **add** means “take two top elements from the stack, do arithmetic addition on their inverses, and put the result back on top of the stack” and **div** means “take the second top value from the stack (position ‘top-1’), do arithmetic division of that value by the first top value on the stack (position ‘top’), and put the result back on top of the stack”.

So, what would be the result of both cases? Essentially, the first calculates an *arithmetic mean*:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i = \frac{1}{n} (x_1 + \dots + x_n)$$

and the second calculates a *harmonic mean*:

$$\bar{x} = \frac{n}{\sum_{i=1}^n \frac{1}{x_i}} = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \dots + \frac{1}{x_n}}$$

This eventually means that by switching the internal semantics of only two operators, **add** and **div**, we can change the behavior of the code. Semantic reassignments are made through a dictionary how it has been explained in Section 3.2.9.

This example demonstrates that having the same code on different nodes, we can make them implement different algorithm by changing their on-board dictionaries. We call it **space-polymorphism**. By applying the same principle for the time domain we can obtain what we call **time-polymorphism**: each node can behave differently over time by using different semantics for the same code. This is archived by switching between dictionaries – this mechanism was also explained in Section 3.2.9.

In the example above operators **add** and **add 2 1** are actually the same operator taking a different number of arguments. This allows to encode them identically. The number of arguments and their values are processed at run-time. This mechanism was explained in Section 3.2.9.

4.4 Code Versioning and Lifecycle

Code versioning allows to keep software on all nodes up-to-date and to control code installation and removal. Each capsule in the system has unique ID and version number. The unique ID allows to identify

4. Overview of Code Optimization Techniques

capsule's type, the version number – its age, or generation. Additionally, for every capsule a lifetime can be specified – as an absolute value (time units, number of packets/capsules processed). All that is done with the following directives:

```
1   Autoupdate 1      # 0 - autoupdate disabled
2                               # 1 - autoupdate enabled
3   Lifetime 10s     # lifetime (0 - stay alive forever)
4   Id 0x41          # capsule's ID/version
```

Listing 4.4: Capsule's Control Directives

AUTOUPDATE enables (0) or disables (1) autoupdate feature. If autoupdate is enabled then a capsule with the newer version and the same ID replaces the older one automatically (it is installed and put into execution by the VM; no further actions from the capsule itself are required). If two capsules have the same ID – no replacement is made automatically. If autoupdate is disabled capsule's replacement is still possible but this must be encoded in the incoming (new) capsule with **replace** operator which takes the replacing capsule's ID as an implicit argument.

If multiple capsules are used to implement some task, only capsules with the same version can react. This avoids version inconsistency. In case if only one capsule in a set is updated and others remain the same, a **bind** operator exists which, if it is called by some capsule on a node, updates all other capsules to its version number. This operation can be potentially dangerous and must be used with care.

LIFETIME defines the **lifecycle** of a capsule. It is specified as an absolute value (currently milliseconds, seconds and minutes are supported: *50 ms*, *5 s*, *1 m*; plus a number of packets/capsules processed: *10p*). After the time (or the threshold) is expired (exceeded) the capsule is removed from the node automatically. If lifetime is set to 0, the capsule will stay alive forever, unless it is erased using **die** operator. **die** operator can be called from within this capsule only. To call the same functionality from an other capsule installed on a node use **kill** operator; it takes removing capsule's ID as an argument. Its use is not appreciated though.

Finally, **ID** field specifies capsule's ID and version number. In the current implementation of *ChameleonVM*, for example, it is an 8-bit value.¹ The most bits specify capsule's ID, the least – version number.

Having all those parameters in place allows to finely control capsule's lifecycle. Lifetime determines for how long the capsule will exist in the system; autoupdate, ID and version number guarantee that no version conflict will happen while upgrading to a newer version. Additionally, versioning in general and the **AUTOUPDATE** feature in particular contribute to an easy and robust implementation of the **viral propagation** method widely used in WSN by avoiding loops in broadcast transmissions.

4.5 Code Robustness

The last aspect we discuss in this chapter is **code robustness**. Code robustness is an essential part of any computer system. In the networking it becomes a critical point as the number of potential sources of errors is bigger: in addition to memory and execution faults, also we have interference (powerline and wireless), packet losses, incorrect signal reception and detection due to the use of weak radio chips. If most network traffic consists of data packets containing user information (content), which is the case in traditional networks (e.g., Internet), those faults could lead to an incorrect content to get delivered, or to a malfunctioning service to get provided, the network functionality will remain the same. In case of active networking model and migrating code the situation changes. Having lost or misinterpreted packets may lead to critical issues, and eventually a collapse.

Code robustness can be seen from many different angles: structural robustness, algorithmic (behavioral) robustness, channel coding, robust instruction set.

Structural robustness refers to how software system is organized. A good example is modular architecture of all modern OS. If some module starts malfunctioning, it can be safely removed from the system and replaced by the good one, the system stays alive.

Algorithmic robustness allows the code to encapsulate some anti-fault strategies. Compared to the structural robustness which comes from the outside (in our example – from the OS), the algorithmic one

¹*FragletVM* does not support the concept of code versioning.

4. Overview of Code Optimization Techniques

makes code self-protective: the code can potentially detect if something goes wrong, and take a set of actions to prevent or to heal malfunctioning behavior. One of the methods using algorithmic robustness is based on **Quines**, programs with no input which produce a copy of their own source code as the only output. The main idea of using Quines is to have pieces of code which are continuously replicating (regenerating) themselves. In case of code deletion the corresponding Quine will create a new copy of the missing code fragment. Corrupted code will not survive and will be erased as, most probably, it will not be able to replicate itself. The first attempt of using Quines for building network protocols was made in the context of CNP [MSTY08].

The next type of robustness exploits the idea of securing the communication channel used for packet exchange. The technique is called **Forward Error Correction (FEC)** and is widely used in radio communications as well as in storage medias (e.g., *Reed-Solomon* codes in CD-drives, *SECDED (Single-Error-Correcting and Double-Error-Detecting)* variant of *Hamming* code in RAM). Sometimes FEC is combined with the re-transmission Automatic Repeat reQuest (ARQ) scheme to form a class of hybrid error-control methods. A lot of studies have been done on applying FEC techniques to WSN domain. Various error control coding schemes and the impact of their use on error rates and energy efficiency have been analyzed, namely, simple *systematic XOR-based* schemes like *SECDED* and *DECTED* [JE03], *BCH* and *Reed-Solomon (RS)* codes [KFC04], *convolutional* [DOM06] codes, as well as some modifications of the above methods. With no surprise, *RS* codes with different error correction capabilities prove to be the best choice for WSN. Having said that, many papers argue the benefit of using FEC in WSN to be rather minimal. The reason for that is the strict requirements on packet size and power saving in WSN. Most WSN platforms set the limit on packet size to approximately 30 bytes of payload. As soon as FEC assumes introducing a significant overhead this becomes unacceptable for many applications as it requires more energy spending thus reducing a lifetime of the system.

A number of hybrid schemes have been proposed to deal with increasing energy consumption. Some of those solutions combine error correction with routing [WDHN03], some provide multi-hop decoding methods [QR07]. *Packet combining* [DFEV06a] also uses the idea of multi-hop processing for corrupted data recovery. But the

experiments have shown that a simple CRC (Cycling Redundancy Check) in each packet (for error detection) accompanied with ARQ (for error correction) still remains the main error-control approach in the area.

Some recent studies have made an attempt to make a use of modern codes in WSN. One of them can be seen as a developing of the *Deluge* idea which we discussed earlier in Section 3.1.1. *Deluge* implement error recovery through ARQ. In order to deal with multiple channel errors *Synapse* [RZS⁺08] propose to use a hybrid ARQ scheme in which the prior encoding is done using *Fountain* codes. *Deluge* and *Synapse* naturally focus on dissemination whole program images, that is big amounts of bulky data (code).

In our work we decided to have a look at the problem of **robustness from the instruction set point of view**. In our system we assume using some kind of pre-compiled interpretable meta-code which is similar to byte-code. Code pieces are normally small to make them fit into one packet (~ 30 bytes). Having those two requirements it becomes obvious that there is no room for FEC. Another important aspect is that the meta-code is intended to implement real-time tasks meaning that ARQ is not the answer either. The next question we faced was: is it possible to encapsulate robustness in the code at the instruction level, and if yes what degree of robustness we could obtain? We have not found the ultimate solution for this problem yet but some considerations are present below.

First, in computing, there is a so-called **robustness principle** stating the following:¹

“Be conservative in what you send; be liberal in what you accept.”

In terms of communication protocols this principle can be reformulated as follows:

“Code that sends commands or data to other machines (or to other programs on the same machine) should conform completely to the specifications, but code that receives input should accept non-conformant input as long as the meaning is clear.”

¹Both formulations below taken from the Wikipedia article located at http://en.wikipedia.org/wiki/Robustness_principle.

4. Overview of Code Optimization Techniques

The other idea we apply is straightforward: The less code we have, the less errors can occur while executing or transmitting it.

The most obvious solution for making code more robust is to introduce some type of redundancy. Adding a complementary parity bit to each instruction would allow detection of 1-bit errors but add an overhead. The same would happen in case of using some type of majority filter. This method assumes duplicating each instruction multiple times. In case of “3:1” filter we will have:

```
1 ...
2 add
3 add -> sub
4 add
5 ...
```

If one of the consecutive **add** instructions is mutated to something else, say **sub**, the code will be able to detect it and execute **add** anyway.

Another option is to use the feature of *Hamming distance* (or *Hamming cube*) for encoding. That would mean stretching each instruction code to the farthest vertex (the biggest *Hamming distance*) on the *Hamming cube*. Lets illustrate this with Figure 4.4.

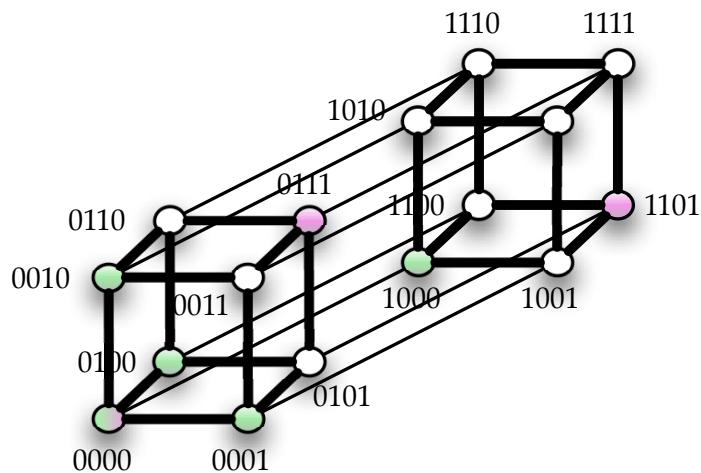


Figure 4.4: 4-bit Hamming Binary Hypercube

If we have 3 instructions we encode them as follows: 0000, 0111 and 1101. These three codes have the largest possible distance which is 3. This encoding is not unique, other combinations are possible as well (e.g., 0010, 0101 and 1111).

The bigger is code distance the less are the chances that one instruction mutates into an other. The hypercube can be grown further thus allowing to increase the distance between codewords but this means again introducing redundancy as more bits would be needed. One thing which hypercube allows us to do is to use a probabilistic execution approach. As 1-bit alterations are more probable than multi-bit ones, then if only 1 bit is flipped we could stretch the code back to the original one (the nearest). For example, if code 0000 has been mutated to 0010 (and 0010 is free), we push it back to 0000 and try to execute it. Alternatively, if we have enough free coding space, we can assign all neighboring vertexes to the same instruction: in the example above it would mean assigning 0000, 0001, 0010, 0100 and 1000 to the same instruction. In this case, if 1-bit error happens the code will still be executed correctly. Yet another approach proposed by Thomas Meyer in [Mey10] is to map all free instructions in the *Hamming hypercube* to the `nop` operator, so that if some instruction is mutated the execution is not interrupted and that instruction is just skipped.

The next thing to mention is that our method of dictionary-based instruction set naturally provides higher chances to detect incorrect instructions. First, instructions are supposed to be long (or to grow in size). Thus, error detection can be done by analyzing instruction patterns. Later, error correction can be done by choosing the instruction which actually fits the pattern. This refers to the idea of probabilistic execution stated above. Some execution patterns are more probable than others. By analyzing a code (which can be done in real time as well) we can rate occurrence of each instruction. If we look at the example in Listing 4.1 once again we can generate an occurrence rate for each instruction pair (see Figure 4.5).

By analyzing this pattern we can see that `exit` are more probable to follow `led` instruction in this example, and in case if we meet an unknown instruction followed by `exit` we could assume that we should execute `led`.

This probabilistic approach has some shortcomings. For example, errors in arguments (not in their number), or two instructions with

4. Overview of Code Optimization Techniques

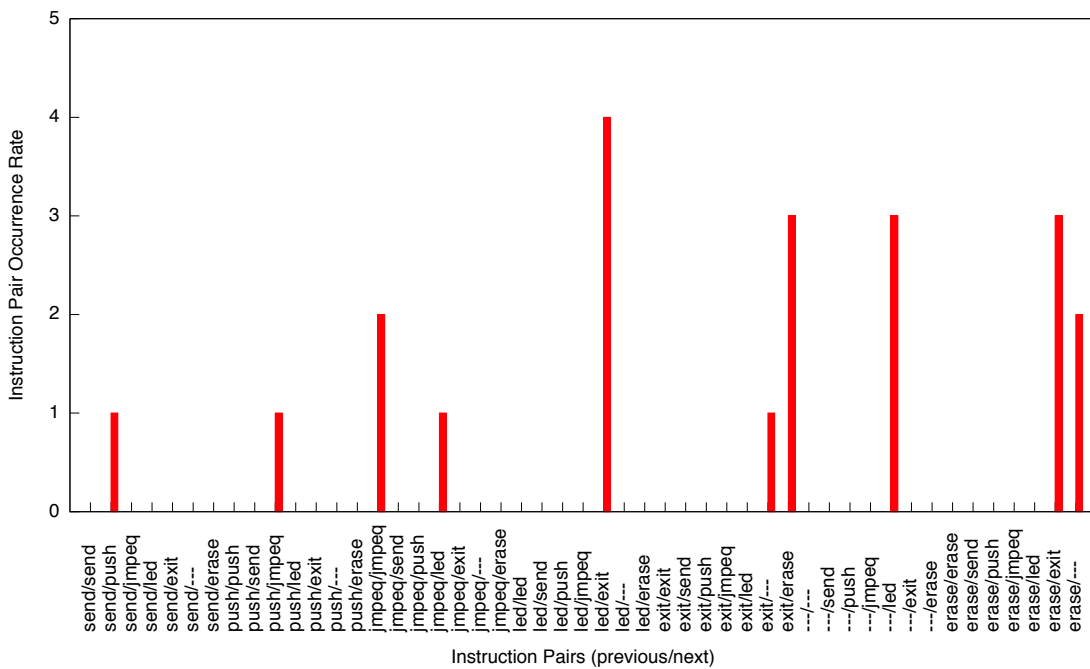


Figure 4.5: Instruction Pair Occurrence Rate

identical patterns are undetectable. Harvard architecture (physically separated storage and signal pathways for instructions and data)?

Using dictionary-based instruction sets give us one more option to detect errors. Dictionary errors might be controlled by locally storing replicas of neighbors' dictionaries.

4.6 Summary

In this chapter, we have described and analyzed five main aspects of run-time optimization of mobile code in network ES: code shrinking, compression, polymorphism, versioning and lifecycle, robustness. Each of these has a significant contribution to the process of changing code representation. For successful operation the system must include all these blocks. However, we have decided to focus our attention on the code compression methods as the most interesting step of the optimization. We have presented the existing work in the area of data and code compression and outlined what is currently missing. It turned out that there is no specialized technique has been developed for use with short instruction sequences. We have planned to do more research in the field of dictionary encoders and try to apply those methods to vari-

ous types of code streams, especially the ones we already developed as part of the two network engines from Chapter 3.

5

Online Code Compression Framework

This chapter presents an online code compression scheme, the core part of the dynamic code morphing framework introduced in Section 1.5.4. In cooperation with the code shrinking and code polymorphism discussed in Chapter 4, code compression provides more efficient code representation which eventually allows reduction in code transmission energy costs. This is especially critical for battery-supplied embedded systems (ES) like WSN. In the beginning we carry out analysis on existing hardware and software based code compression schemes in ES. Then we describe our method in detail and discuss possible application scenarios of different complexity: single-node compression, compression in a group of nodes, distributed version, cloud-compression for networks using Virtual Segmentation (see Section 1.5.1). We look at various aspects of the solution in order to understand its efficiency in terms of compression factor, convergence speed and with regard to the existing schemes mentioned in Section 4.2.

5.1 Kolmogorov Complexity of Code Streams

The field of code compression relates to the Algorithmic Information Theory (AIT) which studies relationship between computation, information and randomness, or, in particular, complexity measures on strings (or other data structures). Any code stream can be described in terms of strings and symbols, thus AIT can be applied here.

In the context of AIT the information content of a string is equivalent to the length of the shortest possible self-contained representation of that string, which is in fact a program in some language. This program outputs the original string. As it can easily be seen this essentially employs the concept of compressibility. Sometimes the term **algorithmic information** is also referred to as **Kolmogorov Complexity (KC)** or **Algorithmic Complexity (AC)**.

Although KC in general is an incomputable function, here in order to establish a correlation between compressibility of code streams and KC we use it as a numeric metric of complexity and compressibility. Let us consider an example where the string

"010101010101010101"

has the short description "10 of '01'", while the string

"01101011110110101110"

most probably has no simpler description other than the original string.

Now we can calculate KC for both strings: $K(s_1) = 8$ and $K(s_2) = 20$. Note, KC is not a unique function; it depends on the selected representation. For example, if we choose to represent the first string as "10x01" then KC of this string would be $K(s_1) = 5$.

In the next two sections we will try to translate the concept of KC to the domain of code compression. We will start from a very common compression case in Section 5.1.1 and move towards the code compression in Section 5.1.2.¹ This will help us to understand what can be expected from the proposed method.

¹The text in Sections 5.1.1–5.1.2 partially cites and is based on the Wikipedia articles rooted at http://en.wikipedia.org/wiki/Kolmogorov_complexity (these pages contain proper referencing to the original sources). Copyrights are held by the corresponding authors.

5.1.1 Translation of KC for a Common Compression

It is straightforward to compute upper bounds for $K(s)$: simply compress the string s with some method, implement the corresponding decompressor in the chosen language, concatenate the decompressor to the compressed string, and measure the resulting string's length.

A string s is compressible by a number c if it has a description whose length does not exceed $|s| - c$. This is equivalent to saying $K(s) \leq |s| - c$. Otherwise s is incompressible by c . Incompressible strings exist, since there are 2^n bit strings of length n but only $2^n - 1$ shorter strings, that is strings of length $n - 1$ or less.

For the same reason, most strings cannot be significantly compressed: $K(s)$ is not much smaller than $|s|$, the length of s in bits.

Additionally, in the theory it has been proven that with the uniform probability distribution on the space of bitstrings of length n , the probability that a string is incompressible by c is at least $1 - 2^{-c+1} + 2^{-n}$.

5.1.2 Translation of KC for Online Code Compression

First, we specify a description language L for strings (code stream). In our case these are two assembler-like dialects (*ChameleonVM*, *Agilla*), L_C and L_A , and one prefix-rewriting language (*FragletVM*), L_F . These were discussed in Sections 3.2 and 3.3, respectively. An encoding is a function which associates to each Turing-machine M (this is our compression engine) a bitstring $\langle M \rangle$. If M is a Turing-machine which on input x outputs string s , then the concatenated string $\langle M \rangle x$ is a description of s .

As it has been said, formally, the KC of a string is the length of the string's shortest description in some fixed description (programming) language L , or its sub-class (L_C , L_A or L_F). This can be written down as follows:

$$K(s) = |\mathit{description}(s)| = |\langle M \rangle x| \quad (5.1)$$

In AIT there are many important theorems and corollaries, which relate to the KC. Here we provide only a few, which seem to be relevant to our further work:

5. Online Code Compression Framework

- The choice of description language affects the value of KS and the effect of changing the description language on the value of KC is bounded:

$$\forall s, L_1, L_2 |K_1(s) - K_2(s)| \leq c$$

This means that by changing the description language (e.g., from L_C to L_F) the KC of the encoded stream will change only by the additive constant which depends only on the language.

- It can be shown that the KC of any string cannot be more than a few bytes larger than the length of the string itself:

$$\forall s K(s) \leq |s| + c$$

In the worst-case scenario when the compression fails to reduce the size of the input stream we can expect at least that it will oversize it too much with system information.

- Among algorithms that decode strings from their descriptions (codes) there exists an optimal one. This algorithm, for all strings, allows codes as short as allowed by any other algorithm up to an additive constant that depends on the algorithms, but not on the strings themselves [Kol65]. This theorem means that the algorithm will work for any program if it works for one.
- “Full employment theorem”: There is no perfect size-optimizing compiler.
- “Chaitin’s incompleteness theorem”: In the set of all possible strings, most strings are complex in the sense that they cannot be described in any significantly “compressed” way. However, the fact that a specific string is complex cannot be formally proved, if the string’s complexity is above a certain threshold.

The last two rather theoretical statements leave us with a chance that some types of code streams might stay incompressible or that they would require a different compression approach.

5.2 Analysis of Existing Solutions

In Chapter 1 we discussed our motivation behind developing a new code compression scheme, as well as a number of supportive concepts

our design involves, namely, the task-oriented approach and network VS. Later, in Section 4.2 we listed and analyzed the most notable general compression techniques. Those methods can be applied for any type of stream but the question of effectiveness remains open. Here, we first would like to analyze the existing compression solutions, which are more specifically tailored to code streams and make a statement as to why there is a need for a new method.

ES feature very limited computational and energy resources. Normally, “sending a single bit of data can consume the same energy as executing thousands of instructions to produce that bit of data” [HM06]. Hence, more should be done on representation of what is to go out before the actual transmission. In Section 4.2 we came to the conclusion that all known data compression schemes more or less fail while being applied to the code. Although these methods offer a number of fundamental features which we will use in our design.

Different criteria can be used to categorize code compression techniques. In general, they can be grouped as follows:

- Compression of pre-compiled platform native code using standard compression algorithms from the data world.
- Various native code optimization tricks at compile-time.¹
- Bytecode compression.
- Modification to ISA to provide shorter opcodes for most frequently used instructions.
- Providing ad-hoc ISA sub-sets for different tasks (e.g., for DSP-related operations).
- User-definable ISA.

These methods can be either compile-time or run-time based and they can be implemented either in hard- or software. Additionally, they can be specifically designed either for the native (binary) code or some sort of bytecode (e.g., Java).

¹Also, some works exist where a binary file is optimized (link-time program rewriting) instead of the source code which allows to compact statically allocated data as well [SBB05].

5. Online Code Compression Framework

5.2.1 Data Compression Techniques for Code

A number of attempts have been made to apply standard data-oriented compression methods to code.

In [TDV08b] the authors use *DEFLATE* [19596] (*GZIP* [Deu96]) to compress loadable binary *ContikiOS*'s ELF modules. They report a 67% dissemination time cut (multi-hop) and a 69% energy consumption saving if *GZIP* is in use. Additionally, they state that using other compression methods like *arithmetic* coding [RL79], *VCDIFF* [KMMV02], *LZARI*, *LZO1X*, *S-LZW* [SM06] and *SBZIP* (*Burrows-Wheeler Transform* [BW94]) show worse results in terms of performance but sometimes those ones feature a smaller memory footprint which might be critically beneficial for some applications.

The authors of [MuuR07] apply several general-purpose data compression techniques to software-based self-test (SBST) programs in WSN. They demonstrate node energy savings by using an adaptive test program compression scheme with a small memory footprint based upon the *Bentley-Sleator-Tarjan-Wei* (*BSTW*) [BSTW86] algorithm in conjunction with *Golomb-Rice* coding. By comparing it to *LZW* and *Dynamic Huffman* (a.k.a., *Faller-Gallager-Knuth* (*FGK*) algorithm) implementations they obtain the results which show that *BSTW*'s compression rates are more than two times lower than *LZW* (however, sometimes *BSTW* outperforms *FGK*) but in terms of memory footprint their solution is unbeatable taking, 15 times less memory space than *LZW*.

The *TinyVM* [BSE06] (forked into *leJOS*, Java for *LEGO Mindstorms*) project offered a specific bytecode/binary code mixed-mode execution environment for WSN where frequently executed code is compiled to binary code and less frequently executed code is compiled to bytecode. This approach is similar to [HATW99] used in the *TriMedia* VLIW processor. To disseminate the bytecode part of the resulting image they split the bytecode stream into op-code stream, number stream and symbol stream [DE02]. After that each stream is compressed using *Huffman* encoding. Adding a dictionary-based approach further improves compression rates. Their experiments showed that *Huffman*-encoded bytecode occupies only 36%–57% of the space of the corresponding binary code, it executes only 2–36 times slower on *TinyVM* (on *ARM Xscale* processor) than binary code with a speed-up of a factor of 3.

Another approach is to incorporate compression into the instruction fetch inside a VM using *Huffman* codes [LF03].

The solutions above are meant to work with a program image. This means that decompression is done on a target by some software module. After that the code is fed into an actual code processor.¹ A number of schemes have been proposed where the compression engine becomes an integral part of the CPU itself. In this case, the code processor “naturally” knows how to execute the compressed code. This does not eliminate a need for the decoding stage but at least no additional software support is required. The solutions of this kind featuring *dictionary-based*, *arithmetic-* and *Huffman-coding*, *Hamming distance* and *operand factorization* exist for RISC [WC92] and VLIW architectures [RS04].

As can be seen all the solutions above utilize the classic data compression methods. In our opinion this limits their use and performance drastically.

5.2.2 Compression of Native vs Bytecode

A number of original bytecode compression schemes have been proposed. Most of those solutions are Java-oriented but to some extent could be applied to other forms of bytecode. All of them are rather complex and sophisticated and do not meet the resource requirements of most WSN platforms [Pug99], [CSCM00].

Native code compression is similar to bytecode but uses specific features of the binary format for a specific platform. This normally allows to achieve better compression rates [CM99], [EEF⁺97], [LW99], [LBCM97], [LM98].

Both approaches are static and software-based; an image is compressed at compile-time.

5.2.3 Instruction Set Compression

Traditionally, code compression is seen as compacting pre-compiled code, binary or any form of byte-code (see Sections 5.2.1 and 5.2.2). We consider it as a problem of changing code representation at runtime by modifying (optimizing) the instruction set. We applied this technique to our own byte-code by creating a task-specific ISA (in fact,

¹Which can also be software-based in case it is a VM.

5. Online Code Compression Framework

a specification for the byte-code) at run-time. It turns out that some similar solutions can be found in the field of microprocessor ISA.

Most of the related techniques are implemented in hardware. Therefore, they are hard-wired and not flexible. On the other hand, hardware implementations offer much better performance measures, which are not achievable using software solutions. Two groups of hardware compression techniques exist:

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

Ad-hoc ISA: Typically specifies two instruction modes: compressed and uncompressed. The main advantage is that decoding is simple and fast since instructions stay compressed in cache. However, decompression is on the critical path and compression rates are low. Mostly used in RISC embedded processors.

Hardware implementations of code compression are used inside microprocessors in order to reduce memory footprint of the on-board program image. Mainly those architectures are used in embedded solutions where memory resources are limited. The reason why compression is used is because all those microprocessors employ RISC architecture, which means they have poor code density. CISC families, such as *Motorola 68K* and *Intel x86*, do not need code compression – their standard code density is still better than the compressed modes of the RISC chips. The architectures discussed below belong to the class of RISC-processors. The survey below partially cites and is based on [Tur04].

ARM Thumb: *Thumb* is a second, independent instruction set mapped onto ARM's standard RISC instruction set. One can switch between the two instruction sets through a mode-switch instruction in the program code. The *Thumb* ISA consists of 16-bit instructions, about 36 in total. By using short instructions instead of ARM's normal 32-bit instructions the size of some code can be reduced by 20 to 30%. *Thumb* code cannot be

5.2 Analysis of Existing Solutions

intermixed with normal *ARM* code. Instead, an explicit switch between standard ISA and *Thumb* is required every time. *Thumb* includes the basic add, subtract, branch, and rotate operations. Because of its simplistic nature some operations are not possible in *Thumb* mode. For example, *Thumb* cannot handle interrupts, exceptions, long-displacement jumps, atomic memory transactions or coprocessor operations. This limits its use to basic arithmetic or logical operations. Everything else has to be done using *ARM*'s standard 32-bit instructions. Parameters are passed between *ARM* code and *Thumb* code through the stack or through a number of registers available in both modes. Switching to and from the *Thumb* mode also takes time and adds code. Several dozen bytes of preamble and postamble are needed to organize pointers and flush the CPU pipeline. Because of this overhead switching makes sense only if the processor remains in *Thumb* mode for several dozen instructions at a time. Performance in *Thumb* mode drops by 15%. This is mainly caused by the overhead of switching between 16- and 32-bit modes. *Thumb* instructions are also less flexible than their 32-bit counterparts (more *Thumb* instructions are needed to do the same job compared to 32-bit instructions). However, *Thumb* makes caches more effective because the instructions are only half as long.

MIPS: *MIPS* employs a second, 16-bit instruction similar to *ARM*'s *Thumb*. The *MIPS16e* instruction set includes a bunch of 16-bit shorthand versions of standard *MIPS* arithmetic, logic, and branch operations. As with *Thumb*, one has to switch in and out of *MIPS16e* mode, a process that introduces some overhead in time and code space. As with *Thumb*, the space savings amount to 20 to 30% in most cases. Neither *MIPS16e* nor *Thumb* really compress code. They just offer alternative opcodes for some operations and the amount of "compression" seen depends on the ratio of short opcodes to long ones. That, in turn, depends on what the code is doing. System-level code, like operating systems and interrupt handlers, cannot use 16-bit instructions at all since its functionality is not enough, so they do not benefit in anyway. Long computations using a lot of arithmetic show a relatively good response to compression. Data does not compress at all, only code. If applications include a lot of static data structures the overall memory savings may be small and at the same time performance degrades by 15%.

5. Online Code Compression Framework

Thumb-2: *Thumb-2* operates like *ARCompact* (see Section 5.2.4) or *Motorola 68K*, allowing to mix 16-bit and 32-bit instructions without mode switching. Overall, *Thumb-2* offers a little bit less compression than *Thumb* but with a less reduction in performance. To this end, *ARM* has a hole in its opcode map: **BL** (“branch and link”), the instruction that switches between *Thumb* and *ARM* modes. **BL** has some unused opcode bits, those previously undefined bit patterns now provide an extension of a whole new instruction set. The biggest advantage of *Thumb-2* is that it is a complete ISA. Programs need never switch back to “normal” 32-bit *ARM* mode. Programs can now handle interrupts, set up MMUs, manage caches and generally behave like real microprocessors. However, *Thumb-2* still causes a decrease in performance. Even though, there is no mode-switching overhead it still takes more *Thumb-2* instructions to perform certain tasks compared with standard *ARM* code. Those extra instructions (and extra cycles) add up to between 15 and 25% speed lost. Eventually, *Thumb-2* replaces both the *ARM* and *Thumb* instruction sets with a single, more compact instruction set.

As can be seen, in the case of *MIPS16e* and *ARM Thumb*, one can choose which parts of code to compact and which not. *PowerPC CodePack* offers compression of an entire image only.

IBM PowerPC CodePack: This approach is hybrid. Compression is done in software, decompression in hardware. Unlike *Thumb* and *MIPS16e*, *CodePack* really does compress executable code. It works like running a compression program (e.g., *GZIP*) on *PowerPC* code. *CodePack* analyzes and compresses entire programs, producing a compressed version that has to be decompressed and executed on the fly. For all its complexity, *CodePack* delivers about the same 20 to 30% space savings as the others. *PowerPC* code is compiled in the normal way, using standard tools. *CodePack* even works on existing code, with or without source code. Before the code is burnt into ROM it is run through the *CodePack* compression utility. It analyzes instruction distributions and produces a pair of unique keys specific to this program only (the upper and lower 16 bits are compressed separately because the upper half of each *PowerPC* instruction (which holds the opcodes bits) has a different frequency distribution than the lower half (which typically holds constants, displacements, or masks – using two different compression algorithms produced better results than any single

algorithm). When this compressed program is executed, a *CodePack*-equipped processor uses the keys to decrypt the compressed code on the fly. The decompression adds a tiny amount of latency to the processor's pipeline. In general, *CodePack*'s performance effects are negligible. However, a number of shortcomings exist. Since every compressed program produces a different set of compression keys, *CodePack* is essentially an encryption system as well as a compression system. Without the keys a program cannot be executed. This also means that compressed *PowerPC* programs are not binary compatible. One cannot just exchange compressed programs with other systems unless their decompression keys are also provided. This makes code distribution tricky.

The main problem with hardware implementations is that they are not scalable and in most cases cannot be modified. They are tailored to a specific instruction set. Most compression schemes have a negative effect on performance. Moreover, compression rate highly depends on the code being compressed.

5.2.4 User-Definable ISA

This class introduces a task-specific instruction set for every job. Both software-based (e.g., *ASVM* [LGC05], [LGC04] based on *Maté*), hardware-based (e.g., *Xtensa*, *ARCompact*) and hybrid (e.g., *Altera NIOS* soft-core) implementations exist. ISA is modified offline and tailored to perform a particular task. Further changes are not possible. This approach gives the best performance gain.

***ARCompact*:** The *ARCtangent* processor has a user-definable instruction set. Hence, any changes to ISA are possible. In the case of *ARCompact*, a number of 16-bit instructions are added to improve code density. What makes *ARCompact* different from *Thumb* and *MIPS16e*, is that it can freely intermix 16-bit and 32-bit instructions. This process requires no mode switching and there is no overhead. *ARC*'s compilers insert 16-bit operations whenever possible. Though the code 32-bit alignment might suffer. The reason why mixing is possible is because compared to *ARM*, *MIPS* or *PowerPC*, *ARC* has a bit in the instruction word to determine size. This allows having variable-length instructions. The feature exists on other architectures like *Tensilica Xtensa*, *Intel x86* or *Motorola 68K* as well.

5. Online Code Compression Framework

NIOS-II: Is a 32-bit RISC soft-core processor architecture which is implemented entirely in the programmable logic and memory blocks of *Altera* FPGA. The soft-core allows the user to specify and generate a custom instruction set tailored to a specific application's requirements.

ASVM: This is an extension for the *Maté* VM mentioned in Section 3.1.2.1. It allows for definition of application-specific bytecode instructions at run-time and to compile it as a part of the VM image. Following this, disseminated programs can use the newly defined opcodes, increasing the code density.

Custom instruction sets provide a very high level of flexibility. They allow the task's representation to reach near optimum. The only problem is that all the solutions are static. Run-time re-definition of ISA is not possible.

5.3 Requirements

Having outlined the existing work in the field we are now ready to formulate what we actually expect from the new scheme.

First, code compression becomes pointless in most configurations, which do not use communications as per bit of memory storage price is dropping and the memory chips' density is going up. Our scheme was originally designed for mobile code, every bit of which being added/removed affects the system's lifetime.

The hardware and software solutions presented in Section 5.2 lack a number of features which are necessary for running programs based on mobile code. For our purposes we mainly assume building network protocols using the network paradigm discussed in Section 3.1.3. Therefore, in the design of our compression scheme we tried to address the following issues:

Run-time operation: Our compression scheme works "online", which means that code is being compressed/decompressed at run-time while the system is running. Pre-compression of the entire program image is not required, nor pre-decompression.¹ The need to

¹To be precise, compression/decompression phases exist but they are transparent from the execution engine. The only cause of delay is the dictionary search operation, which is fast because the dictionaries are small.

meet tight time requirements is critical for building network protocols. The response time for such a system should be negligible in respect to time characteristics of protocols built on top of it.

On-the-fly support: Ability to execute compressed code is important to provide the run-time operation discussed above. Duplication of capsules as mentioned in Section 3.2.11 is the process when capsules are stored in the on-board storage in a decompressed form only. The compression process, running constantly, creates a compressed reference, which is updated as new rules are being introduced. When a capsule leaves a node the compressed reference is sent out. This approach allows compression of code and at the same time no loss of speed while the code is being executed on a node. However, the compression engine can be configured to work in a full on the fly mode – no references are created, the storage contains compressed code.

Small memory footprint: Using compression schemes from Section 4.2 normally requires a lot of memory resources. Those compression/decompression algorithms are computationally expensive. Dictionary compression methods require a lot of space to store dictionary structures. We designed our scheme with the very limited memory resources of ES in mind. Dictionary structure allows representation of even very complicated instructions with very few bits of information. During dictionary synchronization only delta is transmitted, this operation does not consume too much energy. Moreover, dictionary size can be regulated using various parameters, which will not allow it to grow unexpectedly.

No pre-deployed dictionary: Since an instruction dictionary is created while the system is already in a running state, no pre-distribution of dictionaries is required. This saves power on dissemination of the dictionaries.

Adaptability: This is a key factor for building task-specific configurations. Our system can adapt to changes in the environment. The configuration of software, which is currently in use has impact on code compression, this impact becomes visible over time. Furthermore, introduction of the idea to use network VS (overlaid task clouds) allows extension of adaptability to the space domain as explained in Section 1.5.1.

5. Online Code Compression Framework

As shown in Figure 5.1, it might take some time until the system reaches a compressed form (see Section 5.4.9). By introducing new software pieces or removing old ones, we will break this equilibrium. In order for it to settle down again it will take some time.

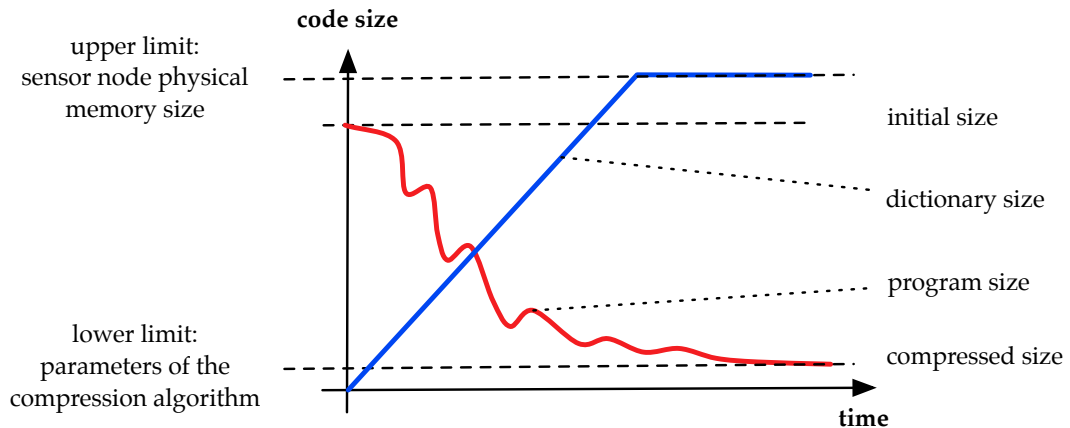


Figure 5.1: Characteristic of Online Compression in Time and Resource Domains

From the “initial code size” the system shrinks down to the level given by the parameters of the compression scheme (e.g., alphabet size, free space, etc. – see Section 5.4.8 below). While program size has a tendency to get smaller, dictionary size grows inversely. Some fluctuations in the program size reduction curve are caused by wrong decisions, which the system might occasionally make while trying to compress the code. As previously stated, it can also happen when new software pieces are introduced into the system and old ones leave. This mechanism is further discussed in Section 5.4.

Similarly to a lower limit the system also has an upper limit defined by the available physical memory on a node. This may cause the dictionary to stop growing at some point where its size gets too big.

5.4 System Architecture

Our compression scheme is based on the principle of constantly re-writing instruction sequences in the code stream, i.e., **dynamic continuous instruction re-encoding**. When a piece of code (a capsule in case

of *ChameleonVM* or a fraglet in case of *FragletVM*) is picked from the local storage for execution, it is sent through a chain of modules which analyze the stream, make a decision on what instructions should be combined next and rewrite the original stream according to the new rules. This mechanism is shown in Figure 5.2.

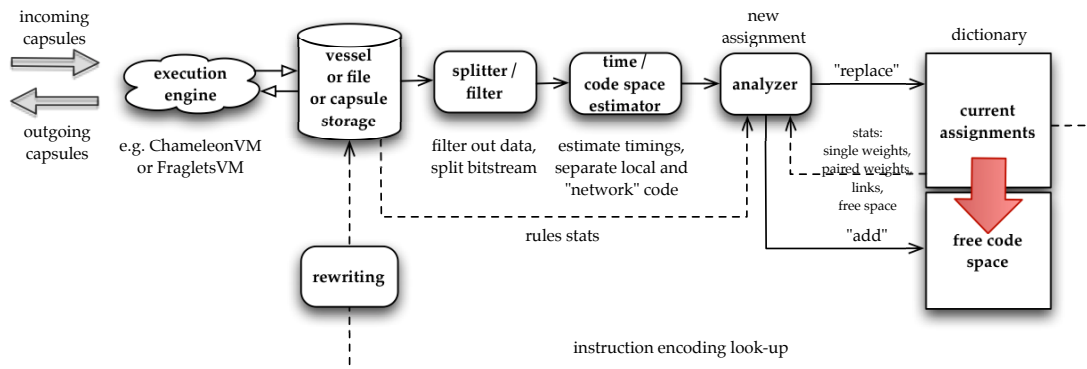


Figure 5.2: Online Compression Scheme: System Architecture

5.4.1 Pair-Wise Stream Search

Code fragments are picked from the capsule storage (a.k.a., **vessel** in *FragletVM*). If we deal with fragmented code stream (see Section 5.4.4), then the *splitter* picks the next code piece (i.e., fraglet). At the same time the *filter* separates the data and code (see Section 5.4.5). Time and code space estimation is then performed. At this stage the system tries to figure out if the further changes would still allow the code to meet time requirements (if the code has to go out immediately the execution interrupts), local code gets separated from the one, which is supposed to be sent later. Later, the result is fed into the *analyzer* which calculates various metrics for the code (weights for each instruction, each instruction pair, etc.). The information already stored in the dictionary is used too. The analyzer is also responsible for making the final decision on which new instruction assignment should be made next. When the decision is made the corresponding change is committed to

5. Online Code Compression Framework

the on-board dictionary. After that the original code is re-written using new encoding and put back into the storage.¹

Code analysis is the central part of the chain described above. Our algorithm for finding the next possible assignment is based on the idea of using a pair-wise sliding window as shown in Figure 5.3.

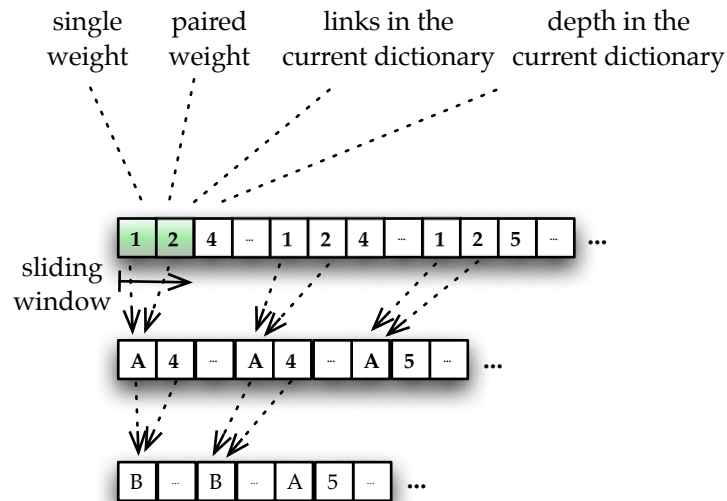


Figure 5.3: Online Compression Scheme: Sliding Window

For the supplied code stream we first calculate the weights of each single instruction, then for each pair of consecutive instructions. By doing this we also keep track of how many links (appearance rate) each instruction already has in the dictionary as well as the level of nesting (depth). Based on this information the most promising pair of instructions is picked. This pair is encoded as a new single instruction. After that the original code stream is re-written where the picked pair is replaced with the new opcode, and the process repeats again.

Example. Let us demonstrate how this works using a concrete example. We will use the code from Listings 4.1 and 4.2, Section 4.1. This example code is simple and has many patterns. For simplification, let's

¹*ChameleonVM* saves a compressed reference to the original code which is used only at the communication stage. The local uncompressed copy is used further by the execution engine.

eliminate all data fields from the original stream first.¹ Now, we use a search algorithm described above to find the most frequent pairs and pack them in a dictionary by replacing with new opcodes. The iterative processes for the two code pieces are shown in Figures 5.4 and 5.5, respectively, in a form of a compression tree; the instruction sequences being replaced are marked with blue.

Note that `---` operators intervene the compression process, as they are used for code shrinking as described in Section 4.1. The compression stops when there are no more pairs of instructions. We assume we have at least 3 free opcodes, otherwise the compression will stop earlier. In the first example (see Figures 5.4), we can observe that instruction pairs having the same weight are chosen randomly which is not exactly true as the pre-existing dictionary content can influence the selection process (see Section 5.4.2). In the second example (see Figures 5.5), we assume that we have only 3 free opcodes, no more. This pushes the `instr3` to grow instead of creating new entries in the dictionary. We can also see that the algorithm is smart enough to discover overlapping sequences and to correspondingly adjust their weights (marked with red in Figure 5.5).

In these examples the selection is straightforward because of the structure of the code. In other cases, the selection procedure might require more steps than just a pair-wise weighting, as described in Section 5.4.2.

5.4.2 Selection Algorithm

The algorithm of selecting the next pair of instructions is straightforward and is based on analysis of the frequency distribution of instructions in the original stream. The following criterion is used first:

$$\max \omega(A_i, A_{i+1})$$

as a pair of consecutive instructions having the biggest weight (appearance frequency). If only one candidate exists, it is picked. If not, for all found pairs we collect the weight for each of the two members in the pair:

$$\omega(A_i), \omega(A_{i+1})$$

¹In fact, this is done by a compiler and the *ChameleonVM* engine at run-time.

5. Online Code Compression Framework

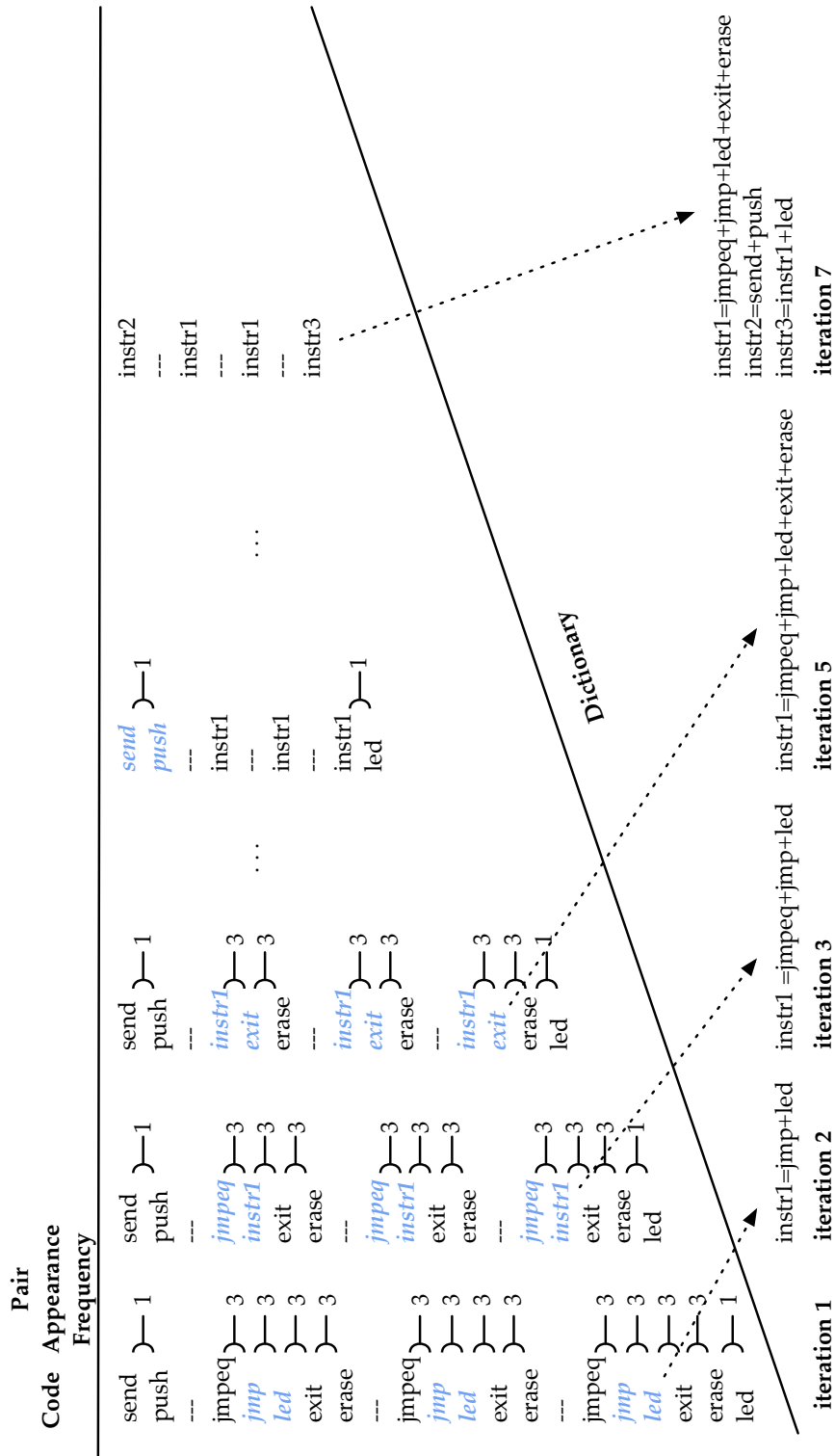


Figure 5.4: Online Compression Scheme: Compression Tree Example (for the code in Listing 4.2)

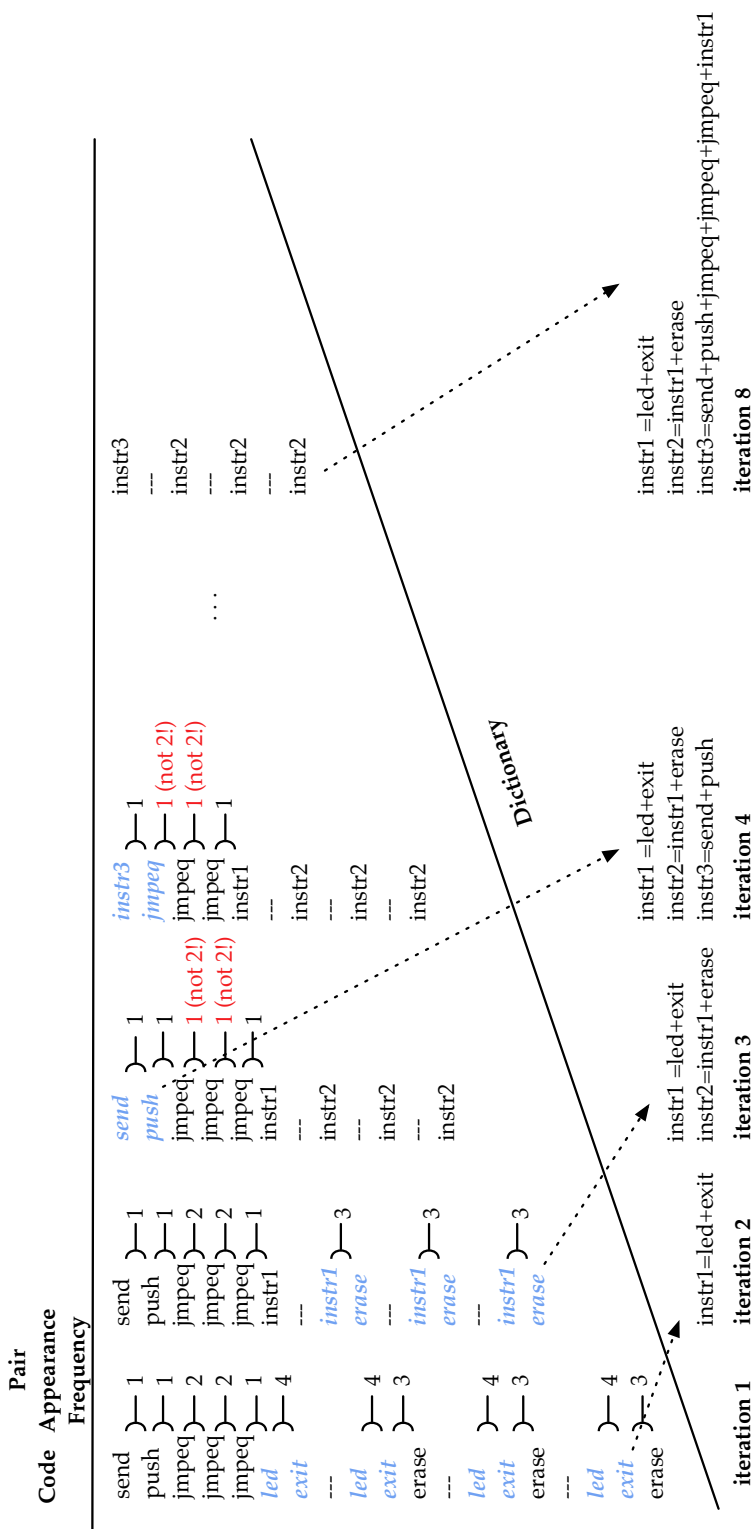


Figure 5.5: Online Compression Scheme: Compression Tree Example (for the code in Listing 4.1)

5. Online Code Compression Framework

and find the one having the maximum value of:

$$\max (\omega(A_i) + \omega(A_{i+1}))$$

If the one is found, it is picked. If not, in order to incorporate the information on a structure of the existing dictionary records in the equation we take the sum of the links the two instructions already have in the dictionary:

$$\max (\lambda(A_i) + \lambda(A_{i+1}))$$

Additionally, the information extracted from the existing dictionary gives the following constraints to the selection process:

- *no duplicates*: e.g., assignment $A \rightarrow D \rightarrow E$ is not allowed if $A \rightarrow B \rightarrow C$ already exists,
- *no cycles*: e.g., assignment $C \rightarrow A \rightarrow B$ is not allowed if $A \rightarrow B \rightarrow C \rightarrow D$ already exists, and
- *rough estimation on the effect of replacement*: e.g., assignments in a form of $A \rightarrow B \rightarrow C$ and $A \rightarrow D \rightarrow E \rightarrow F$ are estimated in order to understand if the replacement is profitable. This gives no guarantees. Nested sub-branches are not analyzed.

Using the above constraints, the pair with the highest number of links is selected. In case of still having multiple choices the random selection is made.

The proposed selection is simple and fast, at the same time it allows, with high probability, to find a nearly optimal pair for the next step. A more sophisticated algorithm can be used instead but this would lead to a very long time in the selection process (see more on convergence speed in Section 5.9). The fluctuations in Figure 5.1 are caused by the fact that selection is not always optimal.

5.4.3 Pairs vs Multi-Sequences

The selection process runs on a pair-basis as was shown in Figure 5.3. The reason being, that forming instructions into pairs provides more granularity for the selection process compared to triplets or larger sequences. Eventually, this allows to achieve better compression factors. However, it results in slowing down the convergence speed. The only

case where larger sequences would be better than a pair-based selection is equi-weight code streams, i.e., streams with no patterns. In reality, it is not the case.

Using the terminology from Section 5.1 we would say that the string will respond more quickly to compression based on multi-sequence selection if the string satisfies the property of **Kolmogorov Randomness**. This property defines a string “as being random if, and only if, it is shorter than any source that can produce that string”. Since “it is impossible to come up with a representation of a random string using a source with the length shorter than the length of the original string”, the theory declares that Kolmogorov random strings are “incompressible”. Below we show that even completely, randomly generated strings have patterns.

5.4.4 Continuous vs Fragmented Code Stream

In the analysis below we consider two cases: continuous and fragmented code streams. The ideal continuous code stream is non-reachable abstraction as any code stream has some level of fragmentation. This fragmentation is caused by mainly two reasons. The first is that code is naturally fragmented at the programming stage. In case of *ChameleonVM* we operate with capsules and their size has an upper bound dictated by the technological constraints of radio chips on WSN nodes. Fraglets are by definition separate computational units. The second cause of fragmentation is an ability of both capsules and fraglets to split and merge their execution flows (**split** and **merge** operations in *ChameleonVM*; **split(at)** and **match** in *FragletVM*). Obviously, if the execution flow splits into multiple threads or merges from multiple thread, it has an impact on the compression process, which would have to revert assignments in a form of: $A \rightarrow B \rightarrow \text{split} \rightarrow C$.

Reverting might be a very complicated operation affecting many dictionary records and breaking dependencies. In order to avoid this our compression algorithm interrupts and wraps up every time it meets **split** or **merge** operations and continues afterwards. As shown in Section 5.5, in the set of experiments with different fragmentation rates, this gives smaller compression factors but eliminates very complex reverse assignments which are critical for timings.

5. Online Code Compression Framework

5.4.5 Data and Code Mix

The von Neumann architecture assumes that code and data are mixed together in one stream. This is in contrast to the Harvard approach where code and data come from different locations. *ChameleonVM* uses a single-stack-based architecture where immediate operands are stored separately from the code. In *FragletVM* there is no distinction between code and data, the fraglets stream is prefix-driven in the sense that the type of the next item in the stream (operand, instruction or synchronization tag) depends on the type of the previous one.

In our research, we focus on compressing instruction sets and code only. Therefore, data is excluded from the compression process as shown in Figure 5.6.¹

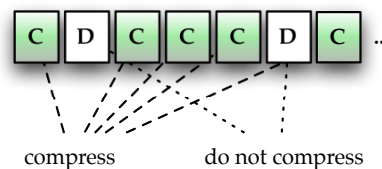


Figure 5.6: Online Compression Scheme: Data and Code Mix

This does not affect the performance. *ChameleonVM* is a stack-based machine which means that parameters are taken from the stack and the results are put back on the stack. Immediate operands are stored separately from the code segment (in the special region of BUFC data buffer) and accessed by an address. In case of *FragletVM*, there is no distinction between data and code which means that code can be treated like data and data like code depending on the execution context. This concept allows us to consider everything as code in this situation.

5.4.6 Dictionary Sub-Classing

ChameleonVM and *FragletVM* incorporate different dictionary models. In *ChameleonVM* each node can have multiple dictionaries for each profile and switch between them. Switching depends on the profile a node is currently executing. This is explained in more detail in

¹In case of *ChameleonVM* this picture is rather illustrative because there is no real intermix between code and data.

Sections 3.2.9 and 3.2.10 and illustrated in Figure 5.7. *FragletVM* holds a single dictionary which is created and maintained in a distributed fashion. Each node can commit changes to the common dictionary. Those changes take effect after they have been accepted by all the nodes (see Section 3.3.8).

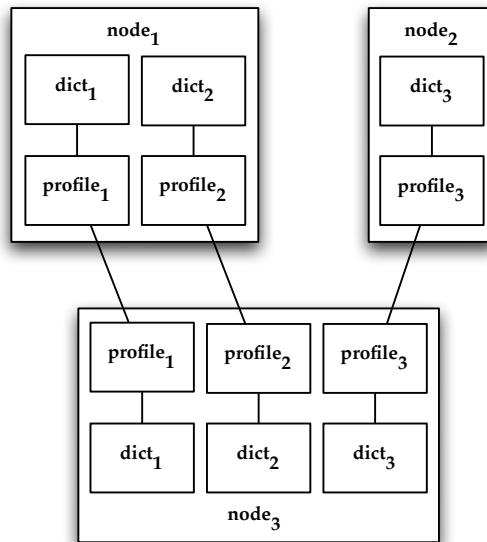


Figure 5.7: Online Compression Scheme: Dictionary-Profile Interlink

Regardless the engine we are using, each of those newly created task-specific dictionaries can be considered as a derivative from the original dictionary. We call this dictionary sub-classing and it is shown in Figure 5.8.

Assume that node A has a dictionary D^A consisting of $N - 1$ instructions. Each derivative D_i^A is, therefore, a sub-class of D^A consisting of $N_i - 1$ instructions.

The dictionary update mechanism is also different in *ChameleonVM* and *FragletVM*. While *ChameleonVM* employs a centralized approach with multiple leaders, in *FragletVM* it is built around a fully distributed architecture. As shown in Figure 5.9a, a *ChameleonVM*-based network can have multiple clouds (profiles), each of those profiles must have at least one leader. The profile leader runs the process of profile optimization. Each node can be a member of multiple profiles (e.g., nodes

5. Online Code Compression Framework

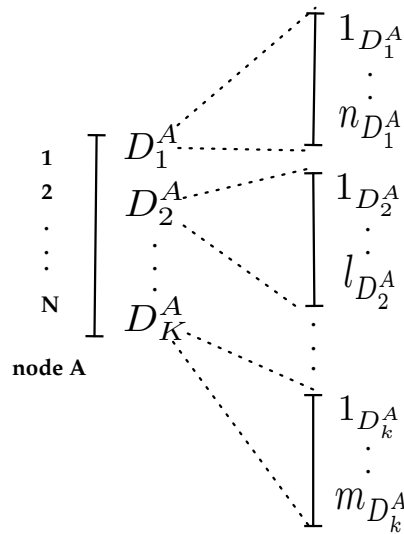


Figure 5.8: Online Compression Scheme: Dictionary Sub-Classing

2, 4 and 5 in Figure 5.9a).¹ With *FragletVM* every node in the network can become a temporary leader. A node should announce a change to the rest of the network. When the change is accepted all the nodes start using it. This is shown in Figure 5.9b.

5.4.7 Instruction Nesting and Unfolding

Instruction nesting happens when a new record is introduced into the dictionary. The opposite effect is known as **instruction unfolding**. Our compression algorithm uses a number of constraints described in Section 5.4.2. The main principle we exploit is to avoid a reverse of previously made assignments. Instead, the system keeps “hoping” to achieve better results over future steps. This is the reason why the compression curve shows fluctuation and reaches a local optimum at each step only. In other words, the compression process holds the Markov property.

5.4.8 Parameters

Before carrying out experiments on real code in Chapter 6 we run simulations on abstract, randomly generated strings of symbols representing

¹Nodes 3 and 5 in Figure 5.9a belong to different profiles and can communicate directly only using the common dictionary level D (see Figure 5.8).

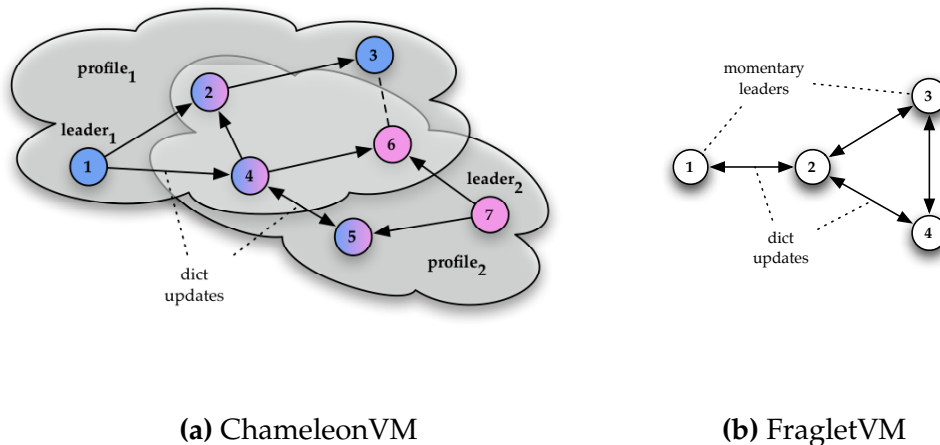


Figure 5.9: Online Compression Scheme: Dictionary Update Models

instructions. We analyze the influence of several system parameters. In particular, these parameters include:

- **Program Size (PS):** The size of code stream being compressed. Continuous code stream is defined by one single parameter, while fragmented stream by a combination of two: number of fragments (FN) and fragment length (FL).
- **Alphabet Size (AS):** The number of unique symbols (instructions) in the original code stream. This depends on the type of program.
- **Free Code Space (FCS):** Basically this parameter defines the dictionary capacity. By tuning this parameter we can regulate up to the size we allow the dictionary to grow. Bigger FCS provides more opportunities for the compression process to find a better representation for the code.

Note, in reality PS and AS are defined by the program, The only tunable parameter in a real setting is FCS. In fact, FCS shows the biggest impact on the compression performance metrics. Nevertheless, in the simulations below we have a closer look at each of the system parameters.

5.4.9 Algorithm Speed

Our compression scheme is a continuous process constantly running in parallel to other system activities. The faster the algorithm works

5. Online Code Compression Framework

the quicker better code representation can be found (the higher convergence speed). On the other hand, compression consumes system resources (memory, CPU). Additionally, dictionary updates have to be sent around. This consumes energy. Therefore, when the compression is running it should not affect system performance and interrupt other system services. In other words, the algorithm speed must be adapted in each particular case according to the duty cycle of the system. That is why in our experiments we measure compression speed in terms of the number of iterations rather than absolute time. Convergence speed is discussed in detail in Section 5.9.

5.4.10 Algorithm Complexity

In order to express our algorithm's complexity we use the "big-O" notation. By considering the selection algorithm from Section 5.4.2 we have:

$\mathcal{O}(n^2)$, where n is the size of the code stream.

n is the total size of all programs (capsules or fraglets) used in the compression process. Only opcodes are counted. In the estimation 5.4.10 we ignore the constant number of extra selection steps and access to the dictionary. The equation 5.4.10 can be easily understood if our compression is considered as a form of sorting algorithm. Then it is similar to the "bubble sort" which has quadratic complexity.

5.4.11 Reference Compression Methods

In order to compare how our compression scheme performs in respect to the existing solutions we have chosen, as a reference, the following classic compression methods:

- *transform-based* encoding: *run-length* coding (see Section 4.2.2), and
- *dictionary-based* encoding: *LZW* (see Section 4.2.4).

We do not consider entropy encoding, i.e., *Huffman*-coding from Section 4.2.5, because it does not actually reduce the number of symbols. Rather it offers an optimal bit-level encoding of the symbols based on the frequency of their appearance in the stream. In fact, *Huffman*-coding

Parameter	Notation	Value	Remarks
MAXPROGSIZE	PS	100	max program size (continuous stream, no fragmentation)
MINPROGSIZE		2	min program size (continuous stream, no fragmentation)
FRAGNUM	FN	20	number of fraglets to generate
MAXFRAGLEN	FL	20	max fraglet's length
MINFRAGLEN		2	min fraglet's length
CURINSTRNUM	AS	40	number of instructions currently in use; typical ISA size for most embedded VM
FREEINSTRNUM	FCS	24	number of free codes for new instructions; should be at least half the size of ISA
SAMPNUM	—	10	number of samples for each measure; reduces uncertainty introduced by randomness
REPRATE	—	0/1/2/3	fragment duplication rate: 0 - fully random, 1 - 100%, 2 - 50%, 3 - 33%, etc.

Table 5.1: Online Code Compression: Simulation Test Setting for Random Code Streams

can be used on top of the online compression for better utilization of binary representation.

5.4.12 Test Setting For Random Code Streams

In the following simulations we model the input code stream using the settings listed in Table 5.1.

As can be seen, we distinguish between “continuous” and “fragmented” code streams. Although there are no ideal infinite code streams and every stream ends at some point some of them, like fraglets, introduce an extra fragmentation degree which we have to take into account. Continuous streams are defined by one parameter, program size (PS), whereas fragmented streams by two, number of fragments (FN) and length of fragments (FL), as discussed in Section 5.4.8.

We reduce the effect which might be introduced by a random code generation process by using averaging over multiple samples (see parameter *SAMPNUM* in Table 5.1).

Additionally, we use a pattern generation technique which allows us to regulate the number of duplicates (instruction sequences) in the code stream. The *REPRATE* parameter basically defines the duplication rate as a percentage of the original code size.

5. Online Code Compression Framework

Variable Parameter	Function	Stream Type	Reference
Program Size (PS)	Compression Factor (CF)	Continuous	Figure 5.11, page 151
Alphabet Size (AS)		Continuous	Figure 5.12, page 152
Free Code Space (FCS)		Continuous	Figure 5.13, page 153
Number of Fragments (FN)		Fragmented	Figure 5.14, page 154
Length of Fragments (FL)		Fragmented	Figure 5.15, page 155
Alphabet Size (AS)		Fragmented	Figure 5.16, page 156
Free Code Space (FCS)		Fragmented	Figure 5.17, page 157

Table 5.2: Single-Node Compression Model: Simulation Overview

5.5 Single-Node Compression Method

We start analyzing different compression models with a very basic one which is the core for all the advanced types discussed further. It is a single-node compression. The model is shown in Figure 5.10.

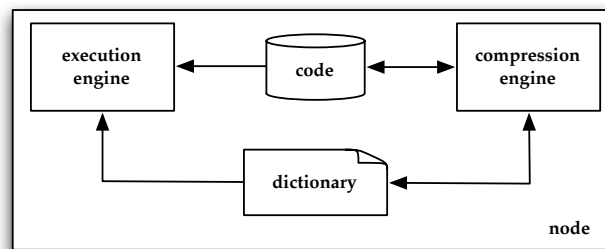


Figure 5.10: Online Compression Scheme: Single-Node Model

In this model, compression is run locally on an individual node only. No code leaves the node, nor dictionary updates are sent out. Technically, it works as it has been discussed in Section 5.4.

Table 5.2 gives an overview of all the simulation runs we have done and the dependencies which have been obtained for the single-node compression model.

The following observations can be made based on analysis of the presented graphs (see Figures 5.11 – 5.17):

- In some of the graphs in Figures 5.11 – 5.17 the characteristic for the run-length algorithm goes out of range which means

that the “compressed” code size is bigger than the original one ($CF > 100\%$).

- Online compression outperforms *LZW* even on highly repetitive streams.
- In continuous streams:
 - The duplication rate has no effect on performance with the same program size (see Figure 5.11, page 151). At the same time, bigger streams give better compression factors as more statistics can be collected.
 - The growing alphabet size “breaks” *LZW*, online compression can resist (see Figure 5.12, page 152).
 - FCS gives a huge positive effect on performance whereas *LZW* cannot naturally benefit from it (see Figure 5.13, page 153).
- In fragmented streams:
 - The growing number of fragments affects the online compression in a negative way since the algorithm resets at each new fragment. Nevertheless, even this allows for better results than *LZW* (see Figure 5.14, page 154).
 - The length of fraglets has a “saddle” on its characteristics which depends on the duplication rate. The higher duplication rate the further the saddle is located: $FL = 8, 12, 18, \dots$ (see Figure 5.15, page 155).
 - The growing alphabet size causes online compression to slow down its performance (see Figure 5.16, page 156).
 - FCS again shows a huge gain as with continuous streams (see Figure 5.17, page 157).

5.6 Group Compression

Group compression is a base for the profile-based tasking which has been previously discussed in Section 5.4.6. This model assumes one leader per profile. The leader runs compression and announces dictionary updates to all the nodes in the profile. The driven nodes only update their local dictionary and execute code. In Figure 5.9a there are two profiles controlled by nodes 1 and 7. They are the leaders. The more common group model structure is shown in Figure 5.18.

5. Online Code Compression Framework

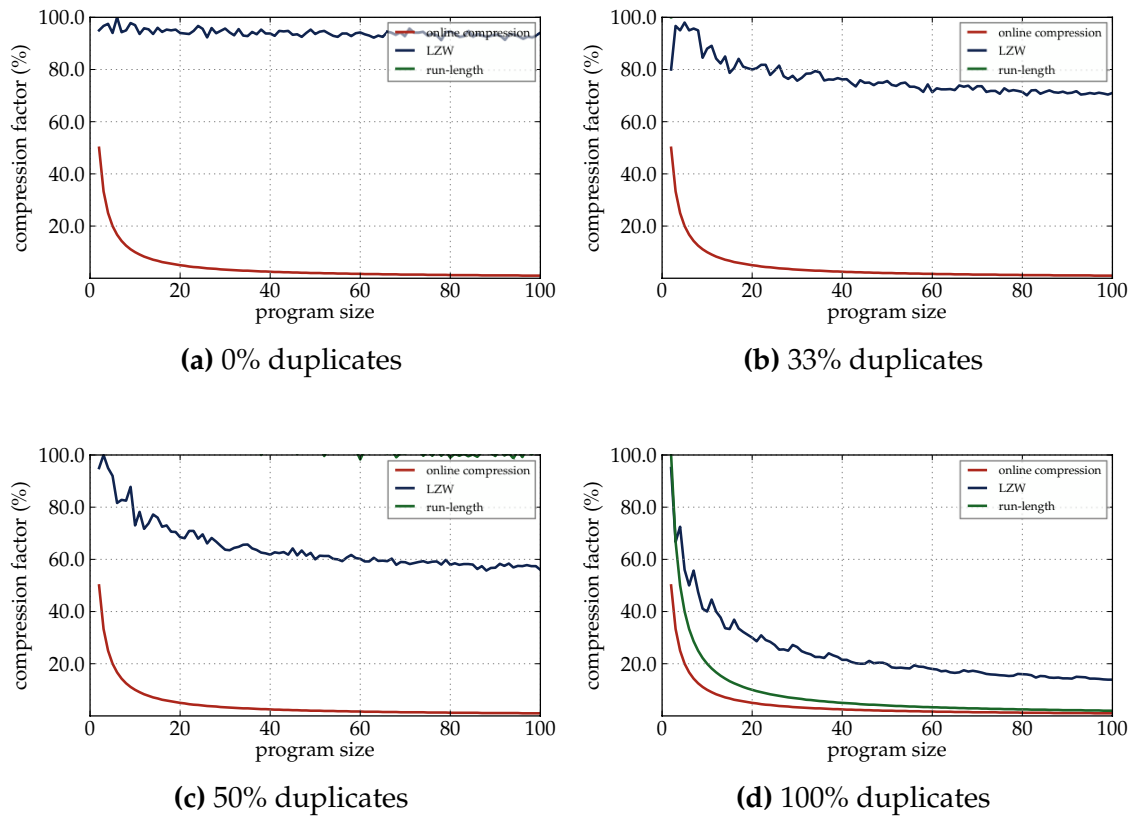


Figure 5.11: Single-Node Compression Model, Continuous Stream: CF vs PS

5.6 Group Compression

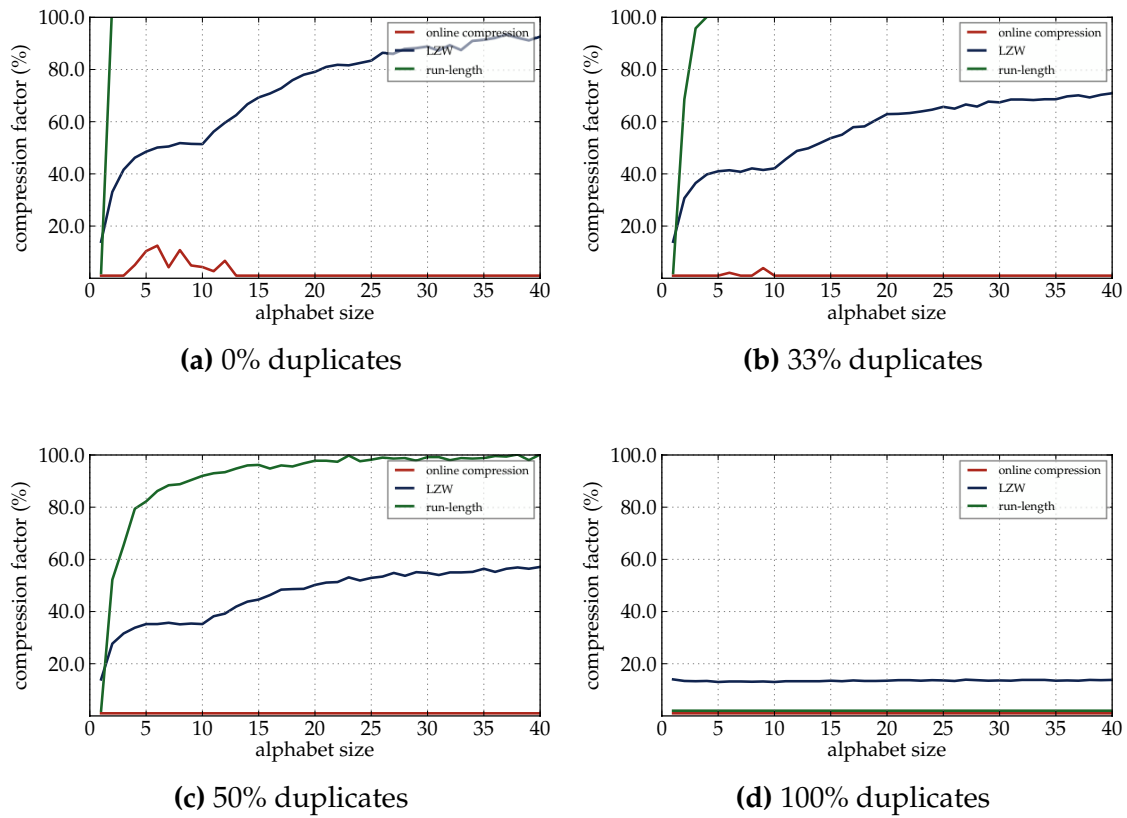


Figure 5.12: Single-Node Compression Model, Continuous Stream: CF vs AS

5. Online Code Compression Framework

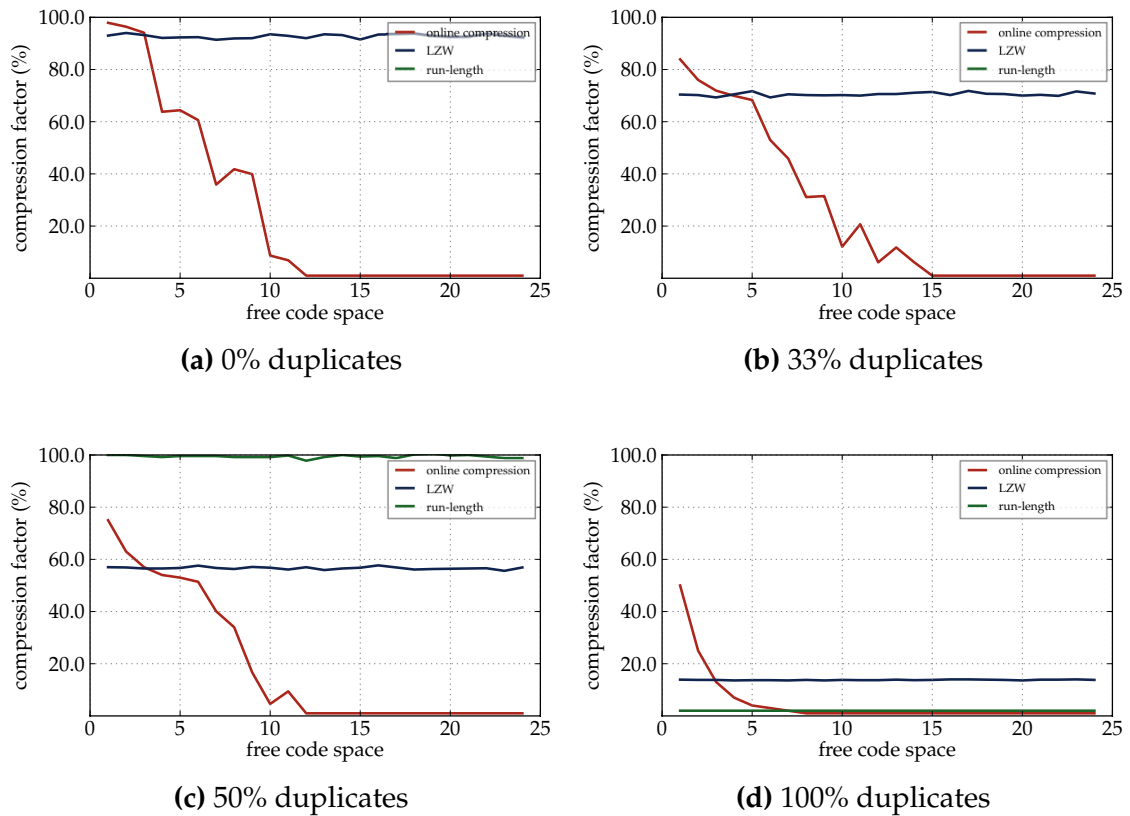


Figure 5.13: Single-Node Compression Model, Continuous Stream: CF vs FCS

5.6 Group Compression

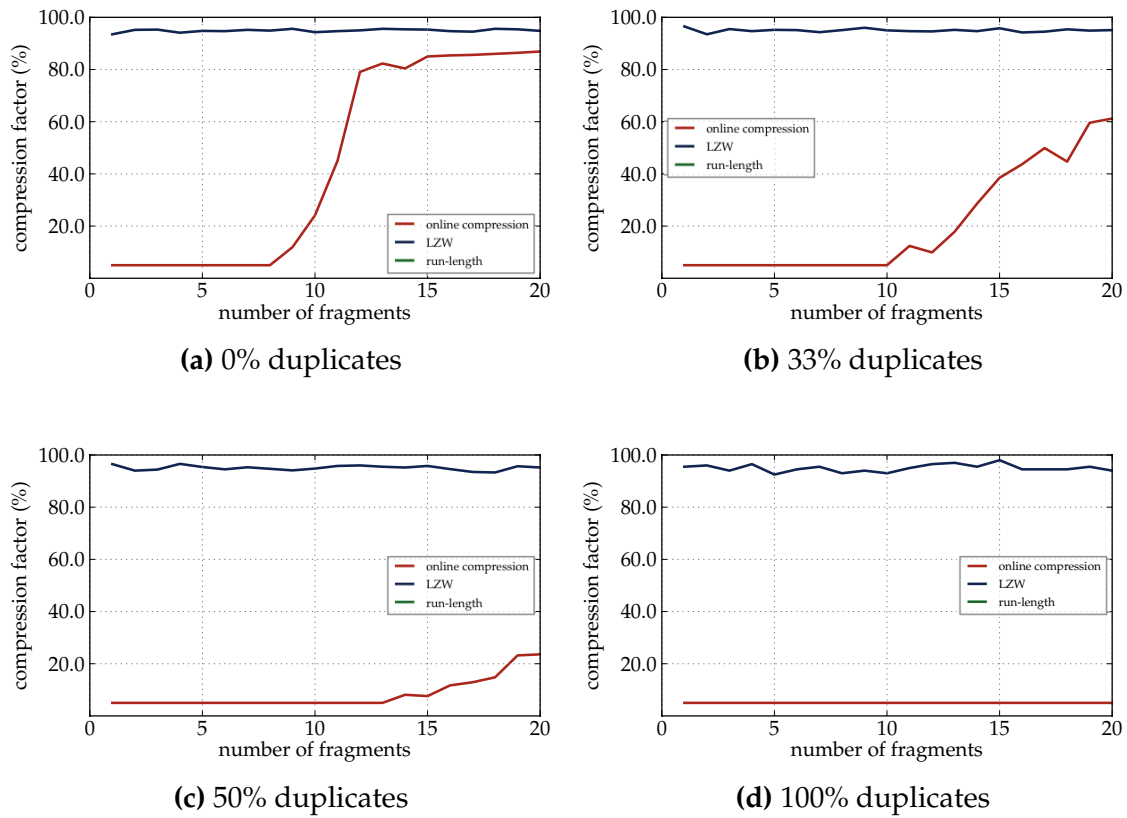


Figure 5.14: Single-Node Compression Model, Fragmented Stream: CF vs FN

5. Online Code Compression Framework

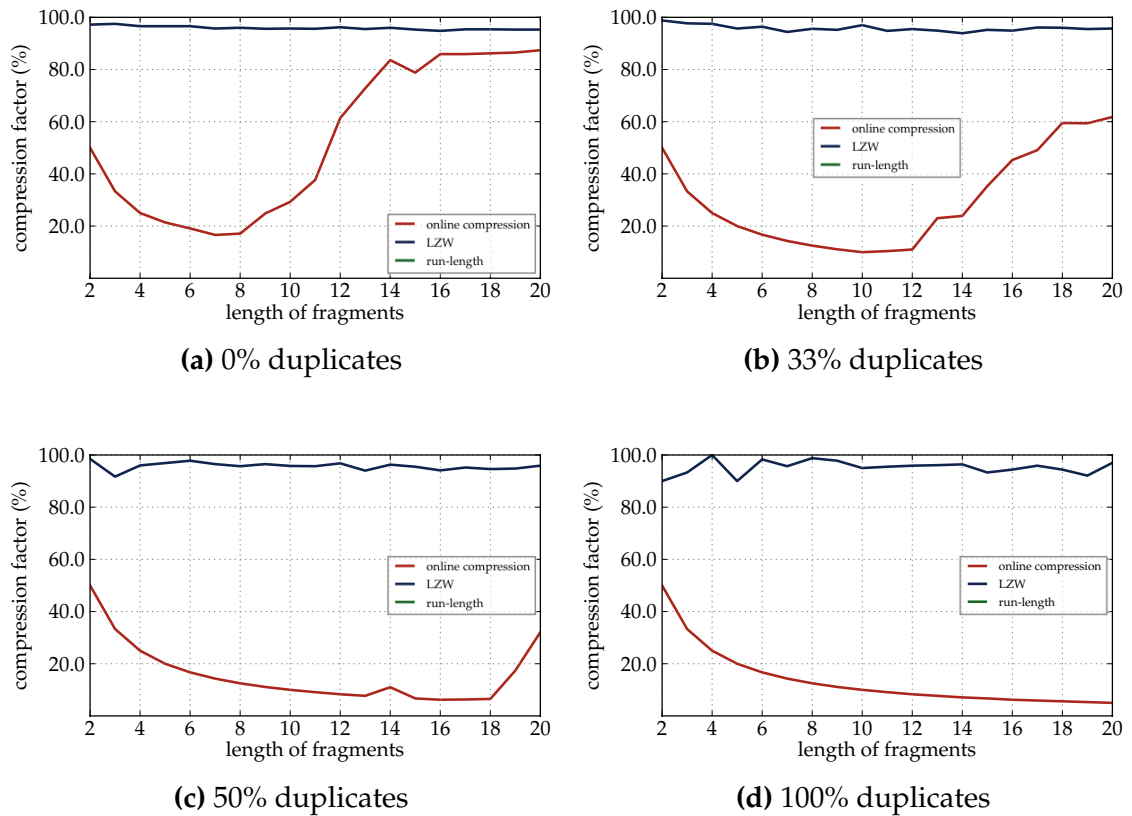


Figure 5.15: Single-Node Compression Model, Fragmented Stream: CF vs FL

5.6 Group Compression

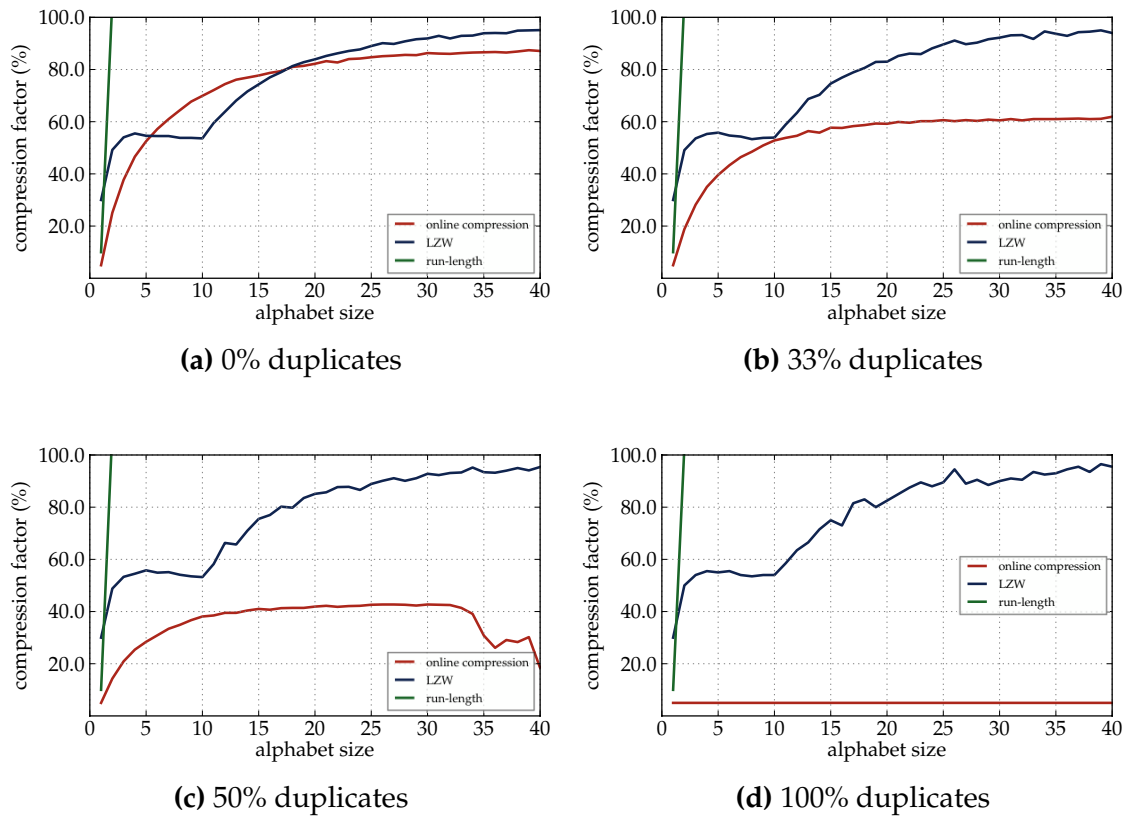


Figure 5.16: Single-Node Compression Model, Fragmented Stream: CF vs AS

5. Online Code Compression Framework

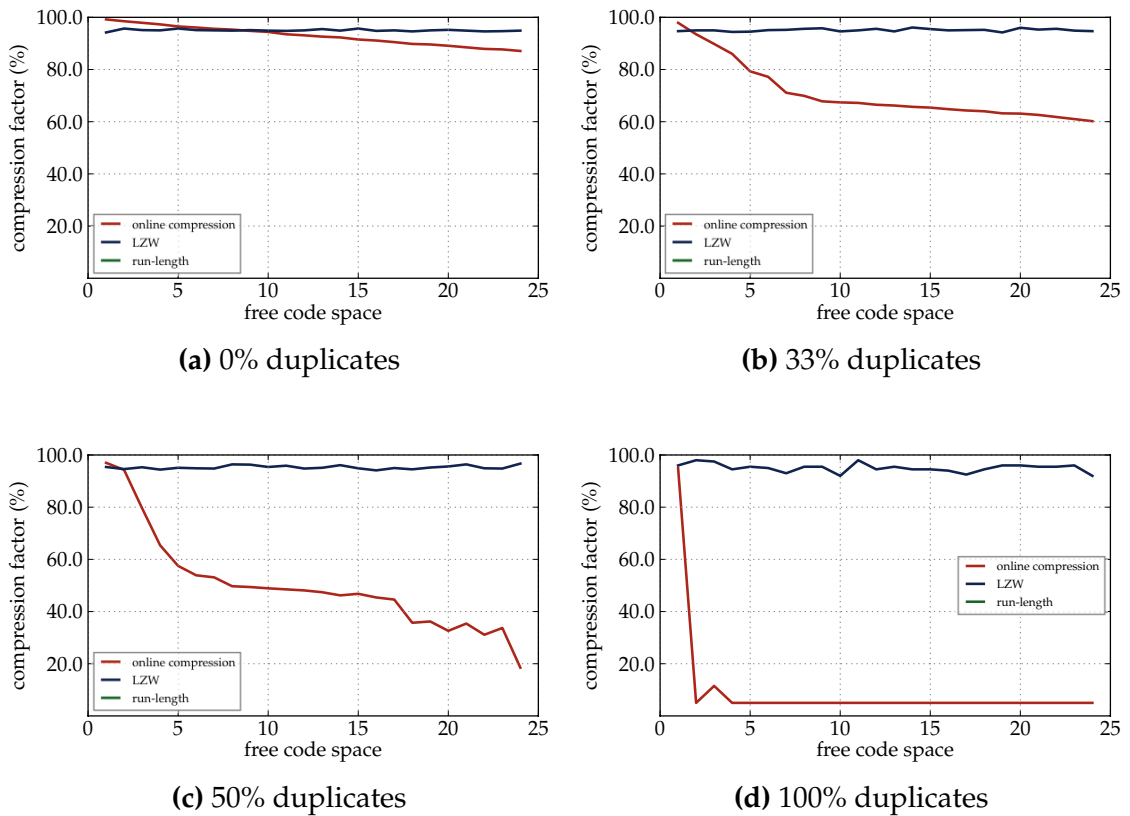


Figure 5.17: Single-Node Compression Model, Fragmented Stream: CF vs FCS

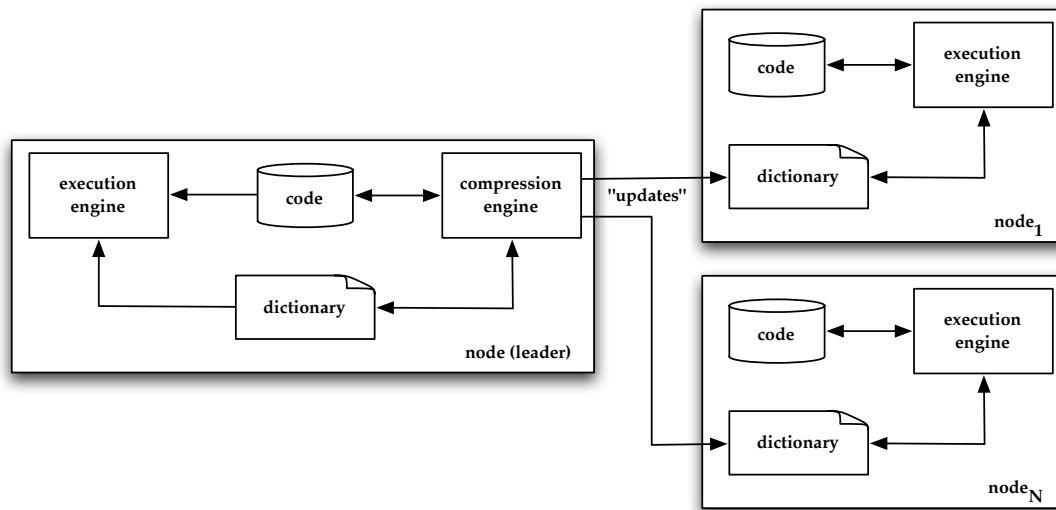


Figure 5.18: Online Compression Scheme: Group Model

In principle, every node can become a leader in this model if it belongs to the corresponding profile. Generally, the election of a leader is not a trivial task. For the moment, we use a manual leader assignment. Dictionary updates are disseminated using a viral propagation method.

5.7 Distributed Compression

Distributed compression is actually the most sophisticated and complicated model of all considered. The following types of distributed compression can be distinguished:

- All nodes hold the same code, different nodes can make a decision on the next compression step at different moments in time. Every node features the architecture shown in Figure 5.10. To announce the decision the node has to send a dictionary update to the others. In our leader-based approach (see Section 5.6) we simplify this scenario by electing a single leader for the entire profile. By doing so, we achieve less transmission load and require less coordination between nodes.
- Nodes hold different code. In this case, nodes have to agree on a common optimal representation for the code chunk which resides remotely. In this work, we do not consider this type of compression as it goes against the whole concept of profiles and

5. Online Code Compression Framework

VS, i.e., grouping nodes having the same code to execute together. We discuss how this could possibly be done in Chapter 7.

- Per-link compression. This type can potentially find a use with fraglets. In this case, a separate dictionary is created on a per-link basis between every two nodes. This would allow achievement of much better compression rates on each particular link rather than on average, which might be beneficial for communication-intensive methods like CNP. In fact, this can also be considered as a special case of the group compression (see Section 5.6) for 2 nodes.¹ In this case, one node becomes a leader for the pair.

5.8 Cloud Compression

Cloud compression is an extended version of a group compression (see Section 5.6) with multiple possibly overlapping profiles (clouds). This model is the closest of all to the real setting. We simulate it by using a so-called “network map” file which describes network structure in terms of clouds, inter-cloud communication and code size. We consider two examples: a very general cloud setting (see Section 5.8.1) and a quasi-real setting (see Section 5.8.2).

5.8.1 General Setting

For the general setting we use the map file shown in Listing 5.1. This setting gives no restrictions on random code generation process. This is the main difference from the quasi-setting.

1	# cloud	parent	nodes	interlocutor	nodes	nodes	in-cloud	inter-cloud
2	# name	cloud	in the		speaking	speaking	code size	code size
3	#		sub-cloud		forward	back	(size/frags)	(size/frags)
4	A1	A	5	S	2	1	50/5	5/1
5	A2	A	1	S	1	1	50/5	4/1
6	S	S	4	A1	1	2	100/7	10/2
7	B	B	3	S	1	1	70/10	5/1
8								
9	# this must be the last record in the file							
10	all	all	–	–	–	–	500/30	–

Listing 5.1: General Cloud Model Setting Map File

¹This is true if only both nodes hold the same code. If code is different we revert back to the previous case.

This file, for example, may describe the network like the one shown in Figure 5.19.¹

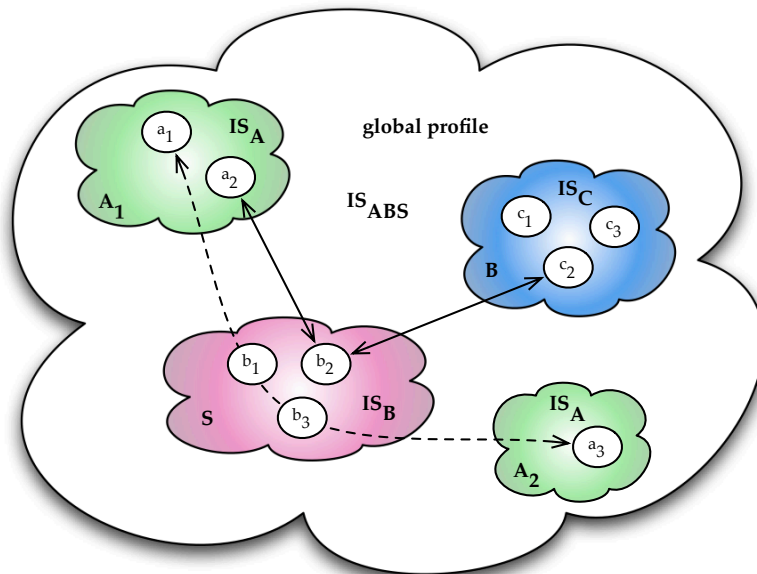


Figure 5.19: Online Compression Scheme: Example of Profile-Oriented Network

Using this map-file we generate code for the entire network (all profiles on all nodes) and fed it to the compression engine.

The results for continuous and fragmented streams are shown in Figures 5.20, page 161 and 5.21, page 162, respectively. As with single-node compression (see Section 5.5), bigger continuous code streams allow us to collect more statistics on them and, therefore, to provide better compression. Alphabet size has very little or no effect. FCS still remains the most influential parameter. This can be easily explained as giving more freedom to the algorithm we allow it to find a better solution (see Figure 5.20, page 161).

With fragmented streams the characteristics remain very similar to those discovered for continuous ones. Growing fragmentation level, length of fragments (with a “saddle” in the middle of the curve) and

¹The general structure is correct, although the number of nodes in the picture 5.19 is different from what is specified in the map-file of Listing 5.1.

5. Online Code Compression Framework

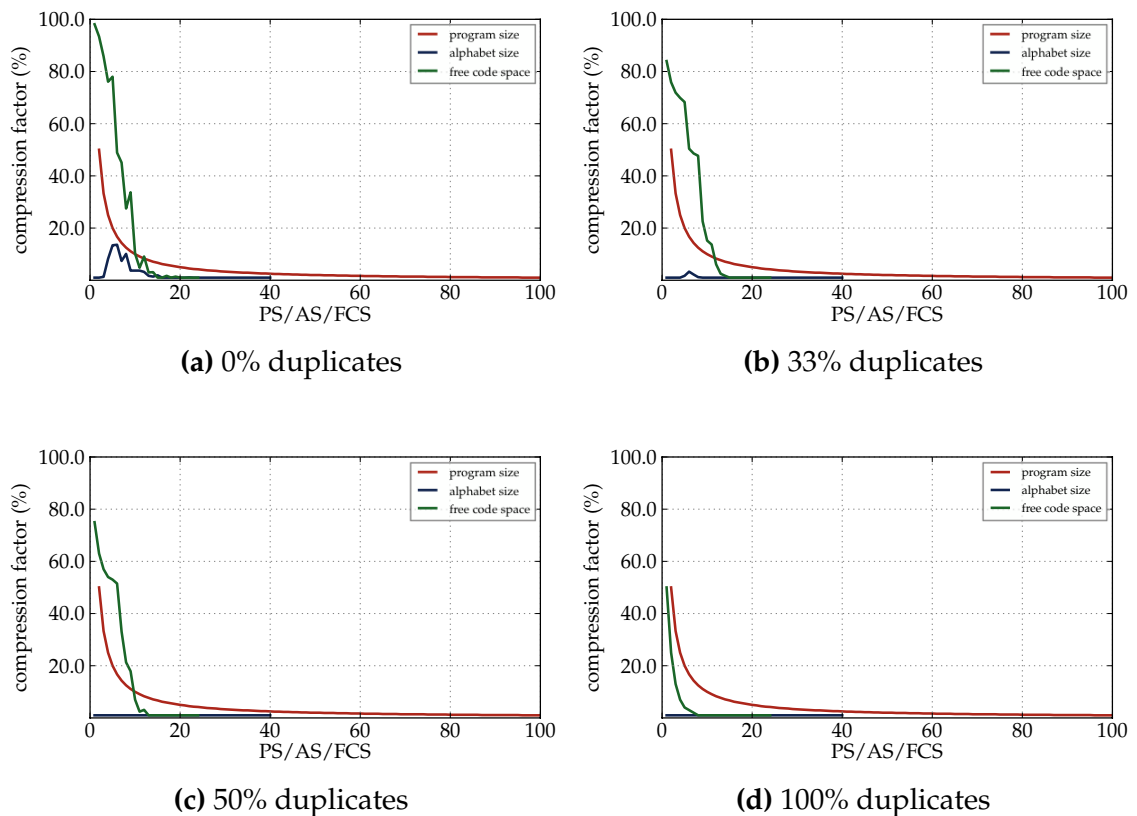


Figure 5.20: Cloud Compression Model: Continuous Stream

alphabet size have a negative impact on the compression factor whereas larger FCS enhances it (see Figure 5.21, page 162).

5.8.2 Quasi-Real Setting

The quasi-real setting is different from the general one in the number of constraints applied. The quasi-real setting features continuous (non-fragmented) stream only, pre-defined code size distribution, fixed initial number of needed instructions (40) and variable free code space size (24).

As FCS seems to be the most interesting parameter, for this test we only show how system performance depends on it. The results are presented in Figure 5.22, page 163.

As we can see in the quasi-real setting the compression factor can be up to 20% for continuous streams and up to 23% for fragmented streams. The convergence speed is very fast at the beginning and slows

5.8 Cloud Compression

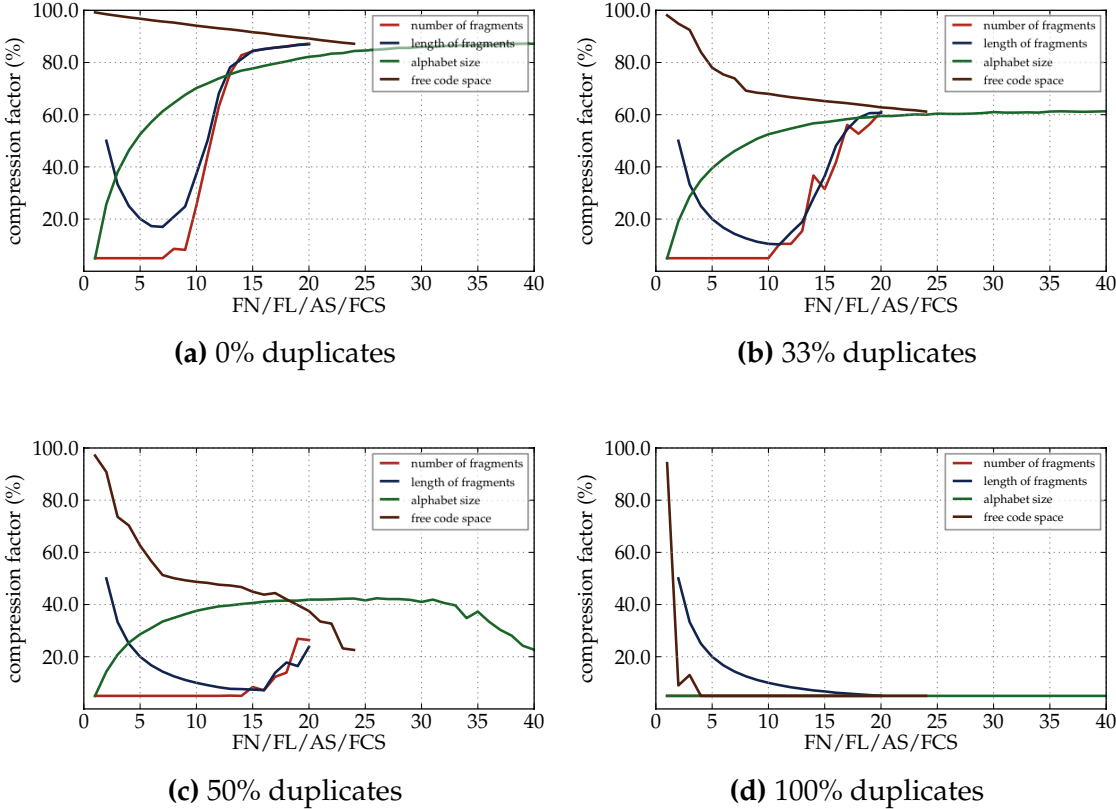


Figure 5.21: Cloud Compression Model: Fragmented Stream

5. Online Code Compression Framework

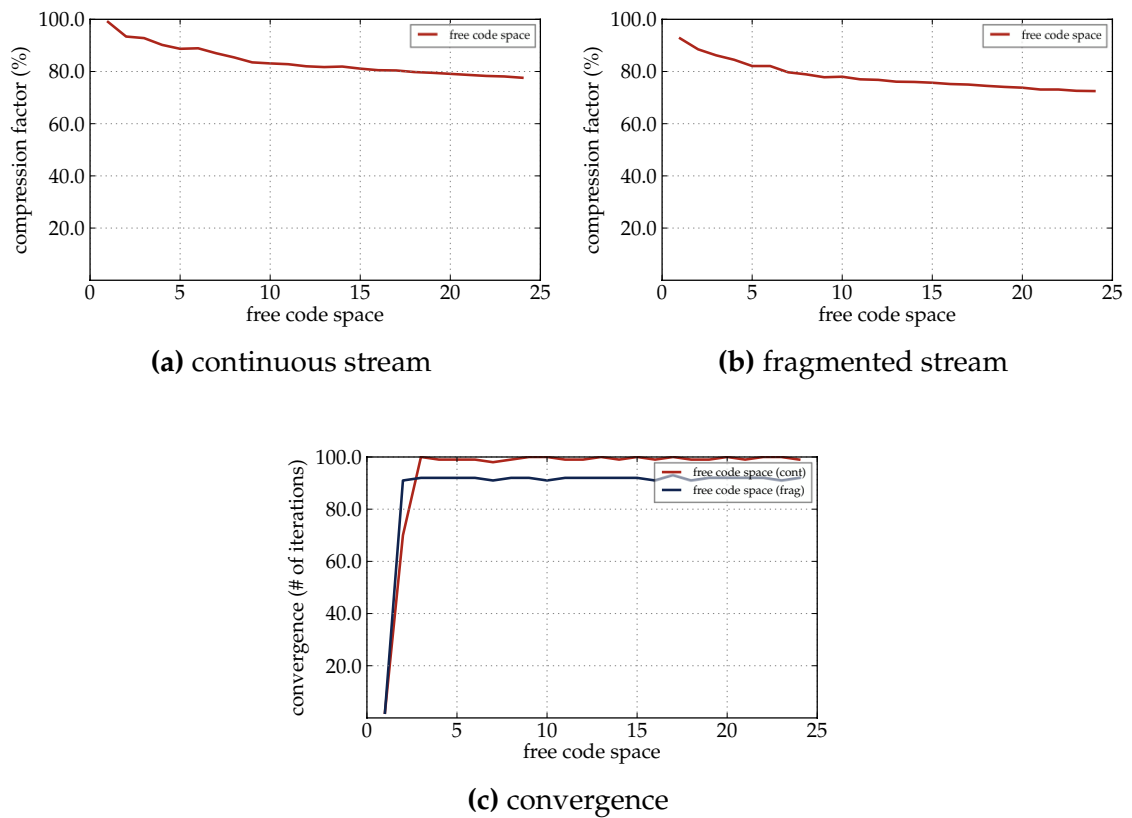


Figure 5.22: Cloud Compression Model: Quasi-Real Setting

Variable Parameter	Function	Model	Stream Type	Reference
Program Size (PS)	Convergence Speed (# number of iterations)	Single-node	Continuous	Figure 5.23, page 165
Number of Fragments (FN)		Single-node	Fragmented	Figure 5.24, page 166
Length of Fragments (FL)		Single-node	Fragmented	Figure 5.25, page 167
Alphabet Size (AS)		Single-node	Continuous and fragmented	Figure 5.26, page 168
Free Code Space (FCS)		Single-node	Continuous and fragmented	Figure 5.27, page 169
Program Size (PS)		Cloud	Continuous	Figure 5.28, page 170
Number of Fragments (FN)		Cloud	Fragmented	Figure 5.29, page 171
Length of Fragments (FL)		Cloud	Fragmented	Figure 5.30, page 172
Alphabet Size (AS)		Cloud	Continuous and fragmented	Figure 5.31, page 173
Free Code Space (FCS)		Cloud	Continuous and fragmented	Figure 5.32, page 174

Table 5.3: Convergence Speed: Simulation Overview

down dramatically after some point. This can be explained by the fact that at the beginning the algorithm is capable of finding multiple pairs very easily. Over time as the code shrinks instruction pair distribution becomes more uniform which causes the algorithm to lose speed as appearance frequency drops.

5.9 Convergence Speed

At the end of our analysis we would like to take a more detailed look at the convergence speed of our algorithm. This parameter is very important as it shows how quickly the system can adapt to changes in software configuration.

Table 5.3 gives an overview of all the situations where we have measured the convergence speed.

The following observations can be made based on analysis of the presented graphs (see Figures 5.23 – 5.32):

- In continuous streams the convergence speed is a linear function of the program size (see Figure 5.23, page 165). For fragmented streams it is difficult to establish any analytic link be-

5. Online Code Compression Framework

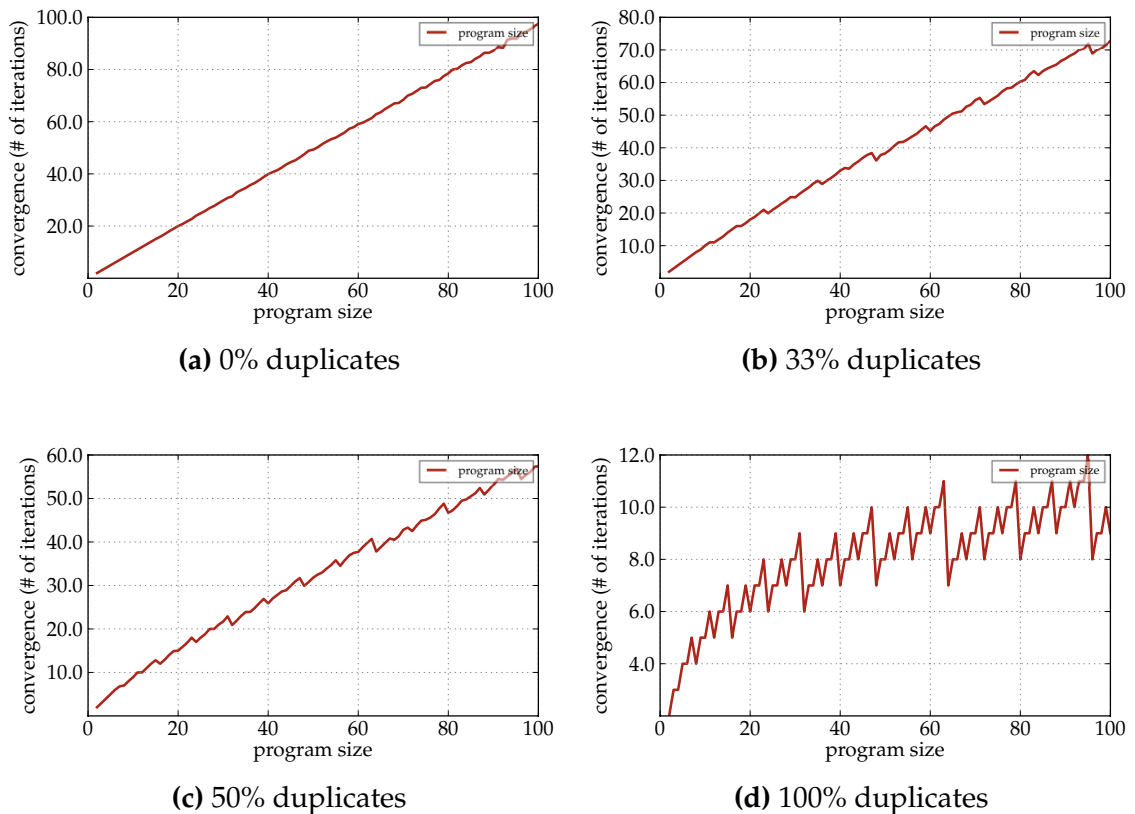


Figure 5.23: Single-Node Compression Model, Continuous Stream: Convergence vs PS

tween code size and convergence (see Figures 5.24, page 166 and 5.25, page 167) but in general it seems there is an optimal number of fragments (8–15) of a particular length (10–18) when the convergence speed reaches maximum.

- Beyond a certain level the alphabet size has no further effect on the convergence speed (see Figure 5.26, page 168).
- The same is true for FCS (see Figure 5.27, page 169).
- Extending the case up to the cloud model does not produce any exceptions nor unexpected anomalies (see Figures 5.28, 5.29, 5.30, 5.31, 5.32, pages 170 – 174).

5.9 Convergence Speed

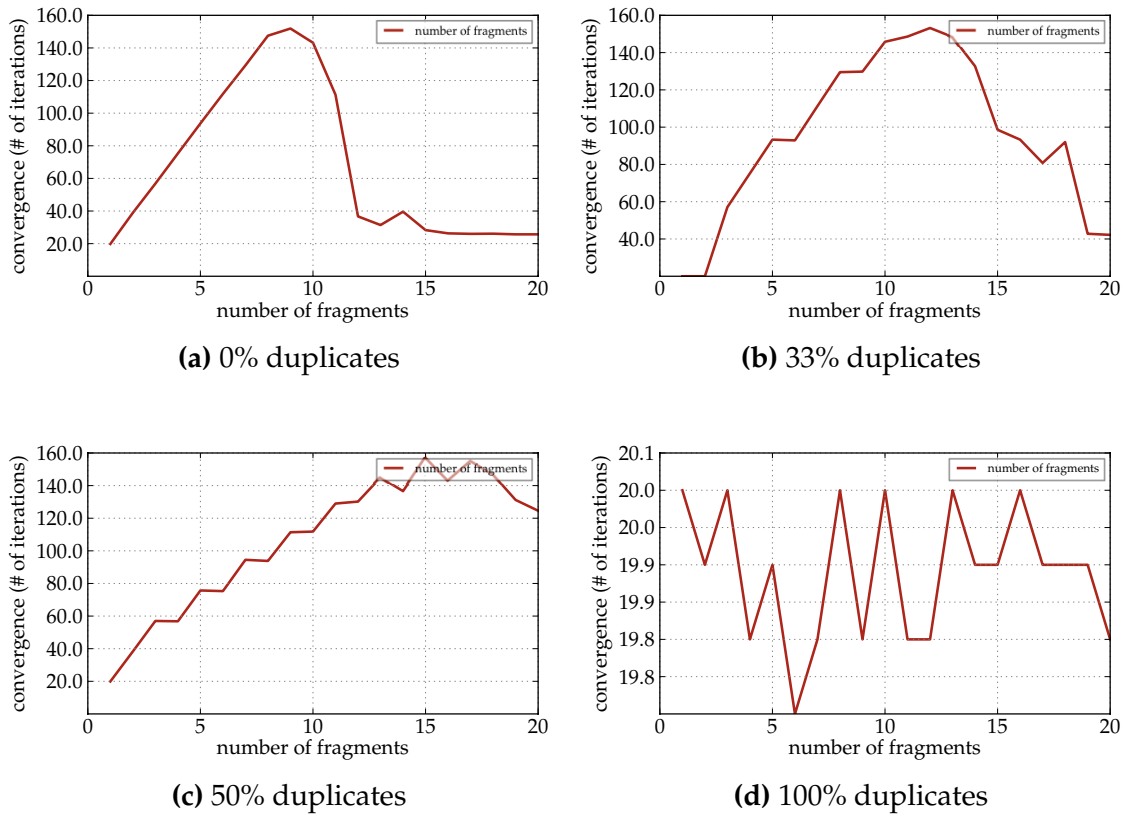


Figure 5.24: Single-Node Compression Model, Fragmented Stream: Convergence vs FN

5. Online Code Compression Framework

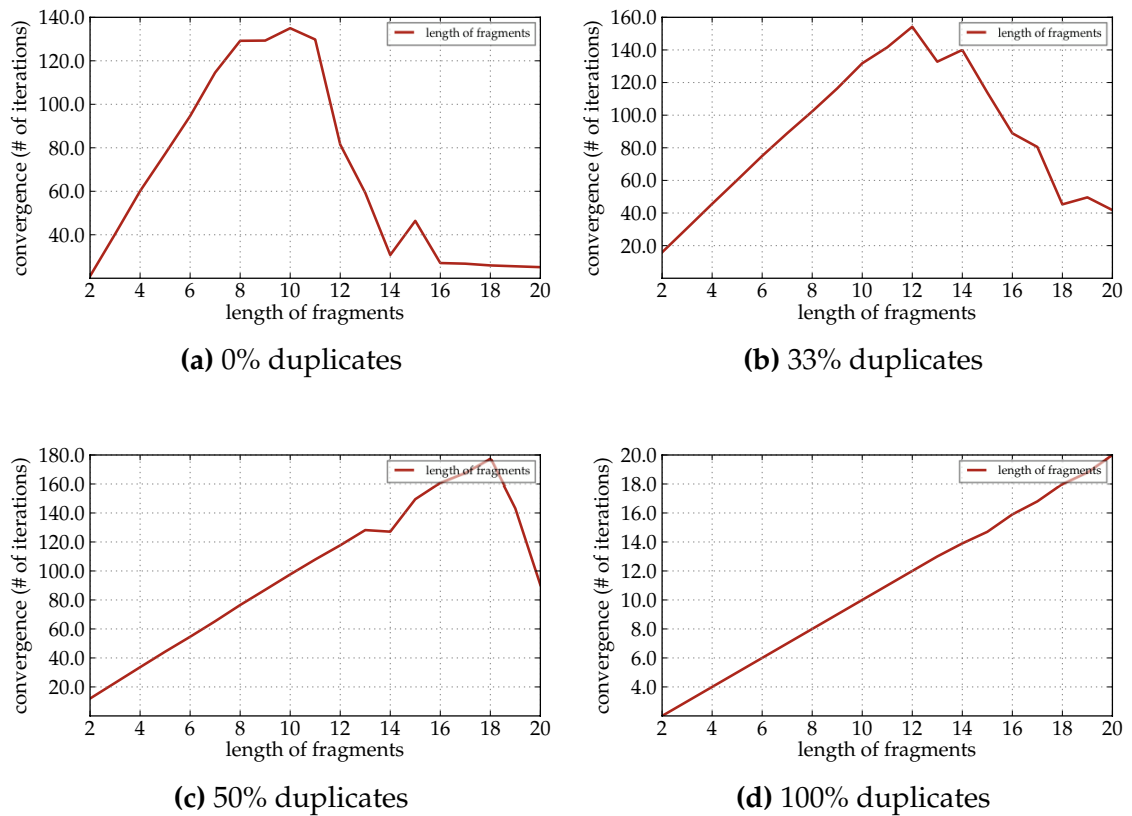
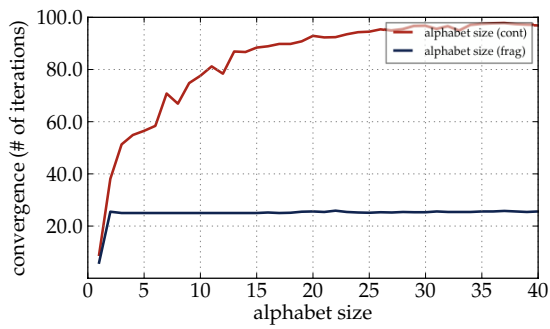
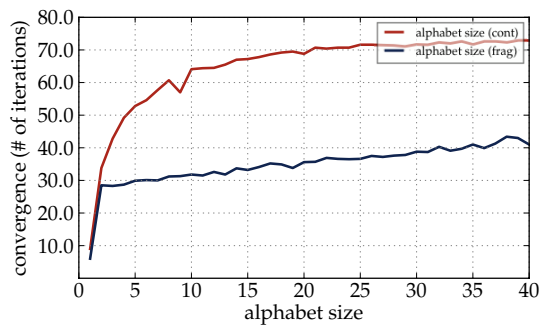


Figure 5.25: Single-Node Compression Model, Fragmented Stream: Convergence vs FL

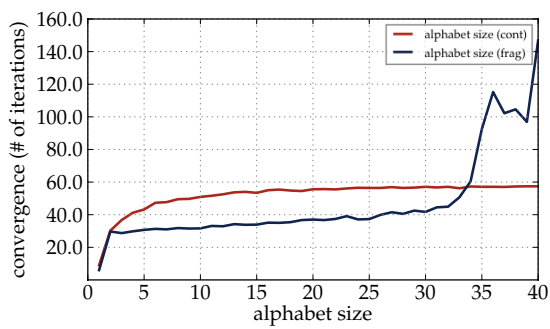
5.9 Convergence Speed



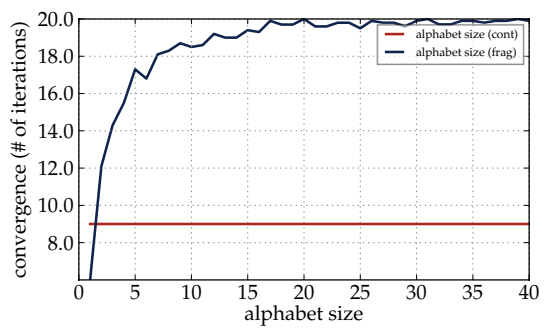
(a) 0% duplicates



(b) 33% duplicates



(c) 50% duplicates



(d) 100% duplicates

Figure 5.26: Single-Node Compression Model: Convergence vs AS

5. Online Code Compression Framework

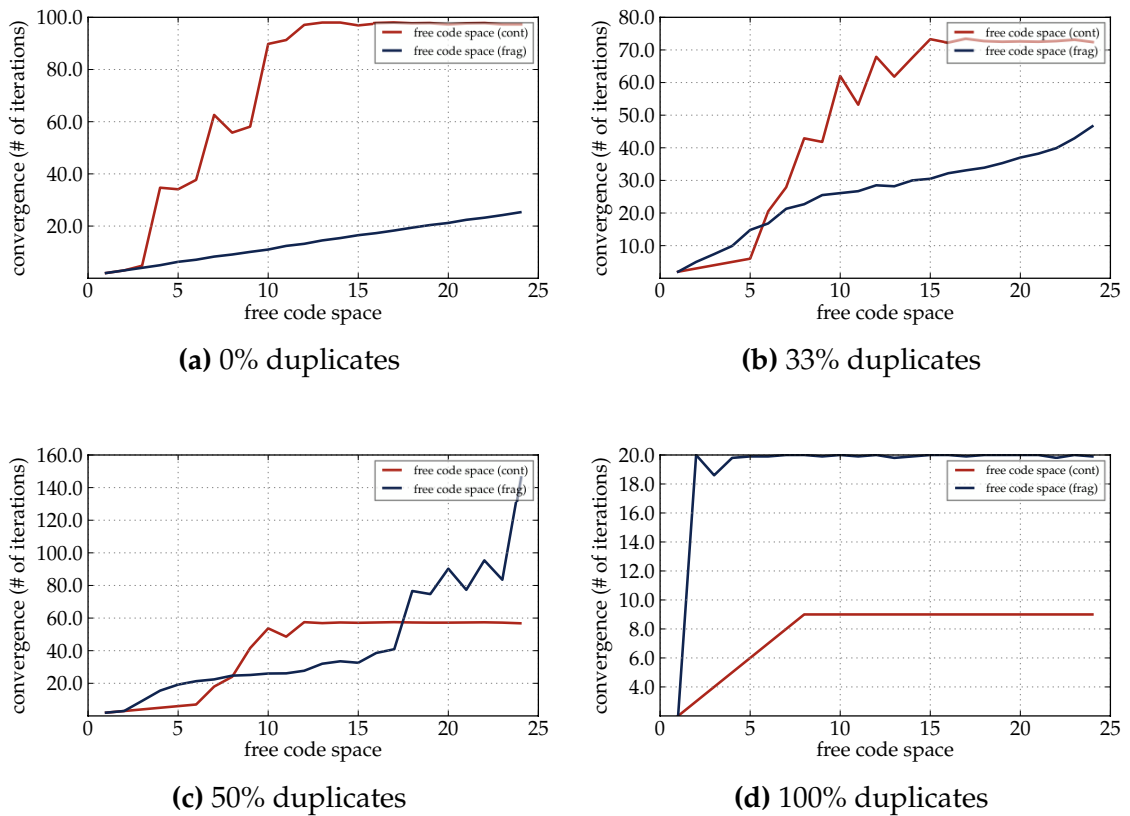


Figure 5.27: Single-Node Compression Model: Convergence vs FCS

5.9 Convergence Speed

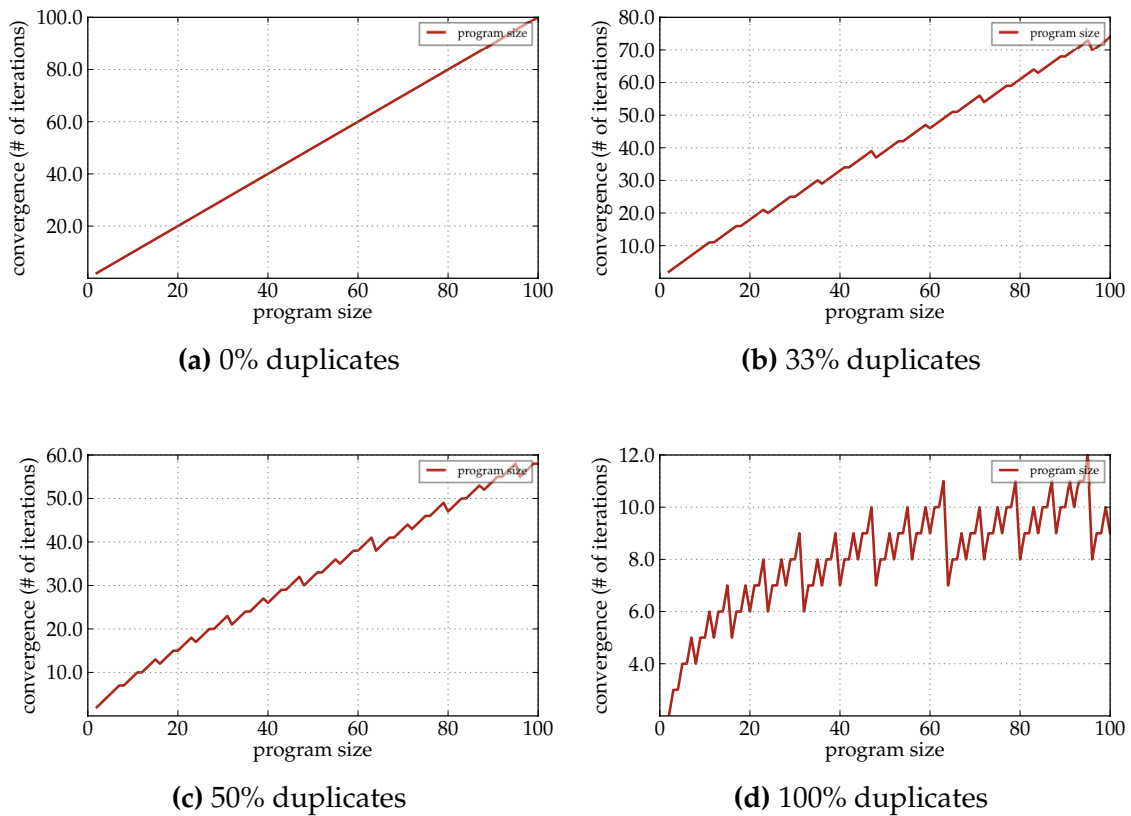


Figure 5.28: Cloud Compression Model, Continuous Stream: Convergence vs PS

5. Online Code Compression Framework

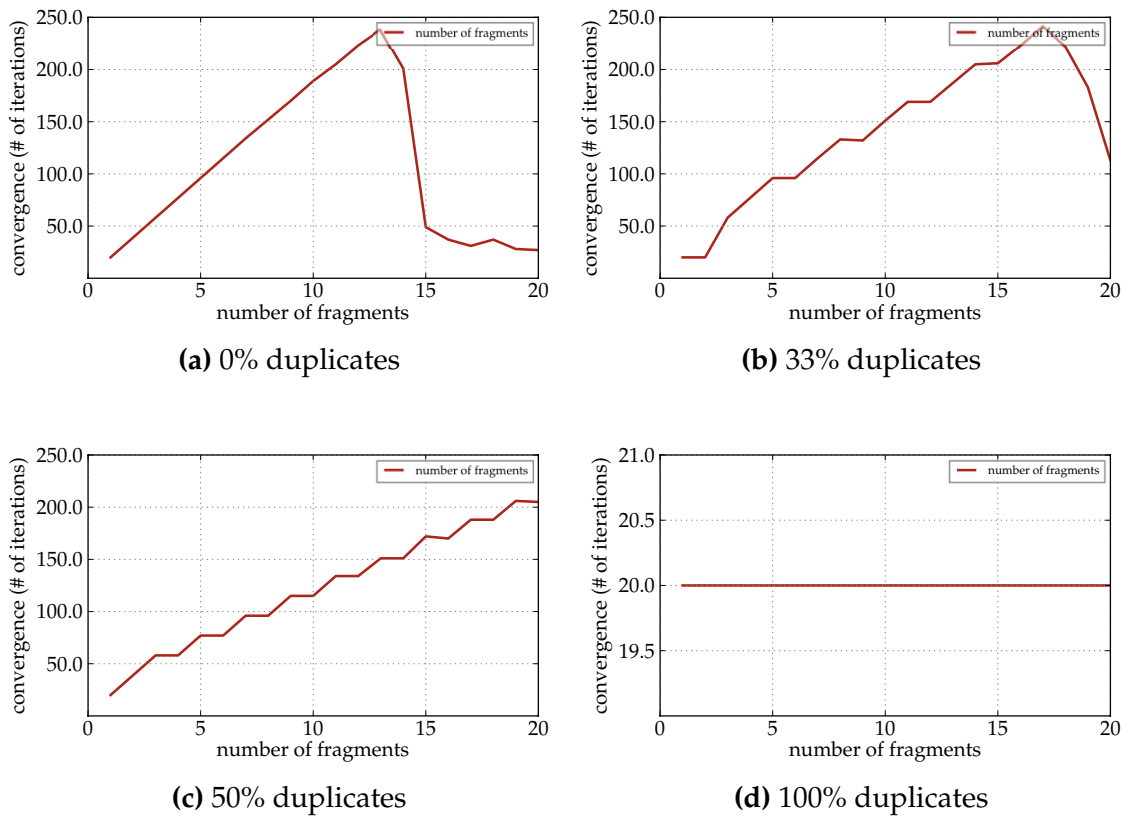
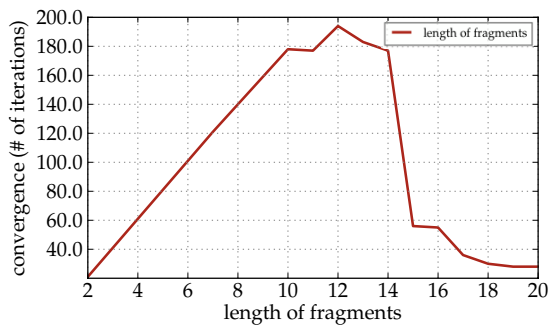
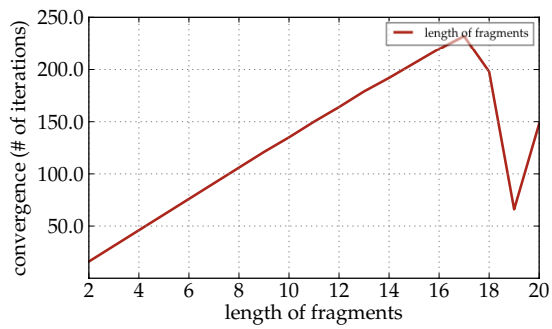


Figure 5.29: Cloud Compression Model, Fragmented Stream: Convergence vs FN

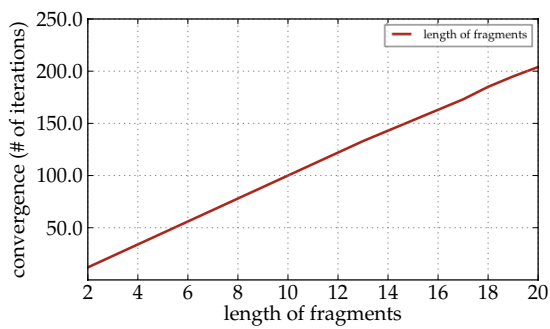
5.9 Convergence Speed



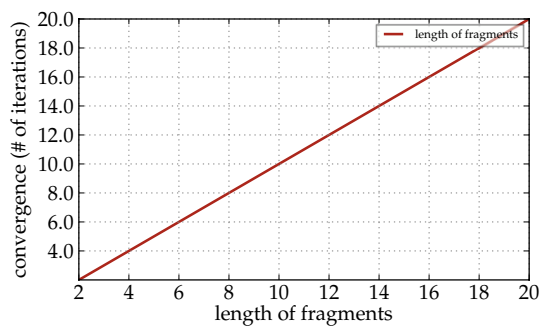
(a) 0% duplicates



(b) 33% duplicates



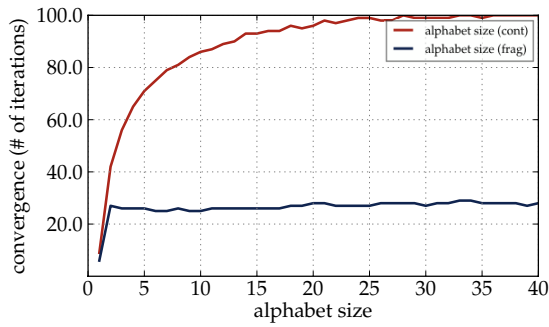
(c) 50% duplicates



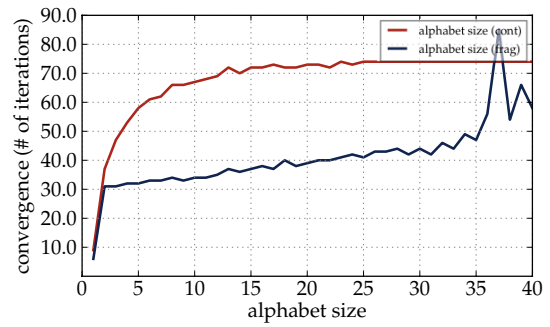
(d) 100% duplicates

Figure 5.30: Cloud Compression Model, Fragmented Stream: Convergence vs FL

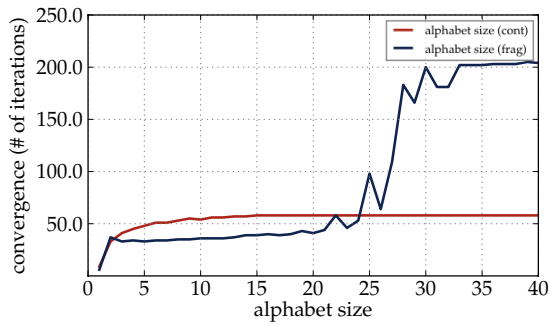
5. Online Code Compression Framework



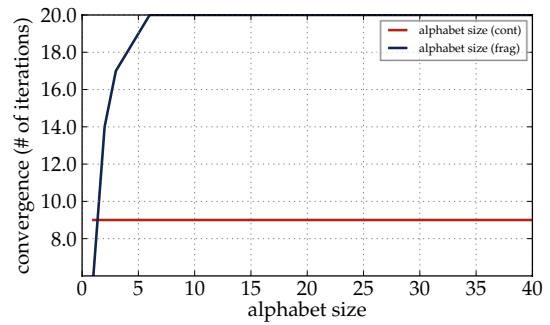
(a) 0% duplicates



(b) 33% duplicates



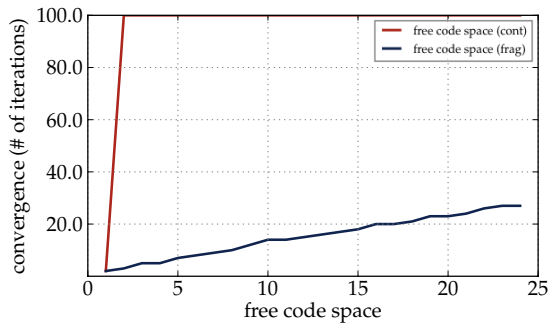
(c) 50% duplicates



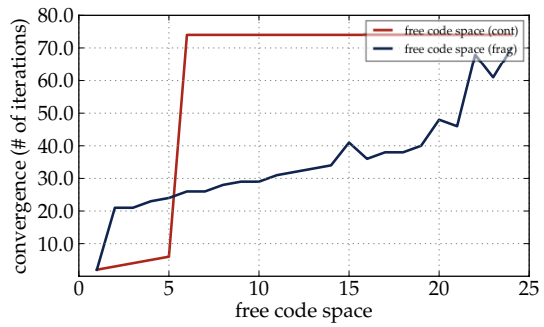
(d) 100% duplicates

Figure 5.31: Cloud Compression Model: Convergence vs AS

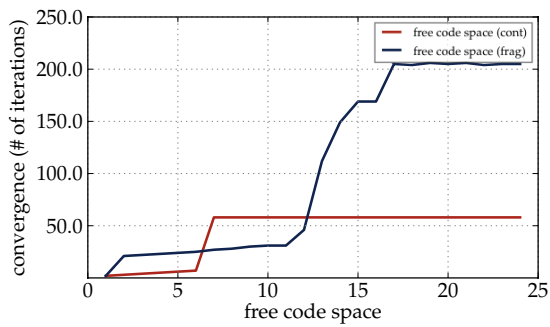
5.9 Convergence Speed



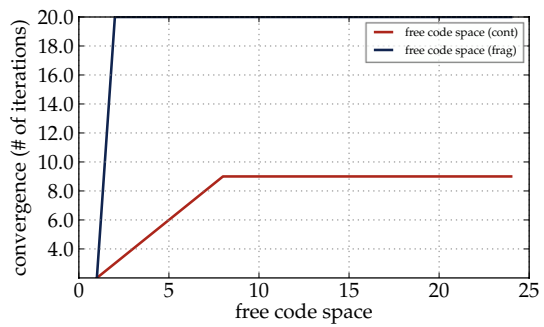
(a) 0% duplicates



(b) 33% duplicates



(c) 50% duplicates



(d) 100% duplicates

Figure 5.32: Cloud Compression Model: Convergence vs FCS

5.10 Summary

Based on the concepts from AIT and KC theories we have described the core part of our work, the run-time code compression method. We have shown that just an attempt to apply an existing data compression method to code streams does not normally give the desired performance. This is due to the different nature of code streams. The new view at code compression allowed us to develop the method which can potentially outperform the existing solutions. After analysis of specific requirements posed by WSN we have given a detailed description of the method and provided a set of simulation results on how the method behaves in various configurations scenarios. The relatively small computational complexity of the method allows it to be integrated into resource-constrained devices like WSN nodes.

6

Experimental Setup

Having introduced the tools, ChameleonVM and FragletVM, in Chapter 3, code optimization methodology in Chapter 4 and our contemporary code compression scheme in Chapter 5, we will now show how all these can be used to build task-specific network protocols. We present a set of widely needed protocols such as a spanning tree builder, route discovery, node ID assignment, time synchronization and others. In these examples we use ChameleonVM as a programming environment. For generalization, we take a look at a real-world scenario of building and optimizing a data collection WSN application. We show that by adding new functional blocks on the fly and applying our compression scheme to optimize code representation we achieve a certain level of power consumption cuts. Additionally, we consider an example for FragletVM, the chemical aggregation protocol. In order to demonstrate how our method can be used in foreign systems we pick, compress and analyze a third-party code, the fire-tracking application using Agilla's mobile agents.

6.1 Overview

In order to demonstrate how our code optimization techniques (see Chapters 4 and 5) can be used in practice we consider a number of examples which traditionally form a part in many real-world network embedded applications. These examples include topology-related tasks (building a spanning tree on a set of nodes, route discovery between two nodes in the network), time synchronization which is a crucial part of most WSN systems, utility functions (count the number of nodes in the network, carry out ID (re-)assignment). We perform these experiments on *ChameleonVM* code (see Section 6.4). In each case we show what impact every optimization step (code shrinking, code compression, code polymorphism and code versioning) has on the program code size and energy saving. Finally, we build a data collection application based on the traditional WSN architectural approach where nodes are formed in a tree with the top at the so-called **sink** node. Measurements are periodically taken and forwarded by each node towards the sink where they are accumulated in a buffer. Further processing may include putting this data in an online database like it is done with *PermaSense* samples (see Section 1.4). From the programming point of view we use the developed practice of adding new functional blocks on the fly in a form of capsules. When the code is deployed we enable our compression scheme and keep a track of how the code representation changes over time. Later we estimate the potential long-term energy cut caused by reduction in code size.

Capsules and mobile code are an alternative to packet-switching networking. Still, applications based on mobile code normally reflect the traditional “request-reply” paradigm, though now not only static data fields are exchanged but also code. To develop this further we look at the CNP (see Section 2.4), a concept which introduces highly fragmented and extremely intensive communication. We pick the protocol called *Disperser* which we have re-implemented for *FragletVM* (see Section 3.3). It is a gossip-like aggregation protocol where each value (all summands and the average) represented by a concentration of small packets (molecules). By balancing the number of those packets on all the nodes an average can be computed. We show that chemical networking can also benefit from code compression as it employs a lot of code transmissions (see Section 6.5).

Characteristic	Capsule (ChameleonVM)	Fraglet (FragletVM)	Mobile Agent (Agilla)
Data segment	size of BUFC (constants and immediate stack operands)	none	immediate stack operands and current heap's content
Instructions	total number of instructions in the program code		
Code resolution	$\log_2(\text{number of instructions})$		
Code segment	only code, data is excluded	everything	only code, data is excluded
System info	header, segment separators, etc.	none	header

In the end (see Section 6.6), we show that our compression scheme can be adapted for different types of mobile code. For this, we pick a fire-tracking application from *Agilla*'s distribution. The application uses an idea of migrating agents, which create a perimeter around the place on fire. It turns out that *Agilla*'s code is perfectly suitable for our method and shows a good performance.

In terms of fragmentation and transmission rates the three code streams above can be classified as follows:

- capsules (low fragmentation, low rates),
- fraglets (high fragmentation, high rate), and
- mobile agents (no fragmentation, very low rates).

In Table 6.1 we list all the applications analyzed later in the chapter along with a set of their code characteristics. The corresponding fields in Table 6.1 must be read as follows:

Note that *Agilla* implements code migration in several steps: stack, code, heap, state. In our code size estimations we include the code exchange phase only. *Agilla* code migration is further discussed in Section 6.6.

In all the cases we do not refer to the actual packet size, which may be different on different platforms, but to the packet payload size.

6.2 Test Settings

In all, except one, of our tests we assume there is an underlying MAC-layer provided by OS, which is responsible for resolving various collisions in transmissions. Nevertheless, in 6.4.6 we present our own

6. Experimental Setup

Application	Data Segment [bytes]	Instructions	Code Resolution [bits]	Code Segment [bytes]	Code Segment + System Info [bytes]	Total [bytes]
<i>ChameleonVM</i>						
"Hello World!"	5	8	2	2.0	7.0	12.0
Route discovery	27	43	3	16.125	21.125	48.125
Spanning tree	25	47	4	23.5	28.5	53.5
Network size estimator	6	10	2	2.5	6.5	12.5
ID re-assignment	22	45	4	22.5	26.5	48.5
SBTSP	71	151	4	75.5	80.5	151.5
Data collection application	136	273	5	170.625	180.625	316.625
<i>FragletVM</i>						
Disperser	0	16	3	6.0	6.0	6.0
<i>Agilla</i>						
Fire Tracker	43	78	5	48.75	52.75	95.75

Table 6.1: Characteristics of Test Applications: Program Size

TDMA-like time synchronization scheme which can be used as a simple MAC-layer. Although the number of nodes in the network remains constant, we explore different topologies as described in Section 6.3.

In Section 5.4.8 we discussed various parameters which can be tuned in our compression scheme. As the program size (including the number of fragments) and the alphabet size are fixed for each particular application, the only parameter we tweak in the tests is the **Free Code Space (FCS)**. We allow FCS to grow up to two times of the original alphabet size.

In order to make estimations on energy spending we have incorporated two radio chip models in our design (see Section 6.3). Therefore, a long term energy assessment is made by extrapolating the values obtained within a short period (e.g., a cycle).¹ The proposed model is relatively rough, as it does not include power consumption for the rest of the module (only the radio chip itself), nor any extra spending on communication, e.g., on re-transmissions.

6.2.1 Code Pre-Processing

Below, in Sections 6.4 – 6.6 for our experiments we use the code of three different types. To represent the code we use a readable, source code form. However, the code that is fed to the compression engine requires some pre-processing. The steps listed below are applied to all three code types, even to the *Agilla* system which in reality uses a different compilation strategy.

First, compact representation is expanded. Then code and data segments get partially separated according to 3.2.2. Relative addresses used in control execution (e.g., `jmp L1`) are translated into offsets. The same happens to memory access operands (e.g., `BUFS[0]`). System variables and constants (e.g., `ME.ID`) are left untouched, they are replaced at run-time. Following this, all volatile immediate operands are moved to the data segment where they stay till the execution starts. They are not involved in the compression. All system information (segments, flags, etc.) is removed, as they have no impact on the compression process

¹Hereinafter, we widely use the term **protocol cycle**. In fact, it has rather a compound meaning. Protocol cycle includes all the operations, which need to be executed in order to complement one logical section. For example, in case of spanning tree building it would mean creating the entire tree from scratch one time, or in case of time-sync protocol this would mean just a single synchronization cycle.

6. Experimental Setup

Platform	Radio Chip	Max Data Rate [kbps], R_{TelosB} or $R_{TinyNode}$	MCU + Radio RX [mA], RX_{TelosB} or $RX_{TinyNode}$	MCU + Radio TX (0dBm) [mA], TX_{TelosB} or $TX_{TinyNode}$	Min Supply Voltage [V], VCC_{TelosB} or $VCC_{TinyNode}$
TelosB	Chipcon CC2420 2.4 GHz	250	21.8	19.5	1.8
TinyNode-584	Semtech XE1205 868 MHz	153	16	25	2.4

Table 6.2: Radio Characteristics of TelosB and TinyNode WSN Platforms

either. The resulting “stripped” code is passed to the compression engine.

6.3 Energy Model

In order to estimate the energy consumption benefits that our method brings to a system we use the following technique. Consumption is estimated for one cycle and then extrapolated over a long-term period. This makes sense since most of our test cases are constantly running protocols. We have picked two modern WSN platforms: *TelosB* [PSC05] and *TinyNode* [FFMM06].¹ The radio characteristics used for our calculations are shown in Table 6.2.

For each test case we take the following measurements:

- P_T , number of received packets (by all nodes participating in the protocol),
- P_R , number of received packets (by all nodes participating in the protocol),
- S_{P_I} , initial packet size before compression,
- S_{P_R} , reduced packet size after compression, and
- F , the number of fragments in the code.

Additionally, we use the electrical parameters from Table 6.2. Having this information we can estimate the energy saving of the two platforms in terms of transmitted code. The energy spent on transmissions (plus reception) of the original code is expressed as:

¹*TinyNode* platform was used for outdoor installations of *PermaSense* project, *TelosB* is a convenient tool for lab use.

$$\begin{aligned}
 \text{energy spent}_{TelosB_I} &= \sum_{i=0}^F \frac{S_{P_I} * P_T}{R_{TelosB}} * TX_{TelosB} * VCC_{TelosB} + \\
 &+ \sum_{i=0}^F \frac{S_{P_I} * P_R}{R_{TelosB}} * RX_{TelosB} * VCC_{TelosB} \quad (6.1)
 \end{aligned}$$

Therefore, the energy spent on transmissions (and reception) of the compressed code can be expressed as follows:

$$\begin{aligned}
 \text{energy spent}_{TelosB_R} &= \sum_{i=0}^F \frac{S_{P_R} * P_T}{R_{TelosB}} * TX_{TelosB} * VCC_{TelosB} + \\
 &+ \sum_{i=0}^F \frac{S_{P_R} * P_R}{R_{TelosB}} * RX_{TelosB} * VCC_{TelosB} \quad (6.2)
 \end{aligned}$$

In fact, using equations 6.1 and 6.2 we simply integrate power consumption over a period of time specified by the number of transmissions in one cycle. The absolute duration of one cycle may vary. Similar equations can be obtained for *TinyNode* by replacing the electrical parameters in Table 6.2.

Finally, the overall energy saving for *TelosB* is estimated as follows (equivalently, for *TinyNode*):

$$\text{energy saving}_{TelosB} = \text{energy spent}_{TelosB_I} - \text{energy spent}_{TelosB_R} \quad (6.3)$$

The equation 6.3 is used as a plot function for energy-related graphs in this chapter (for a visual convenience we take \log_{10} of the value derived from 6.3 which in turn is computed by integrating over 1000 protocol cycles).

Topology has a potentially huge impact on how code is disseminated. Therefore, energy spent on disseminating the same code in different topologies is unlikely to be the same. We have chosen a set of indicative topologies as shown in Tables 6.3, 6.4 and 6.5. For each

6. Experimental Setup

topology we provide a set of equations to estimate the number of transmissions/receptions¹ in each cycle of the particular protocol.

The following conventions are used:

- N , number of nodes (vertices),
- E , number of edges, and
- M , number of passive molecules (in case of fraglets).

Another approach to obtain energy figures would have been using [DOTH07] but this requires extra code instrumentation and long-term experiments in a distributed fashion which is not easy.

Note, in the equation 6.2 we do not take into account the energy spent on dissemination of dictionary updates. This depends on many factors: program size, frequency of re-configuration, compression parameters. Fairly, this should be added to the total energy spending of the system.

6.4 Optimizing ChameleonVM's Code

We show how *ChameleonVM*'s code can be optimized using several examples, each of which represents a typical building block of a WSN system. We start with a very simple case of a route discovery service and go towards the attempt to build a TDMA-like time-sync protocol. In the end we demonstrate how multiple building blocks can be assembled together in a single application. At each step we run code compression, which allows gradual reduction of code size as new blocks are being added.

6.4.1 Mobile Code Version of "Hello World!"

The first example is a basic demonstration of how *ChameleonVM* code can be used to execute a single command on a node (in this case we "toggle a red LED ten times"). The code is presented in Listing 6.1 below.

```
1 . sys # SYSTEM segment
```

¹In the presence of MAC-layer only addressed receptions are counted. For broadcast transmissions the number of receptions must be multiplied by the number of physical neighbors. The equations in Tables 6.3, 6.4 and 6.5 are without MAC.

6.4 Optimizing ChameleonVM's Code





Application	Number of transmission (TX) and receptions (RX) ^a			
	<i>ChameleonVM</i> ($N = 20$)			
	H-topology (all-to-one)	A-topology (all-to-all)	V-topology (all-in-line)	M-topology (mesh)
				
"Hello World!"	$RX = (N - 1) * 2$ $TX = N$	$RX = (N - 1) * N$ $TX = N$	$RX = 1 + (N - 2) * 2 + 1$ $TX = N$	$RX = N * 2$ $TX = N$
Route discovery	$RX = (N - 1) + 1 + (N - 2)$ $TX = 1 + (N - 2) + 1$	$RX = (N - 1) * N$ $TX = 1 + (N - 2) + 1$	$RX = 1 + (\frac{N}{2} - 2) * 2 + 1$ $TX = 2 * \frac{N}{2} - 2$	$RX = N * 2$ $TX = 2 * \frac{N}{2} - 2$
Spanning tree	$RX = (N - 1) * 2$ $TX = N$	$RX = (N - 1) * N$ $TX = N$	$RX = 1 + (N - 2) * 2 + 1$ $TX = N$	$RX = N * 2$ $TX = N$
Network size estimator	$RX_F = (N - 1) * 2$ $TX_F = N$ $RX_B = N - 1$ $TX_B = N - 1$	$RX_F = (N - 1) * N$ $TX_F = N$ $RX_B = (N - 1) * (N - 1)$ $TX_B = N - 1$	$RX_F = 1 + (N - 2) * 2 + 1$ $TX_F = N$ $RX_B = N^2 + N - 1$ $TX_B = sum(1..N - 1)$	$RX_F = N * 2$ $TX_F = N$ $RX_B = N * 2$ $TX_B = N$
ID re-assignment	the same as above	the same as above	the same as above	the same as above
SBTSP	$RX = (N - 1) * 2$ $TX = N$	$RX = (N - 1) * N$ $TX = N$	$RX = 1 + (N - 2) * 2 + 1$ $TX = N$	$RX = N * 2$ $TX = N$
Data collection application	"sense and collect" activities:			
	$RX = (N - 1) * 2$ $TX = N$	$RX = (N - 1) * N$ $TX = N$	$RX = 1 + (N - 2) * 2 + 1$ $TX = N$	$RX = N * 2$ $TX = N$
	+ a combination of the above (spanning tree + ID re-assignment + time-sync)			

Table 6.3: Characteristics of Test Applications: Message Exchange Intensity (ChameleonVM)

^a TX_F and RX_F denote a capsule going forward, TX_B and RX_B – a capsule going backward. This happens when the original capsule splits.

6. Experimental Setup


Application	Number of transmission (TX) and receptions (RX) ^a
<i>FragletVM</i> ($N = 4, E = 4, M = 100$)	
Fixed Test Topology	
	
Disperser	$RX = \frac{2 * E * M}{N} \quad TX = \frac{4 * E * M}{N}$

Table 6.4: Characteristics of Test Applications: Message Exchange Intensity (FragletVM)

^a TX_F and RX_F denote a capsule going forward, TX_B and RX_B – a capsule going backward. This happens when the original capsule splits.


Application	Number of transmission (TX) and receptions (RX) ^a
<i>Agilla</i> ($N = 25$)	
Grid Topology	
	
Fire Tracker	depends on where in topology agents are injected

Table 6.5: Characteristics of Test Applications: Message Exchange Intensity (Agilla)

^a TX_F and RX_F denote a capsule going forward, TX_B and RX_B – a capsule going backward. This happens when the original capsule splits.

6.4 Optimizing ChameleonVM's Code

Short Notation	Executing Code	Function
<code>send ME,ALL</code>	<code>push ALL push 0x01 send</code>	Broadcast itself
<code>led RED,TOGGLE</code>	<code>push 0x22 led</code>	toggle the red LED
<code>delay 1000</code>	<code>push 0 push 1000 timer</code>	fire one-shot timer in 1 second

Table 6.6: ChameleonVM: Source Code Notations

```

2   Autoupdate On      # enable autoupdate (capsules of the
3                       #   higher version will be accepted,
4                       #   capsules of the same or lower
5                       #   versions will be declined)
6   Lifetime 10s      # recognized post-fixes: ms (millisec),
7                       #   s (sec), p (packets)
8   Id 0x10           # 4-bit ID + 4-bit version number
9
10  .code.init         # CODE segment "init"
11    send ME,ALL      #   broadcast itself
12
13  .code.timer0       # CODE segment "timer"
14    led RED,TOGGLE  #   toggle the red led
15    delay 1000      #   sleep for 1s

```

Listing 6.1: WSN Version of "Hello World!"

Hereinafter, in code listings we often use a short and better understood notation as shown below in Table 6.6.¹

This capsule is capable of self-propagating itself using the code in the `.code.init` segment. It is not accepted by the nodes which already have this capsule installed.² The space occupied by the capsule is freed up after 10 seconds giving it a chance to toggle the red LED only 10 times.

Logically, the code consists of three segments: the system segment (`.sys`; various parameters of the capsule), initialization segment

¹The short notation reflects the actual byte-code but is not necessarily a one-to-one correspondence.

²We limit the lifetime by 10 seconds, after this time the capsule will be destroyed and if it comes back again the node will install it.

6. Experimental Setup

(`.code.init`; executed only once when the capsule is installed on a node) and timer segment (`.code.timer`; this is essentially a timer handler). The initialization segment is responsible for self-disseminating (ME) the capsule. Here we use broadcast (ALL) transmission. The timer segment encapsulates the logic for toggling the red LED every second.

If we now apply the compression algorithm to this code in the final stage the code would most likely look different, as it is shown in the short notation in Table 6.6. *ChameleonVM* may run compression over several segments in order to achieve better results. This type of code can reach an extremely good **Compression Factor (CF)** since all static data (addresses, constants) are stored separately within the data segment.

Note that we can use a modified version of the code by moving everything to the initialization segment as shown in Listing 6.2.

```
1 .code.init          # CODE segment "init"
2   send ME,ALL      # broadcast itself
3 L1:
4   led RED,TOGGLE  # toggle the red led
5   delay 1000      # sleep for 1s
6   jmp L1          # do it periodically
```

Listing 6.2: Alternative WSN Version of “Hello World!”

This code would result in a small size increase. What is more important is that the `delay` operation becomes blocking if it is used outside of `.code.timer` segment. The execution will not proceed until the timer expires. In the first version `timer` just sets up a timer for the future and execution continues. Despite this, both versions are functionally identical.

According to the test settings (see Section 6.2) and the topology variants from Section 6.3 we obtain the results shown in Figure 6.1. In our experiments the autoupdate is always switched on in order to avoid backward (viral) propagation.

6.4.2 Route Discovery

As was discussed in Section 3.2.7 capsules can be used for packet processing and traffic control. In this example, we show how to use

6.4 Optimizing Chameleon VM's Code

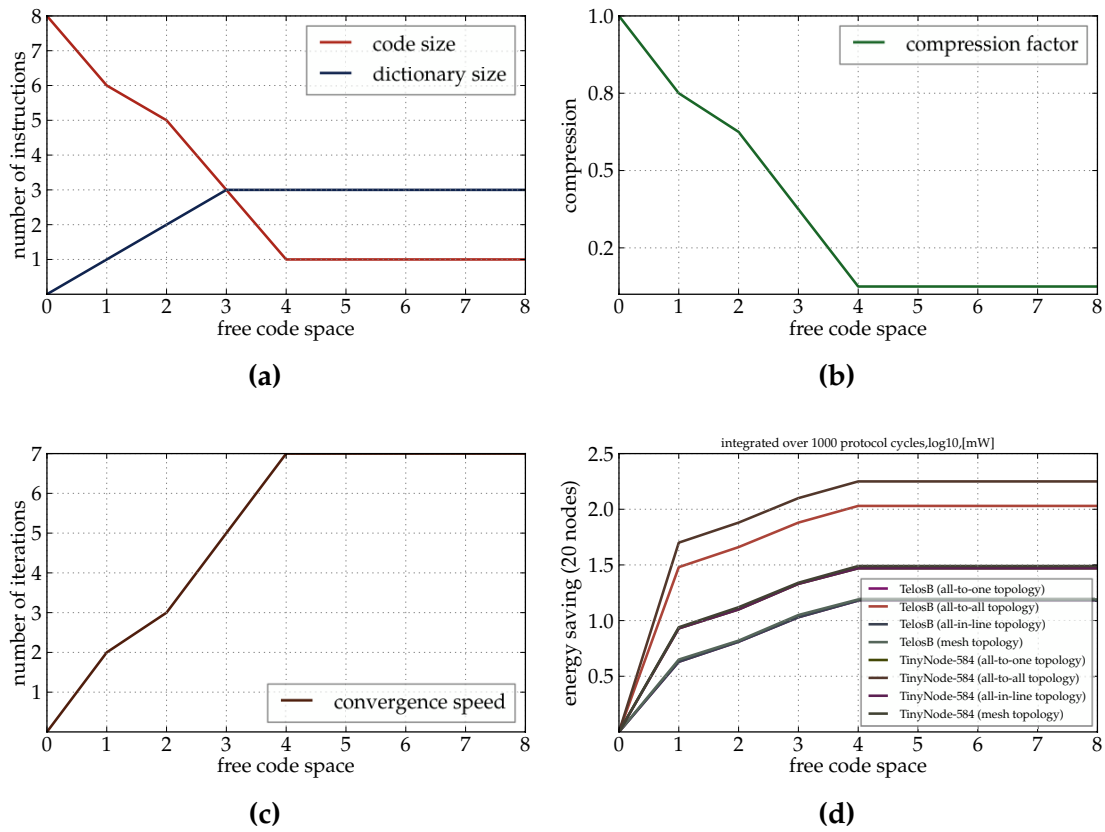


Figure 6.1: Code Compression Process: "Hello World!" Application

6. Experimental Setup

capsules to establish a temporary forwarding channel between two nodes in the network. The scenario is displayed in Figure 6.2.

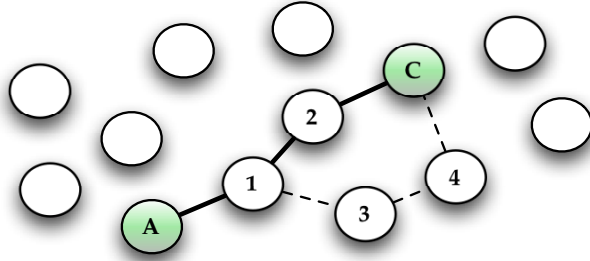


Figure 6.2: Route Discovery

Here two nodes, *A* and *C*, would like to establish a dedicated path for packet exchange. First, let's have a look at the code which achieves this (see Listing 6.3).

```
1 .sys # SYSTEM segment
2   Autoupdate On
3   Lifetime 10s
4   Id 0x20
5
6 .bufc # DATA segment
7   from=A # we start discovering from node A
8   to
9
10 .code.init # CODE segment "init" (executed once)
11   mov from,ME.FROM # remember the "previous" hop
12   jmq NID,C,L1
13   send ME,ALL # broadcast itself
14   exit
15 L1: send NULL,from # send a null packet back
16
17 .code.pack # CODE segment "receive packet"
18   jmq PACK.SRC,A,L2
19   jmq PACK.SRC,C,L3
20   exit
21 L2: jmq PACK.DST,C,L5
22   exit # exit point (the capsule stays alive)
23 L3: jmq PACK.DST,A,L4
24   exit # exit point (the capsule stays alive)
```

6.4 Optimizing ChameleonVM's Code

```
25 L4: send PACK,from      # process packets from C to A
26     mov to,PACK.FROM    # remember the "next" hop
27     exit
28 L5: send PACK,to        # process packets from A to C
```

Listing 6.3: Route Discovery

The following conventions are used: `*.SRC` and `*.DST` are addresses encoded in the underlying link-layer's header (source and destination), their representation depends on the execution environment. `PACK.SRC` and `PACK.DST` are fixed for each packet and do not change, while `PACK.FROM` and `PACK.TO` define previous and subsequent hops and change as the packet goes through. The same is valid for capsules but the `CAP.*` identifier must be used instead. `ME.*` identifies this capsule.

The shown code essentially does the following:

1. Upon arrival on a new node we first check if we have yet reached our destination (*C*). If yes, we send an empty (`NULL`) packet back to the source. Otherwise, we virally self-propagate the code further. The empty packet sent back is used to establish a bi-directional channel between two end points (*A* and *C*), so that the destination (*C*) would be able to send packets to the source (*A*) as well.
2. On all intermediary nodes we just keep a track of "previous" and "next" hops in the path (the "previous" hop is recorded when the capsule goes towards *C* and the "next" hop is determined when the empty packet passes by).
3. Starting from this point all packets originated in the source *A* and addressed for the destination *C* will be directly delivered to *C*. Similarly, for packets generated by *C* for *A*.

Table 6.7 shows the address assignment process for the case illustrated in Figure 6.2.

A number of extra features could be potentially useful within this code. The route discovery process is supposed to start from the source node (*A*) and run towards the destination node (*C*). In case the capsule is injected to a node different from *A* we would have to discover node *A* first, i.e., to complement the code with this functionality. Additionally, we might want to preserve the capsules in the established route from decay after 10 s.

6. Experimental Setup

Address	A	1	2	C
capsule propagation ($A \rightarrow C$)				
src	A	A	A	A
dst	C	C	C	C
from	A	A	1	2
to	?	?	?	?
empty packet propagation ($C \rightarrow A$)				
src	C	C	C	C
dst	A	A	A	A
from	A	A	1	2
to	1	2	C	C

Table 6.7: Example: Address Assignments During Route Discovery Process

The code seen from node C highlights the feature of *ChameleonVM* that even locally generated packets first go through pre-processing by capsules and only afterwards are sent out.

The presented code allows organizing a temporary forwarding without using underlying topological information, e.g., routing tables. The capsule approach for modifying the default routing rules is more convenient as it does not require maintenance of routing tables on each node and can be used point-wise. Since the example is rather simplistic it does not try to find an optimal route. Furthermore, packet duplication might occur, as multiple paths may be active at the same time.

In order to test how this route discovery code reacts to the code compression we make the following assumption: for V- and M-topologies from Table 6.3 the length of the discovered route must be a half of the total number of nodes ($L = \frac{N}{2} = \frac{20}{2} = 10$). This is necessary as the result highly depends on the topology. The results are shown in Figure 6.3.

As it can be seen from the graphs in Figure 6.3 even the doubled FCS does not allow the code to fully converge. The required size of FCS would be 11 in this case.

6.4.3 Spanning Tree

The next example we consider involves building a spanning tree (see Listing 6.4), a typical architectural solution for sensor nets. The existing

6.4 Optimizing Chameleon VM's Code

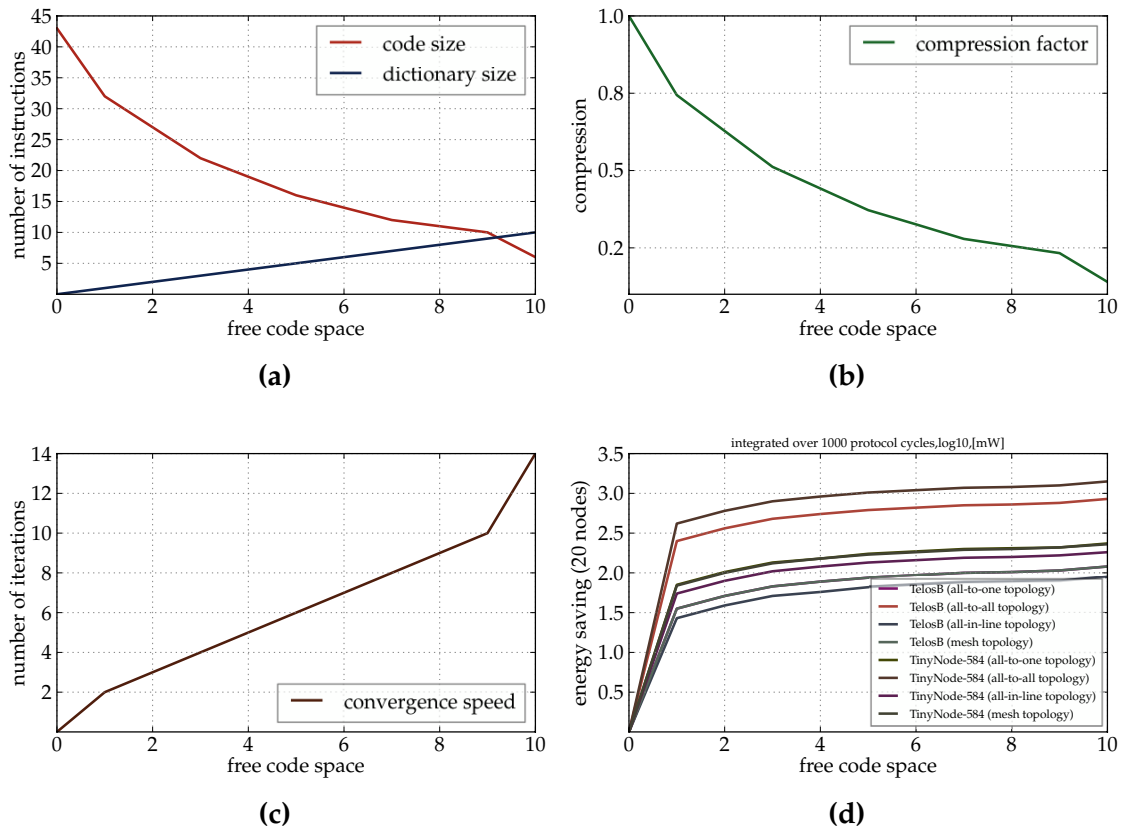


Figure 6.3: Code Compression Process: Route Discovery

6. Experimental Setup

data gathering protocols like [BvRW07]¹ support very sophisticated tree building and maintenance mechanisms. They allow reconstruction of parts of the tree, optimization of its structure and are energy efficient at the same time. Our algorithm is rather simple. It cannot react to possible changes in the topology (e.g., in case of mobile nodes) in a traditional way by modifying its structure. Instead, we periodically create a snapshot of the tree and use it for a limited period of time. The changes in the topology are reflected when the tree is re-built again from scratch. Obviously, such an approach comes with a huge transmission load, which we try to compensate using our compression scheme. To list the other negative aspects, we do not take into account possible collisions during packet transfers, e.g., bad links, different load on links, etc. Nevertheless, the algorithm demonstrates how to temporarily structure the group of nodes in a way to make data extraction possible. Later, in Sections 6.4.4 and 6.4.5, we show how some useful functions can be executed based on this structure.

```
1 .sys # SYSTEM segment
2   Autoupdate Off # disable autoupdate
3   Lifetime 10s
4   Id 0x30
5
6 .bufc # DATA segment (allocated inside the
7       # capsule)
8   from=S # "S" is some real network address
9   hops=0 # these are local variables
10
11 .code.init # CODE segment "init" (executed once)
12   inc hops
13   push BUFS[0] # first we check the ID
14   jmqeq ME.ID,L1 # "ME.*" - this capsule, "CAP.*" -
15                 # capsule, "PACK.*" - packet
16   mov BUFS[0],ME.ID # store ID and "hops" in the shared
17   mov BUFS[1],hops # memory BUFS (allocated from the
18   jmp L2 # node's memory pool)
19 L1: push BUFS[1] # check the distance (level)
20   jmplet hops,L3
21   replace # replace the existing capsule
22 L2: mov from,ME.FROM
23   send ME,ALL # broadcast itself
24   exit
```

¹The modified version of *Dozer* is used in *PermaSense* installations.

6.4 Optimizing Chameleon VM's Code

```
25 L3: die # if none - kill the capsule
26
27 .code.pack # CODE segment "receive packet"
28 # (executed upon receiving a packet)
29 jmqeq PACK.DST,S,L4 # process packets addressed to S
30 # (sink node)
31 exit # exit point (the capsule stays
32 # alive)
33 L4: jmqlet PACK.TO,NID,L5 # ignore all packets addresses to
34 exit # the others
35 L5:
36 send PACK,from # send a packet up the spanning tree
```

Listing 6.4: Spanning Tree

Schematically, the spanning tree building algorithm is shown in Figure 6.4.

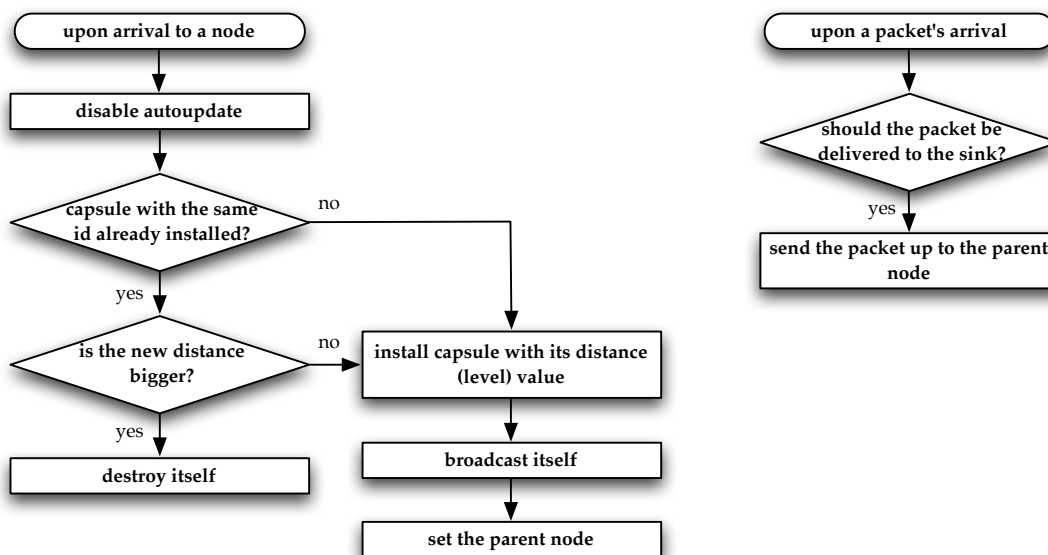


Figure 6.4: Building Spanning Tree Algorithm

In this example, we see that by disabling autoupdate we can control the capsule installation process. Depending on the situation capsules can decide whether to replace the already installed code or not. As with the “Hello World!” example, this code is self-propagating and time-limited. After installation on a node, the code only processes data packets addressed to the sink node (S) via this path and ignores

6. Experimental Setup

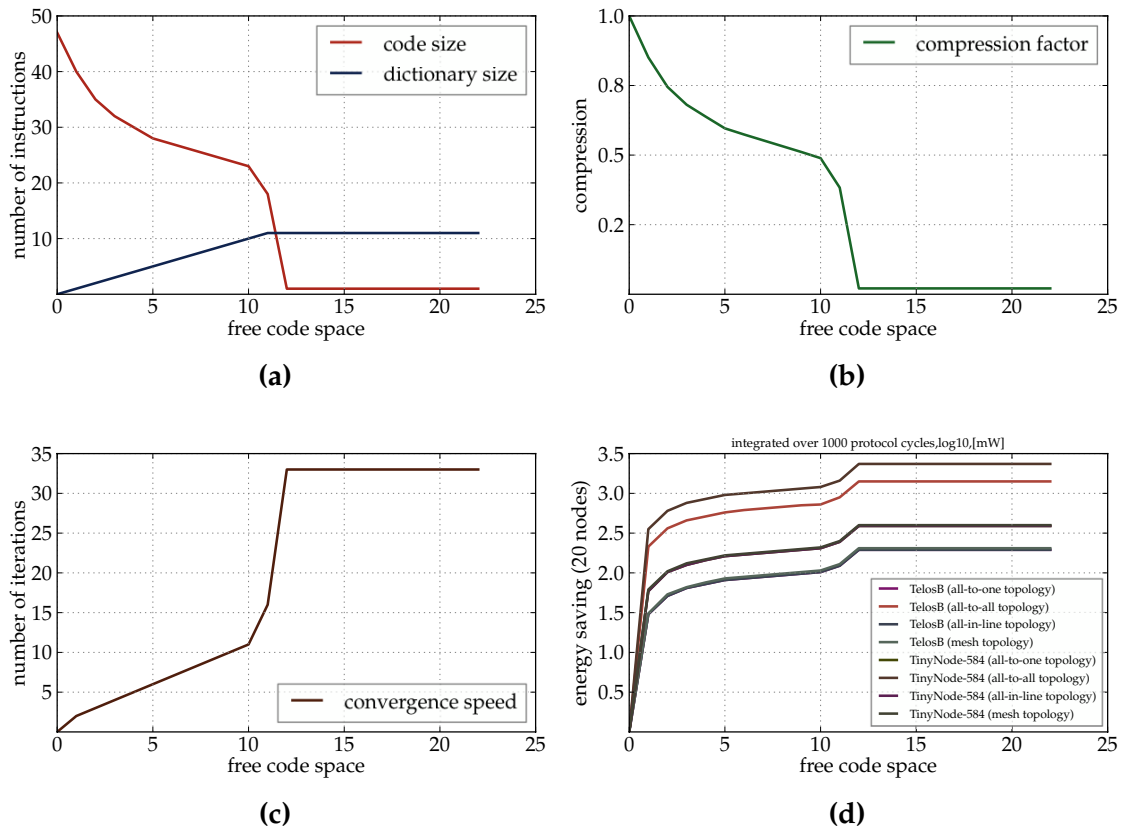


Figure 6.5: Code Compression Process: Spanning Tree Builder

everything else. Packets generated locally or received from the children are pushed upwards in the spanning tree. In order to reflect changes in the tree structure this capsule must periodically (10 s) flood the network. The distribution of the code is supposed to start from the sink node.

The performance of the method depends on the topology as capsules can replace each other while they are leveling the tree. In case of uneven code propagation multiple transmissions are possible. In our experiments we assume uniform code propagation, i.e., each node transmits only once. The results are shown in Figure 6.5.

6.4.4 Count the Number of Nodes

As it has been mentioned before having a structured network allows implementing various useful functions on it. In the next two examples we assume the spanning tree structure of the network, i.e., that the

6.4 Optimizing ChameleonVM's Code

capsule from Section 6.4.3 is already installed and running.¹ Note that the spanning tree source code from Listing 6.4 must be slightly changed to be able to process capsules, not packets.

The first utility allows counting of the number of nodes in the system and collection of this information at the sink node that is to perform network size estimation. The program contains two capsules: the capsule which propagates through the network and marks each node, the “traveling marker”, and the capsule which resides on the sink node and collects information, the “collector”. Traveling markers move down the spanning tree. Its code is presented in Listing 6.5.

```
1 .sys # SYSTEM segment
2 Autoupdate On
3 Lifetime 10s
4 Id 0x40
5
6 .code.init # CODE segment "init"
7 send ME,ALL # broadcast itself
8 erase TOP # clean the code located above
9 sendd ME,ME.FROM,S # send it up the spanning tree
10 die
```

Listing 6.5: Network Size Estimator: Traveling Marker (node)

Note, we used `sendd` instruction instead of `send`. `sendd` allows us to specify the final destination address (`*.DST`) as well as the next hop (`*.TO`). `send` accepts the next hop only.

What a traveling marker essentially does is: it self-propagates and when it reaches a new node it clones to the neighboring nodes, self-modifies into a “signal” capsule and self-navigates to the base. Signals move up the spanning tree. This example demonstrates how code shrinking (see Section 4.1) allows to reduce the size of the code while it moves across the network. This process is shown in Figure 6.6.

Lines 7–8 in Listing 6.5 will be removed from the traveling marker’s code when it turns into a signal. This will reduce pure code size from 9 to 5 instructions.

The “collector” (see Listing 6.6) resides on the sink, counts incoming signal capsules and accumulates it as a readable value.

¹A network can be sequentially programmed with capsules of different nature. Those capsules will co-exist on the same node and interact, if necessary.

6. Experimental Setup

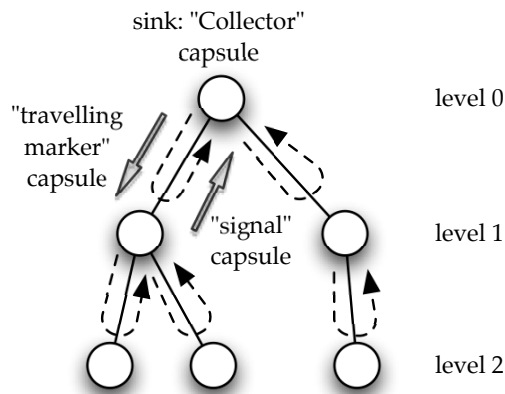


Figure 6.6: Network Size Estimation using Capsules

```
1 .sys # SYSTEM segment
2 Autoupdate On
3 Lifetime 10s
4 Id 0x50
5
6 .code.cap # CODE segment "receive capsule"
7 push CAP.ID # count capsules only
8 jmqeq 0x40,L1
9 exit
10 L1: inc BUFS[0]
```

Listing 6.6: Network Size Estimator: Collector (sink)

In our experiments on code compression rate we consider only the forth (traveling marker) capsule (see Figures 6.7a – 6.7c) whereas energy estimations are made separately for forth and back (signal) capsules. The moment of conversion of one into the other is shown in Figure 6.7d. As it can be seen, code shrinking allows to significantly reducing the size even on the compressed code. In Section 3.2.2 we explained the mechanism of *ChameleonVM* which allows it to efficiently process compressed code in case of split/merge operations. The shown graphs do not include the impact of having a spanning tree builder running as a parallel process. In Section 6.4.7 we demonstrate how code size and energy estimations for a complex application can be done.

6.4 Optimizing Chameleon VM's Code

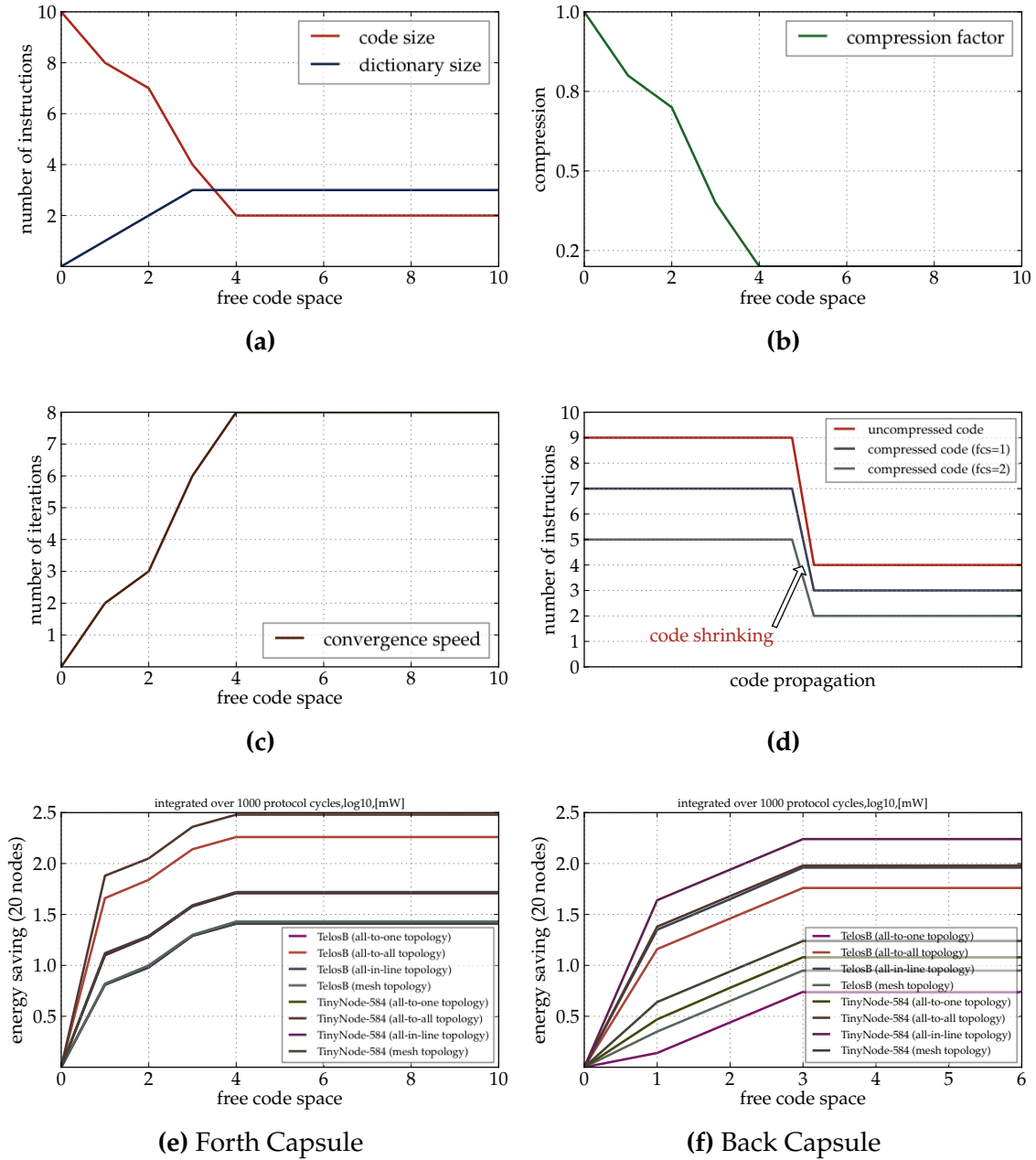


Figure 6.7: Code Compression Process: Network Size Estimator

6. Experimental Setup

6.4.5 ID Assignment

The second of our two examples is a node ID (re-)assignment program. We design it as an extension of the count-program from Section 6.4.4. The main working principle remains the same: the initial capsule propagates through the network, renumbers nodes (assigns them unique new IDs), converts into a signal capsule containing the newly assigned ID and moves back to the root where all IDs are accumulated in a buffer.

Sometimes it is required to (re-)assign new IDs to already deployed nodes. This could be required for various reasons including optimization of address space or in case of nodes with the same ID colliding. This is particularly important for WSN applications where a local node ID is used as a network-wide ID in many protocols. Again, we use the spanning tree from Section 6.4.3 as a base for our program. The (re-)assignment is done using the capsule shown in Listing 6.7.

```
1 .sys # SYSTEM segment
2   Autoupdate 0n
3   Lifetime 10s
4   Id 0x60
5
6 .code.init # CODE segment "init"
7   send ME,ALL # broadcast itself
8   sense TEMP # choose initializer for A
9   pop BUFS[0] # A = 18000 * (A & 65535) + (A >> 16)
10  push BUFS[0]
11  and 65535
12  mult 18000
13  push BUFS[0]
14  rsh 16
15  add
16  sense TEMP # choose initializer for B
17  pop BUFS[0] # B = 36969 * (B & 65535) + (B >> 16)
18  push BUFS[0]
19  and 65535
20  mult 36969
21  push BUFS[0]
22  rsh 16
23  add
24  lsh 16 # (B << 16) + A (32-bit result)
25  add
26  pop BUFC[0] # store the new ID in the capsule
```

```
27 erase TOP # clean up the top part
28 sendd ME,ME.FROM,S # send it up the spanning tree
29 die
```

Listing 6.7: Node ID Assignment: Traveling Capsule (node)

This capsule performs automatic node ID assignment based on a measured temperature value (can be humidity, or any other available 16-bit sensor, or a mix) and a simple pseudo-random number generator shown in Listing 6.8.¹ The use of sensor values allows generating truly random IDs.

```
1 m_w = <choose-initializer>; /* must not be zero */
2 m_z = <choose-initializer>; /* must not be zero */
3
4 uint get_random()
5 {
6     m_z = 36969 * (m_z & 65535) + (m_z >> 16);
7     m_w = 18000 * (m_w & 65535) + (m_w >> 16);
8     return (m_z << 16) + m_w; /* 32-bit result */
9 }
```

Listing 6.8: Node ID Assignment: “Multiply-With-Carry” Random Number Generator

The capsule in Listing 6.7 is distributed only once. Upon arrival on a node it is executed there. The new ID is assigned to the node. As a result a signal capsule is formed which contains the newly generated node ID. This capsule makes its way back to the top of the spanning tree.

Similarly to the “collector” from Section 6.4.4, a simple capsule could be designed which would reside on the sink node, collect all incoming signal capsules containing IDs and store them in a buffer.

The results of running code compression on this code are shown in Figure 6.8. As with the count-example from Section 6.4.4, we show only the forth (traveling) capsule in the code compression related graphs (see Figures 6.8a – 6.8c) whereas energy estimations are illustrated separately for forth and back (signal) capsules. The turning moment and its implication in code size reduction process are shown in Figure 6.8d.

¹The shown implementation is of G. Marsaglia.

6. Experimental Setup

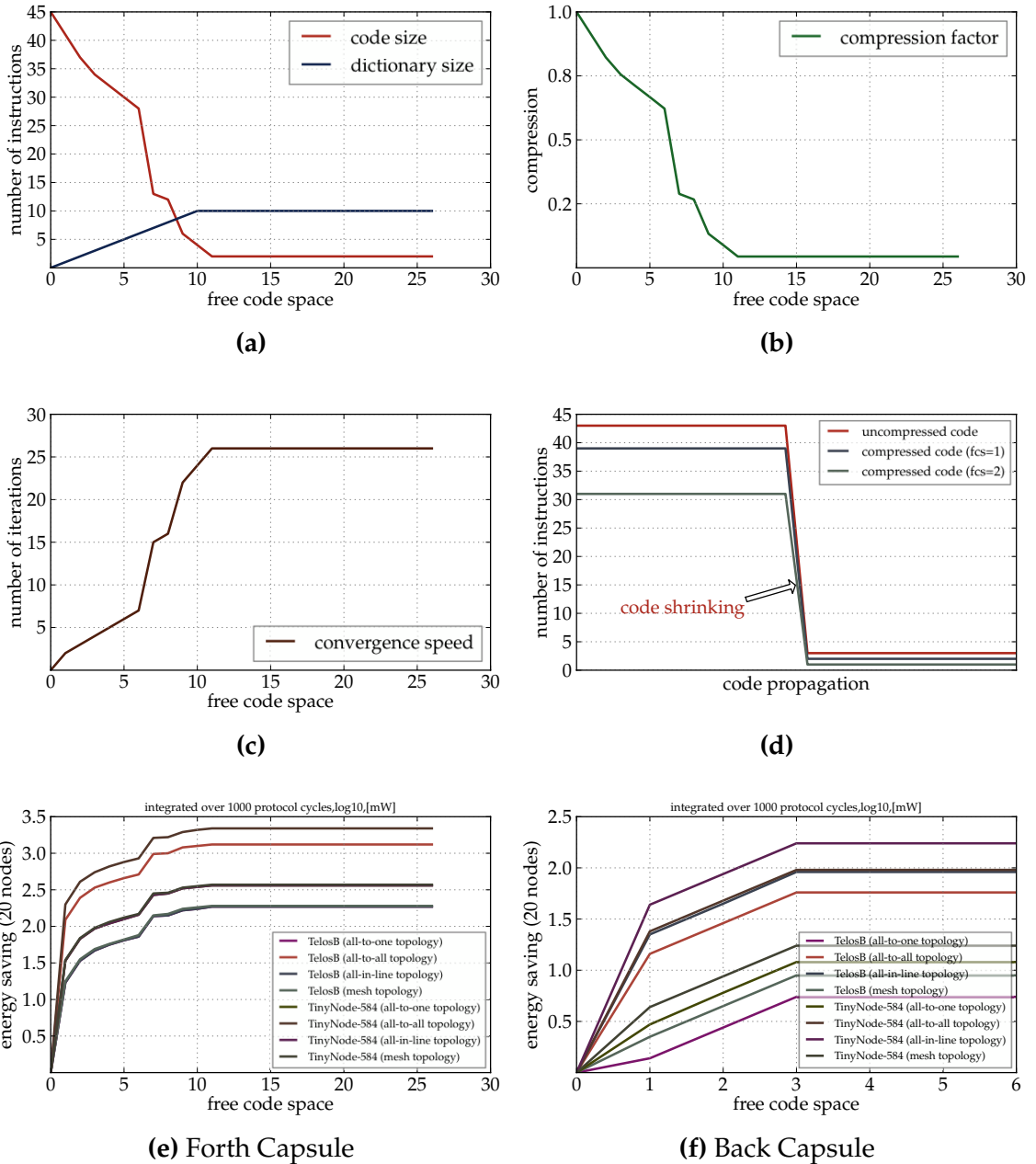


Figure 6.8: Code Compression Process: Node ID Assignment

6.4.6 Skew-Balance Time Synchronization Protocol

After analyzing the relatively simple examples in Sections 6.4.1 – 6.4.5 we move towards building a more sophisticated program and for that we re-design a lightweight time synchronization protocol [Tal08] which we previously developed during the test-phases of the *PermaSense* project mentioned in Section 1.4. The initial implementation was done under *TinyOS 1.x*. The pseudo-code and *TinyOS*-code for the protocol are shown in Appendix A, in Listings A.1 and A.2, respectively.

Time synchronization is an essential part of many WSN applications. Measurement series have value only when they are arranged in time. A time division approach also allows putting nodes into deep sleep between measurement and data exchange cycles, thus saving energy. A lot of research has been done in the area of time-synchronization for WSN domain [RN10]. The most notable works include: *RBS* [EGE02], *TPSN* [GKS03], Römer's scheme [RÖ1], *LTS* [vGR03], *FTSP* [MKSL04], *GTSP* [SW09], *PulseSync* [LSW09]. Our goal at that stage of the *PermaSense* project was to develop a simple protocol for use in prototyping our WSN-based system, at the same time it was meant to be tailored to the specific needs of our project.

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

The protocol is based on a common sync time window where each node has a time slot when it has to send a beacon message. This beacon

6. Experimental Setup

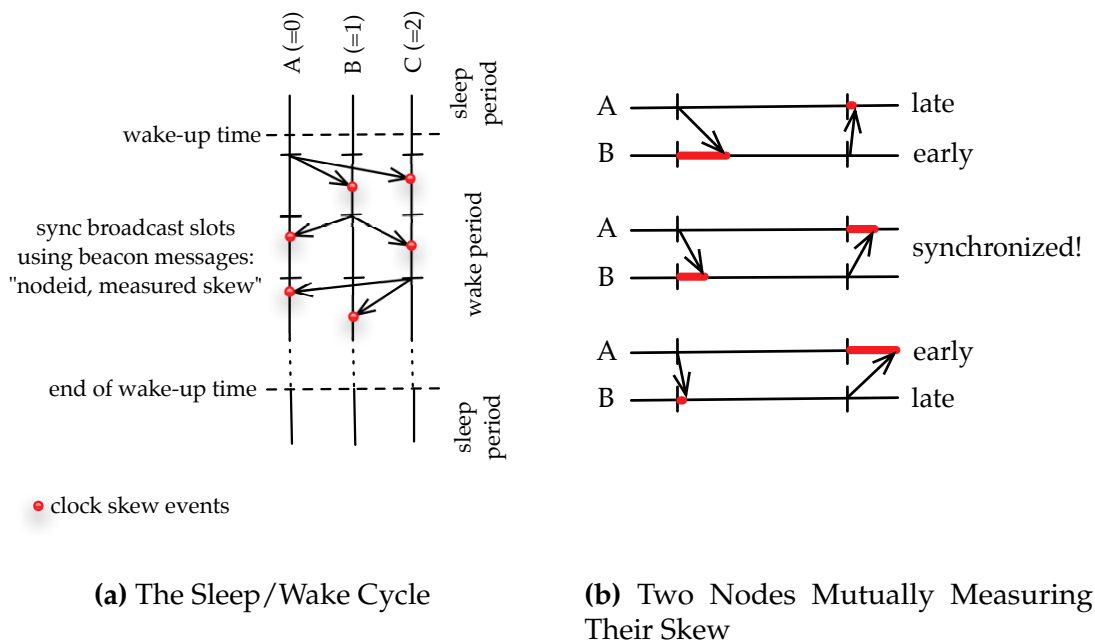


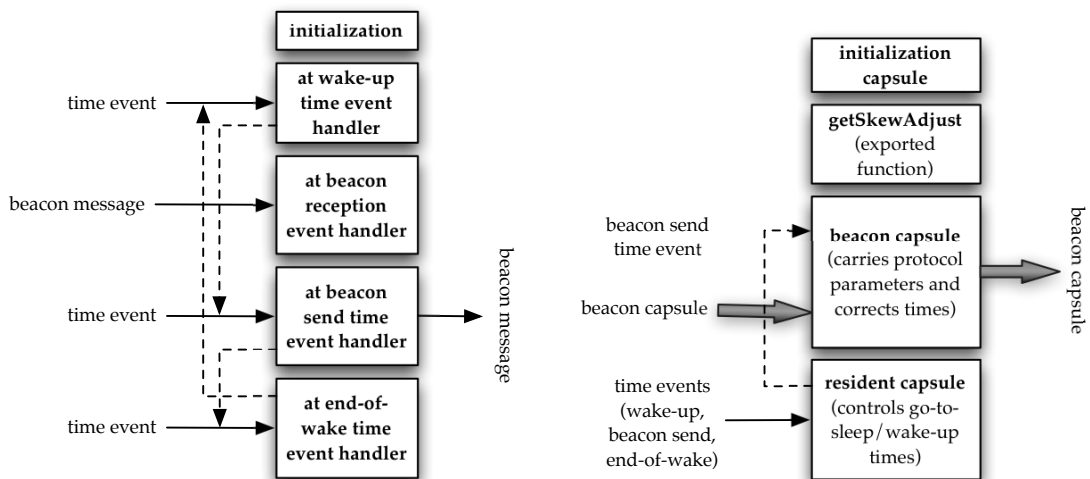
Figure 6.9: SBTSP: Working Principles

message is broadcast and serves for: 1) letting other nodes measure how much skew their local clock has when compared to the sender, and 2) letting the node announce how much skew on average it measures. The beacon exchange process is shown in Figure 6.9a. Figure 6.9b illustrates how two nodes mutually measure their skew. The mathematical model and the proof of convergence of the algorithm are discussed in detail in [Tal08]. The document also describes various aspects of the design such as recovery strategies, scaling mechanism, post-hoc calibration. Here we do not discuss them, as it is not important in the current context.

The original implementation was based on the traditional message exchange mechanism (see Figure 6.10a). We have re-designed the protocol using the concept of mobile code and the programming tools provided by *ChameleonVM*. If we compare these two versions we will see that the implementation based on code exchange (see Figure 6.10b) is more straight-forward and has a clearer structure; functionally the two versions are identical.

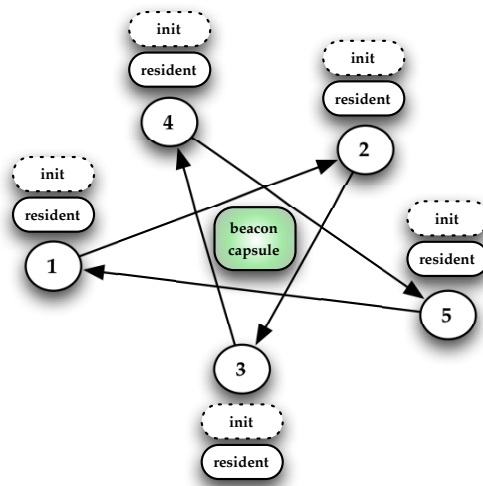
In the mobile code version, the protocol is functionally encapsulated into three capsules and one export function (for more on export functions see Section 3.2.13). The initialization capsule (see Listing A.3

6.4 Optimizing ChameleonVM's Code



(a) Message-Based Implementation

(b) Mobile Code and ChameleonVM-Based Implementation



(c) Capsule Location

Figure 6.10: SBTSP: Implementation

6. Experimental Setup

in Appendix A) sets up necessary variables and constants. A resident capsule resides on each node and switches the system between sleep and awake modes (Listing A.4, Appendix A). The beacon capsule moves across the network and allows nodes to measure and correct their mutual skews (Listing A.5, Appendix A). This process is shown in Figure 6.10c. The export function *getSkewAdjust* (Listing A.6, Appendix A) implements a linear regression algorithm which is used to correct skews; it is called from the resident capsule.¹

The init and resident capsules are disseminated only once. Moreover, the init capsules needs to be executed only once, after that it can be removed from a node. The capsule dissemination order matters: 1) the resident capsule must be installed before the init, and 2) the beacon capsule must be installed after the init and resident. Both limitations can be overcome by switching the `AUTOEXEC` flag off (see Section 3.2.6) for all the capsules and then triggering execution of the init capsule manually. Alternatively, the capsule's presence can be checked from the source code but this would increase the size. To simplify the design we distribute a linear regression algorithm as a *ContikiOS* loadable module. It is then called from *ChameleonVM* capsules as an export function through an instruction assignment.

The beacon capsule is moving across the network all the time in order to keep the nodes synchronized. The protocol uses local node IDs as a reference for the slot within the synchronization window (see Figure 6.9a: node *A* (ID=1) transmits in the first slot, node *B* (ID=2) in the second, *C* (ID=3) in the third, and so on). Each node transmits exactly 1 and receives $N - 1$ beacon capsules in every round. Previous capsules are replaced by newer ones, the last (out of $N - 1$) arrived capsule is used by the node itself to participate in the synchronization round. As it can be seen from the code in Listing A.5 the init segment of the beacon capsule plays the role of *message.received()* handler on the recipient side. Algorithmically, the process can be seen as in Figure 6.10c where in each cycle the beacon capsule travels the same path:

¹In the program listings, in order to save space we use a pseudo C-like code which can be easily translated into *ChameleonVM* notation used above. `SHMEM[X]`, `BUFS[X]`, `BUFC[X]` denote the memory region where the corresponding variable or constant *X* resides.

6.4 Optimizing ChameleonVM's Code

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

Table 6.8: SBTSP: Comparison of Static and Mobile Versions

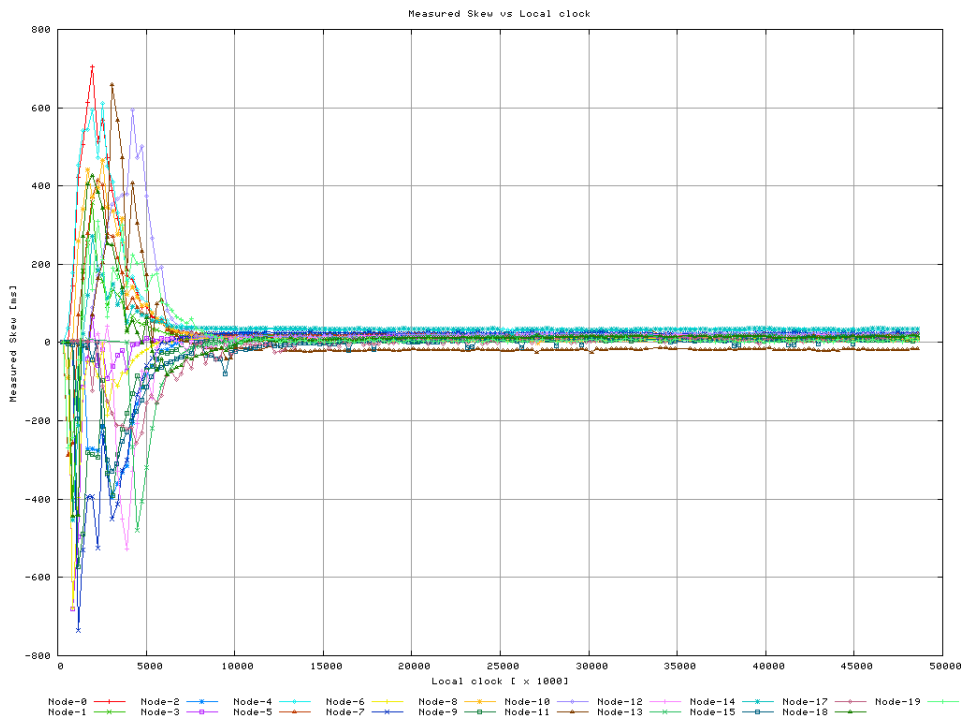
1 → 2 → 3 → 4 → 5.² In reality, in Figure 6.10c nodes might not even be physically interconnected with each other via direct links, but via multiple hops. Logically, code goes around the network and gets back to the node 1 where it waits for the beginning of the next synchronization cycle. Since *ChameleonVM* environment is dynamic, the process continues as long as the beacon capsule keeps moving across, with no connectivity for a long period of time the code will decay.

The mobile version of the protocol behaves identically to its static *TinyOS*-based predecessor. The experiment carried out on the real network of 20 *TelosB* nodes grouped in a fully meshed topology and observed for about 14 hours is shown in Figure 6.11a. Test results with different topologies and settings are presented in [Tal08]. The mutually measured skew shown in the graph is proportional to the adjustment (see Figure 6.11b) applied on each node to correct its wake-up time (the output of the *getSkewAdjust* function from Listing A.6). The first 8 rounds are without any skew adjustments because nodes are filling their circular averaging buffer. Then comes a sudden onset where skew is fully corrected and afterwards we have gradual adjustments due to the drift. After this initial phase we have about 20 rounds where the system tries to set itself up and converge. After reaching a balanced state, the system applies just a slight correction in each round in order to maintain this state.

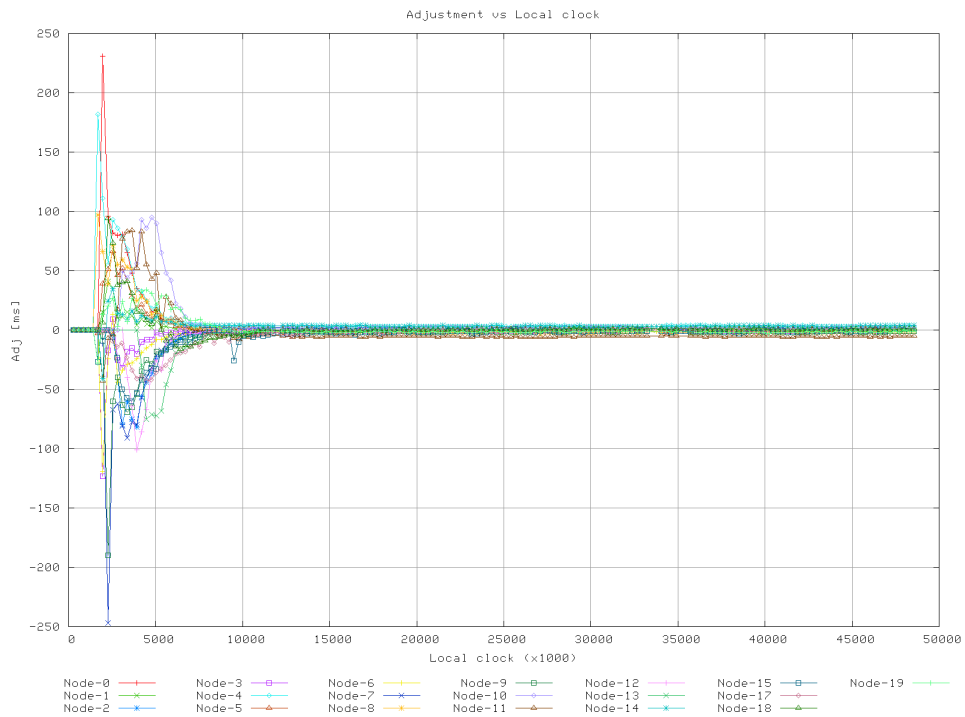
Table 6.8 shows a comparison between static and mobile versions of the protocol in regard to code size.

²Schematically it seems to be one single capsule traveling the network. In fact, it is continuously being replaced by a newly arrived copy.

6. Experimental Setup



(a) Measured Skew



(b) Adjustment

Figure 6.11: SBTSP: 5 min sync interval, almost fully mesh topology, first 14 hours of the run shown

6.4 Optimizing ChameleonVM's Code

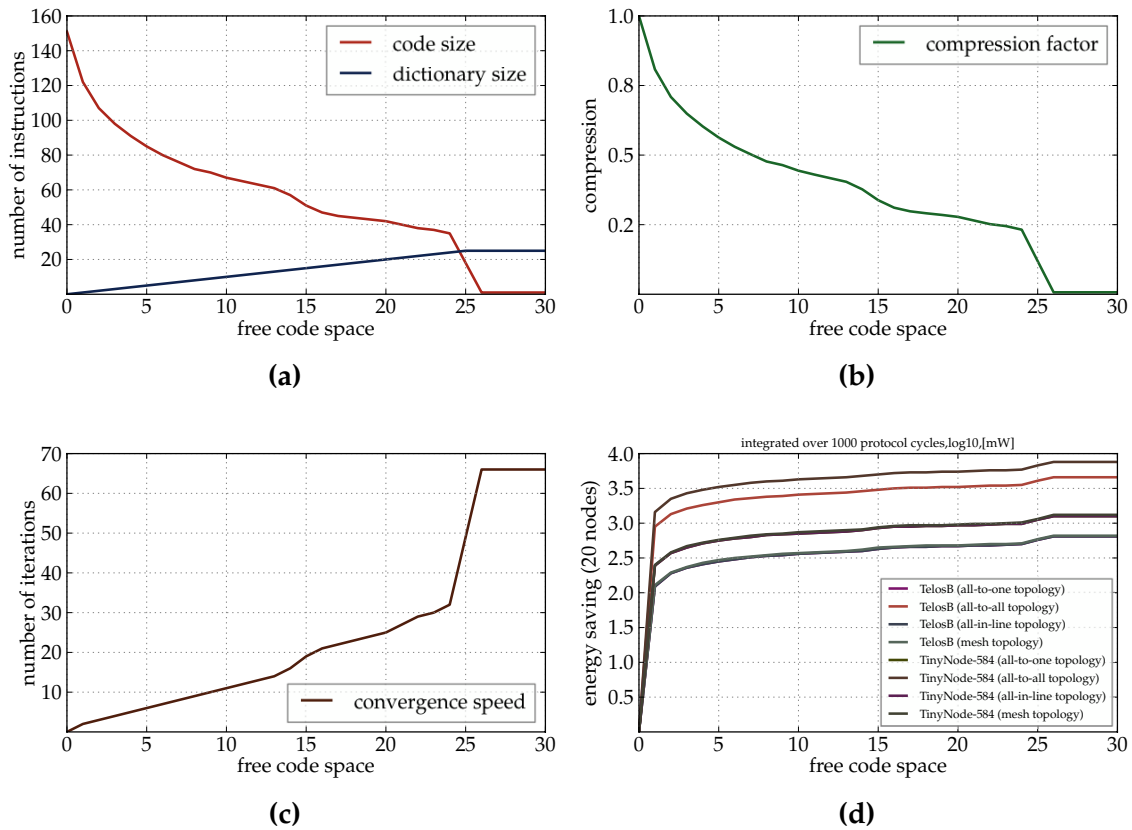


Figure 6.12: Code Compression Process: SBTSP

With mobile code version the packet size increases significantly, since now we transmit actual code and not only static data fields. This will indeed have a negative impact on the energy spending in a long term. On the other hand, applying code compression to the traveling code we manage to almost half the packet size (see Table 6.8: 152 vs 72 bytes).

Our solution does not suffer from possible code losses, as there is always, at least, one copy of the beacon capsule on each node. In this respect it is similar to the static counterpart.

Since the init and resident capsules are supposed to be disseminated only once they are not included in the compression process. Thus, we consider only the beacon capsule here. The results are shown in Figure 6.12.

A couple of observations could be made from the analysis of the graphs in Figure 6.12. Full convergence (single instruction encoding) is

6. Experimental Setup

possible if FCS is big enough. In case of *SBTSP* the threshold is 26 free opcodes. Pushing FCS beyond this level does not result in any gain in compression. The curve in Figure 6.12b has a clear exponential decay form. This was not so obvious in the previous examples. This is easily explainable. The original stream has more pair-wise patterns. The more code shrinks the fewer patterns can be found. Consequently, the convergence speed (number of algorithmic cycles) in Figure 6.12c shows an exponential growth. In fact, it is inverse to the CF in Figure 6.12b.

6.4.7 Data Collection Application

Finally, we come to the point where we can start building a real-world scenario using the software pieces presented above. We intend to build a data-collection WSN application with the following features:

- The WSN has one sink which is also used as an access node. Code dissemination starts from the sink.
- Data sensing is done periodically, data is pushed towards the sink in each cycle if there is connectivity.
- Measurements must be synchronized in time.
- The sink stores the collected data in a buffer for further processing.
- No pre-configured software apart from the *ChameleonVM* instance is installed on each node.
- Nodes do not have pre-configured addresses.

In order to achieve all of these goals we use the independent code pieces previously described. The following capsules are used as building blocks for the application:

1. MAC-layer and time-sync: *SBTSP* (Section 6.4.6),
2. routing: spanning tree builder (Section 6.4.3),
3. address resolution: ID assignment (Section 6.4.5), and
4. sense/collect functionality.

The only part, which is still missing is the sense/collect functionality. The following two capsules are responsible for taking and collecting measurements. As before, the algorithm assumes that we already have an established tree topology in the network. The node-based capsule

6.4 Optimizing ChameleonVM's Code

periodically initiates sensing on each node, accumulates a buffer of 10 measurements and sends it back up to the top of the spanning tree. The code is present in Listing 6.9. The sink node does not execute this capsule.

```
1 .sys # SYSTEM segment
2   AUTOUPDATE On
3   LIFETIME 0 # live forever
4   ID 0x70
5
6 .code.init
7   push 10
8   append BUFC,NID # prepend the buffer with node ID
9
10 .code.timer0
11   dec
12 L1: ifeq 0,L2 # check the counter
13   sense # "sense" is "sense TEMP" on this node
14   append BUFC # append to BUFC
15   delay 60s # sleep 1 min
16   exit
17 L2: push 10 # start loop again
18   append BUFC,NID # prepend the buffer with node ID
19   delay 60s
20   erase TOP
21   sendd ME,ME.FROM,S # send the buffer up
22   die
```

Listing 6.9: Data Collection Application: sense and send measurements to the sink

The resulting application would have code size characteristics as specified in Table 6.1 (sink-based capsule is excluded).

Line 13 in Listing 6.9 above is an example of using code polymorphism. Instead of explicitly specifying the parameter (what to measure: temperature, humidity, etc.) for the `sense` operation by pushing it on the stack, the definition is made directly in the dictionary. This eliminates the need to specify it in the code. On multiple nodes this instruction can be defined differently bringing different sensing in action when it is called. The transmitted code becomes more generic. Also note, this capsule does not have a lifetime limitation. It stays on each node forever.

6. Experimental Setup

The last capsule of the application framework resides on the sink node. It receives measurements from the other nodes and stores them in the **SHMEM** buffer (see Listing 6.10) in the following format:

$$[nodeid_1, 10 \text{ values}], [nodeid_2, 10 \text{ values}], \dots$$

This data can be read out by an external program for further processing using access methods to the shared memory.

```
1 .sys # SYSTEM segment
2     AUTOUPDATE On
3     LIFETIME 0
4     ID 0x80
5
6 .code.cap # CODE segment "receive capsule"
7     push CAP.ID # count " capsules only
8     jmqeq 0x70,L1
9     exit
10 L1: append SHMEM,CAP.BUFC # append to shared memory:
11 # [ID1,10 values], [ID2,10 values],
12 # ...
```

Listing 6.10: Data Collection Application: collect and store measurements in a sink's buffer

The resulting code (all capsules in the set) has 26 unique instructions and a code resolution of 5 bits, correspondingly. Table 6.9 illustrates the code size estimates of each capsule in the set.

Obviously, the number of instructions remains the same but the growing alphabet size causes the binary code stream to increase in size, as more bits are needed to encode each instruction. This simple rule is a fundamental principle of task-specific profiles used in our design. The instruction set is optimized at run-time to provide only necessary instructions to encode the task. By extending functionality of the application we increase the common alphabet, which has an impact on the code size. This principle also works if we remove functionality. Consequently, the compressed form changes as well. This is shown in Figure 6.13.

Another important feature of the scheme is that it re-encodes the entire stream from the beginning if some block has been added or removed. This allows achievement of a better CF. In Figure 6.14 we

6.4 Optimizing Chameleon VM's Code

FCS	SBTSP	Spanning tree	Sense and deliver	ID assignment
<i>independently, local alphabet^a [instr/bytes]</i>				
0	151/75.5	47/23.5	30/15.0	45/22.5
5	85/42.5	28/14.0	15/7.5	30/15.0
10	67/33.5	23/11.5	2/1.0	4/2.0
20	42/21.0	1/0.5	2/1.0	2/1.0
30	1/0.5	1/0.5	2/1.0	2/1.0
<i>independently, task-specific global alphabet^b [instr/bytes]</i>				
0	151/94.375	47/29.375	30/18.75	45/28.125
5	85/53.12	28/17.5	15/9.38	30/18.75
10	67/41.88	23/14.38	2/1.25	4/2.5
20	42/26.25	1/0.62	2/1.25	2/1.25
30	1/0.62	1/0.62	2/1.25	2/1.25
<i>independently, common global alphabet^c [instr/bytes]</i>				
0	151/113.25	47/35.25	30/22.5	45/33.75
5	85/63.75	28/21.0	15/11.25	30/22.5
10	67/50.25	23/17.25	2/1.5	4/3.0
20	42/31.50	1/0.75	2/1.5	2/1.5
30	1/0.75	1/0.75	2/1.5	2/1.5
<i>in the extending set, global alphabet^d [instr/bytes]</i>				
0	151/75.5	198/123.75	228/142.5	273/170.625
5	85/42.5	116/72.5	140/87.5	176/110.0
10	67/33.5	96/60.0	119/74.38	150/93.75
20	42/21.0	64/40.0	86/53.75	121/75.62
30	1/0.5	41/25.62	67/41.88	85/53.12
40	1/0.5	2/1.25	4/2.5	70/43.75
50	1/0.5	2/1.25	4/2.5	6/3.75

Table 6.9: Code Size Estimation: Data Collection Application

^aAssess each module using only its own opcode range independently from other modules in the set.

^bAssess each module using common opcode range (26 opcodes) but independently from other modules in the set.

^cAssess each module using common opcode range (50 opcodes) but independently from other modules in the set.

^dAssess each module using common opcode range within the extending set (capsules are being added up).

6. Experimental Setup

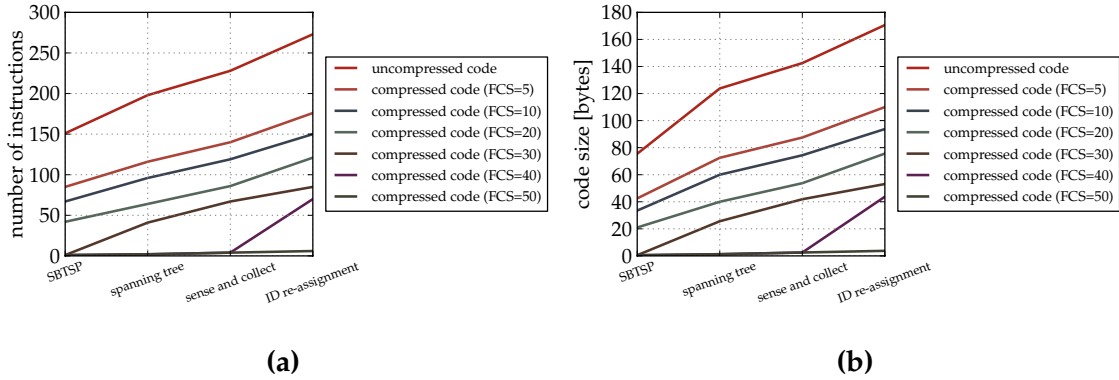


Figure 6.13: Data Collection Application: Code Build-Up

Capsule	Dissemination/Execution Frequency (times)
SBTSP	10 (nominal)
Spanning tree	1 (nominal/10)
Sense and deliver	0.1 (nominal/100)
ID assignment	0.01 (nominal/1000)

Table 6.10: Data Collection Application: Dissemination Frequency Setting

present the results for the set of 4 capsules: *SBTSP*, spanning tree, sense (Listing 6.9) and ID assignment. The collector capsule (Listing 6.10) is excluded as it statically resides on the sink node.

Different capsules building up the application are disseminated and executed with different frequencies. In order to estimate energy saving (see Figure 6.14d) we used the setting from Table 6.10. Furthermore, we assumed that the system is in the stable state. This means that we do not take into account energy spent on initial deployment of capsules or propagation of dictionary updates.

6.5 Optimizing FragletVM's Code

Chemical protocols as discussed in Section 2.4 are normally communication-abundant. Compared to packet-based traditional architectures, the number of transmissions required by CNP is much higher due to the dynamic nature of protocol design. CNP operate with concentrations rather than data fields. That is, some protocol parameters are represented by the number of molecules rather than a symbolic value.

6.5 Optimizing FragletVM's Code

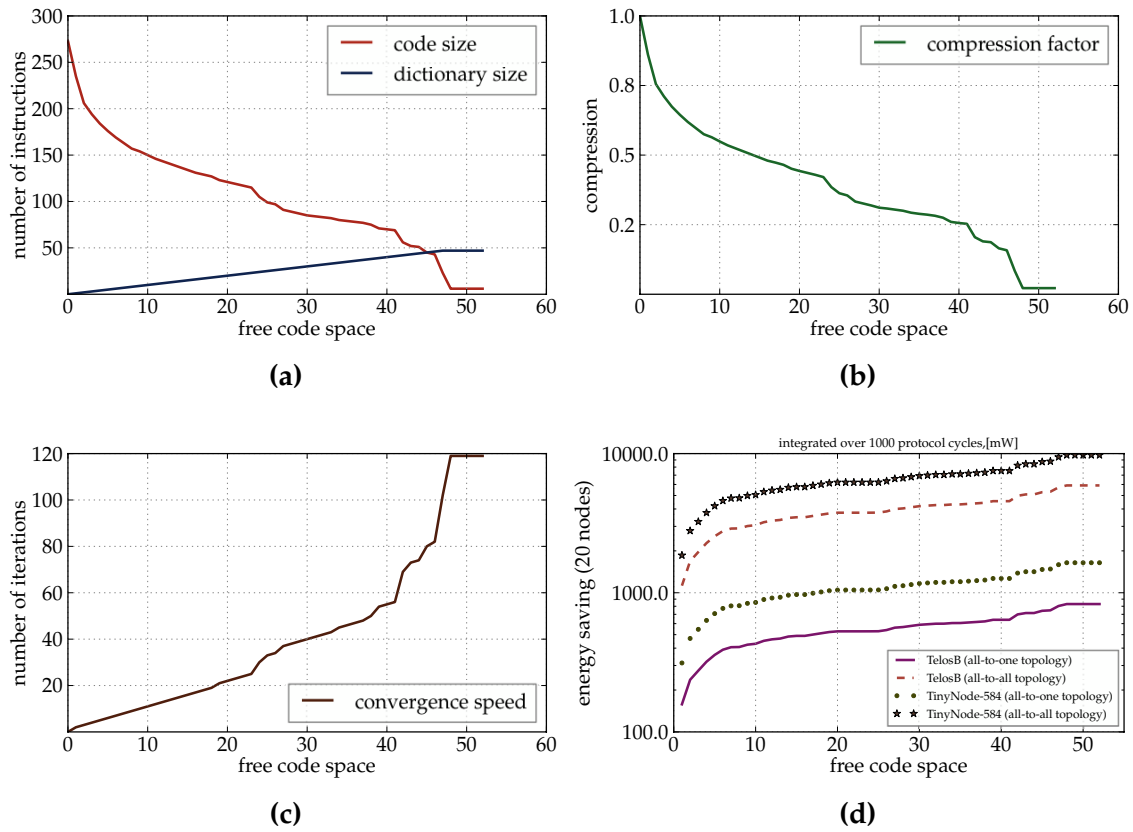


Figure 6.14: Code Compression Process: Data Collection Application

6. Experimental Setup

Transmissions (packets/molecules) are short but the transmission frequency is high. This model shows a potential use for our compression method. As an example we take the protocol developed in [Mey10] and re-encoded it for *FragletVM* (see Section 3.3): *Disperser*, a gossip-style aggregation protocol.

6.5.1 Disperser

Disperser was developed and analyzed for various topologies and configurations in detail in [Mey10]. *Disperser* is a CNP that calculates the average of distributed values. That is, it can be used as an aggregation protocol, which in turn is a typical task in many WSN applications. The computation of the average values is carried out by the dynamics of the distributed reaction system.

Lets demonstrate this with a simple example. The network consists of N nodes taking temperature measurements. We would like to know the average of the temperature value over the entire field (each node must learn the average value through providing its own measurement and interaction with other nodes). Using a more classic approach we would need to use a suitable routing protocol, one of the **aggregation functions** and some efficient data representation as described in [FRWZ07]. With CNP things become much simpler as this networking principle basically requires only three steps: 1) to represent a particular value as concentration of passive molecules (operands; in our case $30\text{ }^{\circ}\text{C}$ would be probably represented as 30 passive molecules),¹ 2) to introduce control molecules on each node, and 3) to choose a proper scheduling. Although the key factor to provide the right level of dynamics is to use the *Law of Mass Action (LoMA)* scheduling (probabilistic selection, [MT11]), our simplified scheduling mechanism for embedded implementations using quasi-random selection (see Section 3.3.4) provides an equivalent level of dynamics and has shown very similar results.

In [Mey10] the protocol has been proven to work for any topologies given that the number of control molecules is equal for each link in the network. Here we pick two implementations: 1) 4-nodes with fixed links, a simplified version we use to explain the protocol's design, and

¹Negative values could be represented with concentrations as well by defining an offset. It depends how we select the "temperature-concentration" conversion function.

2) a generalized version for unknown topologies which we apply our compression scheme to.

Logically the protocol consists of two fraglets (molecules): 1) control (active), and 2) operand (passive). To simplify things we assume that the control fraglet is manually pre-installed on each node.² First, we consider the 4-nodes setting example, code for which is shown in Listing 6.11.

```
1 # Distributes a load (average some value) equally over the set
2 # of 4 nodes
3
4 # Disperser program for node n1
5 f matchp X send n2 X # Send [X] to neighbor n2
6 p 100 # Inject 100 X molecules
7 f X
8
9 # Disperser program for node n2
10 f matchp X send n1 X # Send [X] to neighbor n1
11 f matchp X send n3 X # Send [X] to neighbor n3
12 f matchp X send n4 X # Send [X] to neighbor n4
13
14 # Disperser program for node n3
15 f matchp X send n2 X # Send [X] to neighbor n2
16 f matchp X send n4 X # Send [X] to neighbor n4
17
18 # Disperser program for node n4
19 f matchp X send n2 X # Send [X] to neighbor n2
20 f matchp X send n3 X # Send [X] to neighbor n3
```

Listing 6.11: Disperser Protocol: 4-nodes topology with fixed links

In this program we assume that *X* is a passive molecule representing a physical value (e.g., temperature). The more *X* that are generated locally by the node, the bigger the sensor reading is. Each node also contains a number of control molecules, which are proportional to the number of links the nodes has. This is required for the protocol to operate properly. The number of nodes and the topology are fixed and

²Viral propagation and deployment using fraglets is currently an unresolved issue since there is no “no-match” operation. To overcome this limitation a special form of control fraglets ([Quines](#)) should be used. These control loops regulate fraglets’ population by continuously generating new fraglets (but not allowing them to overcrowd the network) in exchange for those being consumed and re-generating themselves at the same time. More on Quines see [Mey10].

6. Experimental Setup

do not change over time. The topology which corresponds to the code in Listing 6.11 is shown in Figure 6.15.

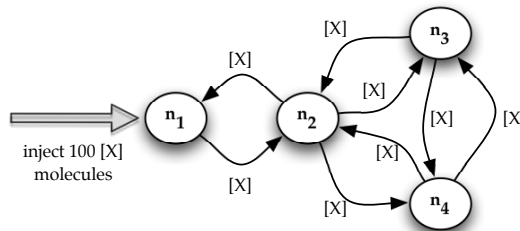


Figure 6.15: Disperser Protocol: 4-nodes topology with fixed links

Here we avoid a detailed analysis of the chemical reaction network for this protocol. The more interesting question is what type of message exchange this scheme triggers. We simulate the reading by injecting 100 X molecules on node n_1 (see Figure 6.15). These passive molecules start interacting with active fraglets on node n_1 which results in sending passive fraglets to n_2 . Their reaction continues and propagates further. Eventually, the entire network converts into a never-ending reaction flow where molecules react locally and generate new molecules, which are sent to the neighboring nodes and so on. The protocol guarantees that after a number of cycles and with some acceptable deviation the concentration of passive molecules X will be the same on each node. This process is illustrated in Figure 6.16.

Unfortunately from the compression point of view the above example is a no-go since nodes only exchange static information (synchronization tag X), no active code exchange is done. That is why we have to have a look at the more general version of the protocol which uses active messages (see Listing 6.12). In this version the protocol can automatically discover its network neighborhood. In fact, the neighbor discovery capability brings active code to the protocol. As before, for the very first time the fraglet must be installed on each node.

```
1 # Distributes a load (average some value) equally over the set
2 # of nodes
3
4 n i          # Disperser program for node i
```

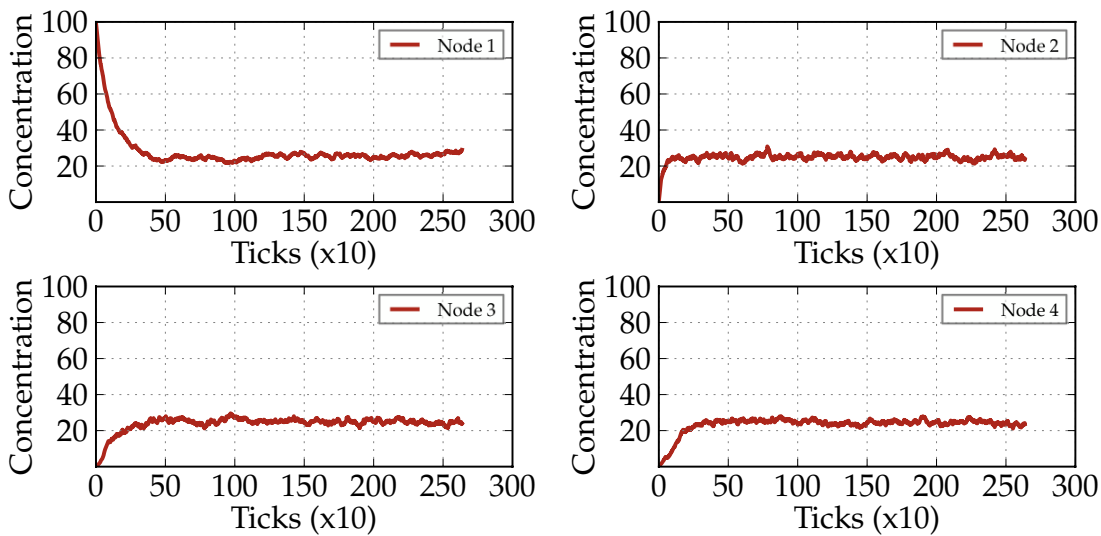


Figure 6.16: Disperser Protocol: Convergence

```

5 # Discover the neighbor and send [X] to him
6 f matchps X send all snodeid _ send i spush X match X send
7 p 100      # Inject 100 X molecules
8 f X

```

Listing 6.12: Disperser Protocol: unknown topology

Compared to the 4-nodes example from Listing 6.11 this implementation uses code exchange. The following communication steps are involved in each protocol cycle:

1. Each node i broadcasts the fraglet to all the neighbors: `snodeid _ send i spush X match X send`. The original active fraglet remains on the node, plus an extra copy of X is generated to compensate the consumed one.
2. Each neighbor j executes the received fraglet by appending the local node ID to its tail and sending it back to the originating node: `spush X match X send j`.
3. The returned fraglet is restructured to become a control fraglet: `match X send j X`.
4. The control fraglet reacts with a passive X molecule and sends it to the previously discovered neighbor j .

The neighbor discovery functionality (see Listing 6.13) can be extracted from the code and used as a building block for a routing pro-

6. Experimental Setup

toocol. Additionally, the source node ID can be obtained via `snodeid` operation at the beginning of the original fraglet instead of explicitly specifying it in the code. This would make code more portable.

```
1 f snodeid _ send all snodeid _ send
```

Listing 6.13: Neighbor Discovery

In order to show the experimental results of applying our compression scheme to the fraglets code stream we pick the network from Figure 6.15 and install the code from Listing 6.12 on it. Because of the dynamic and morphable nature of fraglets the resulting code traveling the network is normally much bigger than the original encoded stream. This explains the amount of code actually transmitted in each cycle as specified in Table 6.1. The number of transmissions from Table 6.4 is given by the fact that for each link we:

- transmit `snodeid _ send i spush X match X send`,
- receive `spush X match X send j`, and
- transmit `X` again.

The results are presented in Figure 6.17. The fraglet’s code shows a good response to the compression. This can be explained by a normally high rate of duplicates (patterns) in the transmitted code. As with the data collection application from Section 6.4.7 our energy estimations here are based on the fact that the system is in the stable state. Moreover, the results are valid only for the scheduling scheme used by *FragletVM*, a tick-based quasi-random scheduling. With the original *LoMA* which requires a non-discrete time base the results would be different as the transmission rate is not correlated with the clock domain (ticks).

The last but not the least thing to say is that we do not try to argue the potential application of fraglets in real WSN installations. This communication model is rather targeted at computationally rich environments. From the energy consumption point of view the chemical approach can hardly compete with classical energy-aware protocols. Here, we have only shown how a different approach to protocol design could benefit from our code compression methodology.

6.5 Optimizing FragletVM's Code

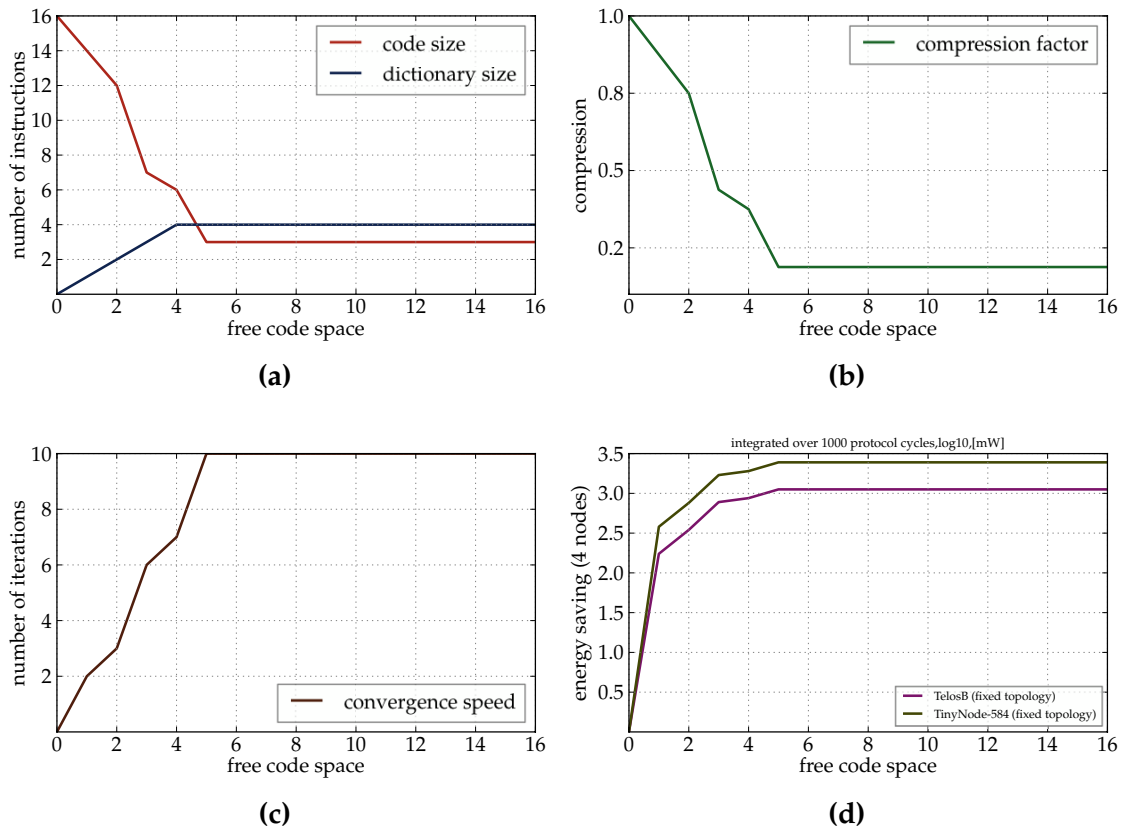


Figure 6.17: Code Compression Process: Disperser Protocol

6. Experimental Setup

Instruction	Description
<code>smove</code>	Performs a strong migration to a remote node. A strong migration does not affect the execution of the agent (the program counter, stack and heap are all maintained).
<code>wmove</code>	Performs a weak migration to a remote node. A weak migration does not save the agent execution state. The agent restarts from the beginning when it arrives.
<code>sclone</code>	Clones an agent to a remote node. The clone inherits all of the original agent's execution state.
<code>wclone</code>	Clones an agent to a remote node. The clone resumes running from the beginning.

Table 6.11: Agilla: Agent Manipulation Instructions

6.6 Optimizing Foreign Code: Fire Tracker in Agilla

In order to demonstrate how our method could be used with a third-party code we have picked a fire tracking application designed for *Agilla*, by its author, as a demo.

Agilla is based on *Maté* but uses its own customized instruction set. Initially we wanted to use the original *Maté* code but finding an example was a problem. *Maté* uses an underlying propagation layer. Hence, programs do not normally incorporate the mobile features. Although the code moves it does not have control over this. Nevertheless, *Maté* supports the `forw` instruction which is used to transmit a capsule to other nodes. First, the capsule is broadcast to network neighbors. These nodes install it and then call `forw` again when they execute the capsule's code. This way, the capsule is forwarded to their local neighbors; and so on. As reported in [LC02], "a capsule can also forward other installed capsules with the `forwo` ('forward other') instruction. This is useful if the desired program is composed of several capsules; a temporary clock capsule that forwards every capsule can be installed, then as each component capsule is installed it will be forwarded. Once the entire network has installed all of these capsules, the clock capsule can be replaced with a program to drive the application". *Agilla* operates with mobile agents and, therefore, provides a set of instructions to manipulate the installed entities as shown in Table 6.11.

Agilla also features instructions to manage local and remote tuple spaces (access to remote data sets). *Agilla* is also capable of managing information about neighbors in a high-level manner. These all are made

6.6 Optimizing Foreign Code: Fire Tracker in Agilla

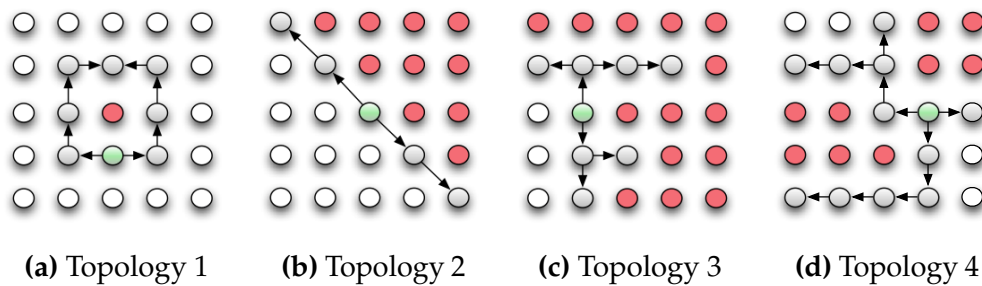


Figure 6.18: Fire Tracker Possible Topologies

possible by the background middleware-like activities carried out by the framework. For an application most operations remain transparent.

For our experiments we have borrowed a fire-tracking application from *Agilla's* distribution. The original implementation consists of two parts: 1) a fire tracking agent (tracks the fire), and 2) a fire agent (simulates the fire). We are interested only in the second part whose source code is shown in Listing B.1, Appendix B and the schematic algorithm in Figure B.1.

In this example, the network is built in a grid topology (5x5 nodes; see Table 6.5). After injection the fire tracker code detects and forms a perimeter around a fire, which has been previously modeled by the fire agent. The fire-tracking agent initially moves across the network looking for fire. Upon arrival to a node, it checks if any neighbors are on fire. In case it is true, the cloning happens onto nodes within two horizontal and vertical hops off the fire in the grid. The clones repeat this process as well. Eventually, a perimeter around the hotbed is formed. Fire detection agents continuously check the fire and proactively move to surround the hotbed. Some possible agent propagation scenarios are shown in Figure 6.18.¹²

We used the topologies in Figure 6.18 to model energy saving of the compressed code according to the propagation rules just mentioned.

As mentioned previously, *Agilla* instances are continuously exchanging a lot of system messages in order to maintain the backbone network

¹The fire agent is colored with green, area on fire with red, and the agent propagation path with gray.

²This example is borrowed from http://mobilab.cse.wustl.edu/projects/agilla/Examples/single_agent_fire_detection/index.html.

6. Experimental Setup

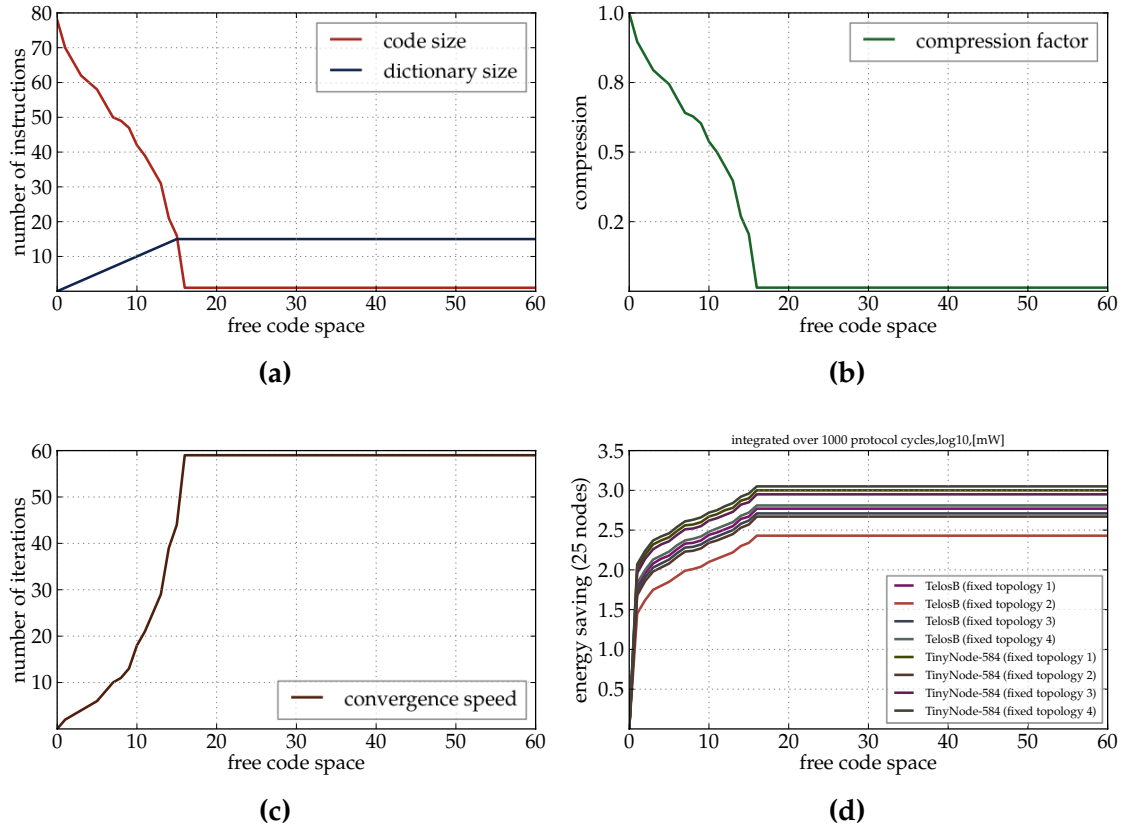


Figure 6.19: Code Compression Process: Fire Tracker

ready for agent migration. Moreover, the *Agilla*'s agent migration process is divided into phases: heap, stack, code and tuple spaces are transmitted separately. We do not take all these into account, only pure code transmissions. The results of applying code compression to the fire tracker agent's code are shown in Figure 6.19. Energy estimations are made using the scenario from Figure 6.18.

Agilla is based on *Maté* VM whose architecture and, therefore, code execution style is slightly different from the one used in *ChameleonVM*. We had to pre-modify the code in Listing B.1 to be able to feed it to our compressor: intermediate operands (numeric values) were removed. In contrast to *ChameleonVM*, *Agilla* does not use the concept of code merger/split. An entire agent is always transmitted and code compression is applied to the entire code block.

As can be seen from the Figure 6.19 the *Agilla*'s code responds to code compression well, very similar to the *ChameleonVM*'s code. The

reason is that bytecode streams of both have a very similar structure, though the binary representations differ. As a result the agent's size can be reduced significantly.

6.7 Final Considerations

At the end we would like to outline some common trends on compression of various code streams presented in this chapter. The level of success in compressing a code stream depends on many factors including:

- stream type,
- stream size,
- fragmentation level, and
- patterns.

The above characteristics are given by the source stream and normally cannot be changed, although some sort of pre-arrangement of the source code (**code re-factoring**) can be done before applying compression (see Sections 4.1 and 4.3). Parameters of the compression algorithm have a high impact on the process as well. In our experiments we have mainly explored the influence of FCS, the number of spare opcodes in the entire ISA. Table 6.12 shows the original dictionary size for each application and the point of convergence, the number of newly introduced instructions which leads to more compressed code representation. Pushing FCS beyond this point does not give any further gain in compression. As can be seen there is no common law. Some code streams require FCS to be as big as at least the original dictionary; others need it to be even twice the size to converge. The last statement is mainly true for bigger streams with relatively large original dictionaries.

Table 6.13 and Figure 6.20 show the results of the online compression for all test applications presented earlier. In most cases we can see that the CF has a clear exponential decay characteristic in the most part of the curve. Most characteristics also have a cut at the end where the algorithm shows the highest convergence. The larger code streams take more iteration to converge as the algorithm processes only a pair of instructions in each cycle.

6. Experimental Setup

Application	Dictionary Size	Point of Convergence
<i>ChameleonVM</i>		
"Hello World!"	4	4
Route discovery	5	11
Spanning tree	11	12
Network Size Estimator	5	4
ID re-assignment	13	13
SBTSP	15	26
Data Collection Application	26	47
<i>FragletVM</i>		
Disperser	8	5
<i>Agilla</i>		
Fire Tracker	30	16

Table 6.12: Dictionary Size and Point of Convergence of Test Applications

Application	FCS=3	FCS=5	FCS=10	FCS=20	FCS=30	FCS=40	FCS=50
<i>ChameleonVM</i>							
"Hello World!"	37.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
Route discovery	51.2%	37.2%	14.0%	2.3%	2.3%	2.3%	2.3%
Spanning tree	68.1%	59.6%	48.9%	2.1%	2.1%	2.1%	2.1%
Network Size Estimator	60.0%	20.0%	20.0%	20.0%	20.0%	20.0%	20.0%
ID re-assignment	75.6%	66.7%	24.4%	4.4%	4.4%	4.4%	4.4%
SBTSP	64.9%	56.3%	44.4%	27.8%	0.7%	0.7%	0.7%
Data Collection Application	71.1%	64.5%	54.9%	44.3%	31.1%	25.6%	2.2%
<i>FragletVM</i>							
Disperser	43.8%	18.8%	18.8%	18.8%	18.8%	18.8%	18.8%
<i>Agilla</i>							
Fire Tracker	79.5%	74.4%	53.8%	1.3%	1.3%	1.3%	1.3%

Table 6.13: Compression Factor of Test Applications

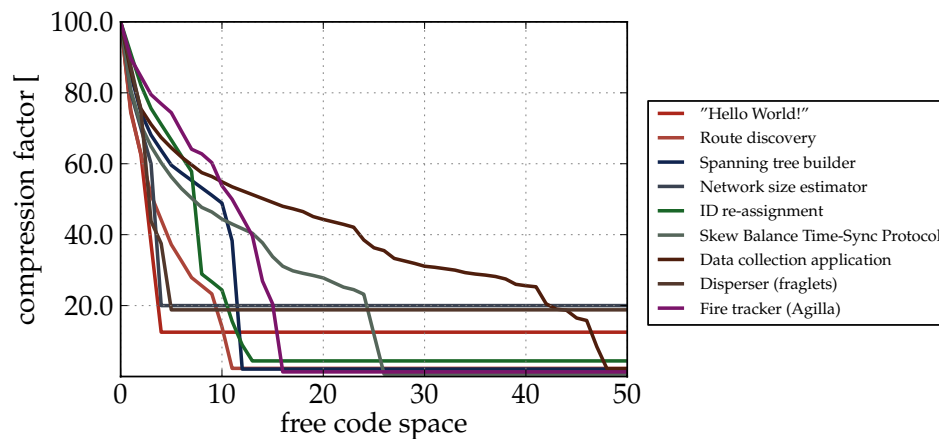


Figure 6.20: Compression Factor of Test Applications

Our energy estimations should be seen only as approximations as many other factors such as the topology can potentially influence the power consumption. In our model we have tried to cover the most common situations.

Besides all the positive aspects caused by applying code compression a number of negative sides can be highlighted. By extracting semantics from the traveling code and putting it in the on-board dictionary we occupy memory resources of the node. This has not been a problem for our test cases but extremely big dictionary sizes might become a significant constraint.

Some complex protocols might require initial packets to be of a relatively big size, which in turn might not fit into the platforms-specific packet format. In this case, we would have to split code into pieces, disseminate the entire code in a form of several packets (currently, this is already supported by *ChameleonVM*), assemble them back together on the node, run the code compression up the point where code can fit into payload limit and only after that initiate protocol execution. Compression could then be continued at run-time.

Another thing to mention is that our compression is a rather leisurely process. Results can be seen only after many cycles. In this context, frequent re-configuration activities might not be efficient at the end. Re-programming requires energy to disseminate program updates and compression itself requires energy to announce dictionary updates to

6. Experimental Setup

the rest of the network. This trade-off must be assessed beforehand on a case-by-case basis.

6.8 Summary

In this chapter, we have carried out a set of experiments on code optimization techniques previously described in Chapters 4 and 5 using three types of code streams with different properties. For these experiments we tried to pick examples, which can find a use in real-world scenarios. We have proved that our method shows comparable performance for all analyzed code stream. Moreover, it outperforms the-best-in-class existing solutions. Besides evaluation of the code size related properties we have also established the correlation with energy consumption for long-term operations. We have found out that our method can potentially provide big energy savings for a system. The trade-off between code reduction and computational complexity accompanied by communication overhead still allows the method to show positive effect on the system characteristics.

7

Discussion and Future Directions

In the previous chapters, we have presented the set of methods for dynamic code morphing in network embedded systems (ES). Many questions remain open though. These questions point at the design level as well as at the implementation details. In this chapter, we first give a formal evaluation of the proposed method and discuss its possible applications. Then, we try to summarize all open issues and find some directions of how to address them in the future. We also highlight several conceptual offshoots, which may stimulate further research in the area. We show how our results could be used to achieve that, and what benefits or drawbacks we should expect.

7.1 Formal Evaluation of the Proposed Method

In this work, we have investigated the research problem of bringing program code representation (encoding) to a form, which is most optimal in the current context, for the current task being performed. It belongs to the class of NP-problems like many types of optimization tasks. This means there is neither optimal nor fast solution known. The time and computational resources required to locate such a solution increases very quickly as the size of the problem grows. In our case, this would mean growing alphabet size, total program size and free code space. In Section 5.9 we have shown how these algorithm parameters affect the convergence speed. In the world of embedded devices, where resources are extremely limited, this becomes even a bigger issue. Traditionally, NP-complete problems are often addressed by using approximation algorithms. We have adopted this technique in our research too. We do not try to find the most optimal representation but rather to push the system towards it at each iteration of the algorithm. The local solutions made may not be optimal and could even make the system drift away temporarily. But in the long term, the behavior must show a positive tendency. This behavior was explained in Section 5.3. Our method satisfies these two assumptions.

From the system architecture point of view, the method employs some characteristics of intelligent decision-making. Although its capabilities are limited, as the available resources do not allow building comprehensive algorithmic structures. The system is naturally adaptive and capable of doing “switch”-logic. This means it can adapt to changes in the environment that is software re-configuration in our case.

7.2 Application Fields

The code morphing techniques described in this work are of a broader nature than just network ES. Moreover, generally code morphing is not limited to the network systems only. Task-specific code optimization can be applied to the local code too if memory/CPU constraints are too tight. Such optimization might potentially save energy. However, the effect of local compression might not be very big. In this case, traditional compiler- and linker-level optimization tricks would probably be more efficient.

So far in this work we have mainly concentrated on ES and WSN in particular. Two recent network-related paradigms, discussed below, could potentially benefit from using code morphing as well.

NoC (Network-on-a-Chip): This is a hardware-oriented approach to designing the communication layer between IP cores in a SoC (System-on-Chip), a sub-class of VLSI systems. In [SSM⁺01] the authors also referred to it as “the layered-stack approach to the design of the on-chip intercore communications”. NoC applies networking principles and methods to on-board (on-chip) communication. This allows for overcoming the limitations of traditional bus/link interconnections. According to [NK08], in a NoC system, modules (e.g., processor soft-cores, memory units, specialized IP blocks) exchange data chunks using network principles. It is made up of multiple point-to-point data lines (de-)multiplexed by switches (a.k.a., routers). The switches make routing decisions so that message exchange can be established between any source and any destination modules over several lines. A NoC shows many similarities with a traditional telecommunications network that uses bit-packet switching over multiple interconnected links. Proposals utilizing circuit-switching techniques exist too. NoC improves the scalability of SoC, as well as the power efficiency.

The NoC paradigm does not specify the topology of the on-chip network. Currently, there is a rising wave of research on synthesis of application-specific NoC topologies. We believe that the next stage of this process would be integration of code exchange between modules into such on-chip architectures. Looking further ahead we can see the blending between local and inter-node communication channels and creation of a global transparent communication bus between nodes and modules inside the nodes. Applying our code compression method to this communication model may be beneficial, as it would provide higher bandwidth for such a bus.

Dynamic stack composition: This is another emerging field of research where an optimal network stack structure is considered as a target. Compared to the traditional static stacks, the process of dynamic composition allows adapting the stack to the continuously changing external factors; e.g., traffic conditions, channel quality, functional redundancy/shortcoming. Various approaches have been proposed. [KHM⁺08] look at the problem from the traditional point of view by

7. Discussion and Future Directions

creating modularity at all levels of the stack architecture and providing a rebinding property for dynamic re-configuration. In [RMYT08] the authors propose to make the system run “continuous experiments with alternative protocols online, in parallel as well as serially, in order to automatically select those that best match the application’s needs under the current network conditions”. The work of [IT10] brings the above two ideas together. The highly decomposed protocol stacks are built at run-time from multiple independent functional blocks (e.g., FEC modules of different types, network and transport layers with different properties, etc.) by learning about the optimal configuration from the environment using genetic algorithms.

The stack composition can be seen as a special case of our task-oriented network profiling technique. The difference is that stack composition looks only at the communication level whereas our concept is more general and can be used to describe any type of in-network activities. In this sense, code compression has every chance to be useful for stack composition as well. That is, after bringing the stack to the desired, optimal configuration of modules the further optimization at the encoding level could be made using our method.

Emulation mode: *ChameleonVM* presented in Section 3.2 can be used outside the code morphing framework. Because of its dynamically tunable ISA it can be used to emulate other existing VM. We have not investigated this though. Potentially, many problems will rise in the emulation mode regarding task scheduling, memory usage, etc. Moreover, if the semantics of some instruction cannot be emulated using basic ISA provided by *ChameleonVM* then it should be added to the system at compile-time which contradicts the main principle of our framework’s design.

The presented examples show that the proposed method can be useful in other domain different from WSN where optimal program representation is important.

7.3 Open Questions

In our research, we have not answered many questions regarding the architecture and behavior of our re-tasking and compression method, and the tools. On one hand, these issues are not critical for development of working implementations. On the other hand, there are some

situations in which our system has not been analyzed and tested yet. The further understanding is necessary to make the picture complete. We discuss these below.

Auxiliary data compression: Our compression scheme is code-oriented. It does not deal with the data part of a program stream. That is why in *ChameleonVM* code and data are separated at the system level (see Section 3.2.2). An improvement would be to extend the design and to also apply compression to the data part. This extra compression step could involve one of the data-oriented methods (e.g., *LZW* or some other discussed in Section 4.2). It is not obvious if our method could be applied here or not. Generally, data have much higher entropy. However, in the case of capsules there are not many immediate operands in the stream, mostly static addresses and system variables which opcodes are fixed. Therefore, entropy of such a stream should not be too high. More analysis is needed here. By complementing code compression with data compression, we could achieve even better results.

Packet size: Network ES feature cheap, short-range radio components. Such systems dictate a certain level of limitation on packet transmission size. Normally, a packet in such systems can be of variable length up to a fixed maximum size.

For example, *CC2420* packet¹ radio chip used on many popular WSN platforms like *MicaZ* and *TelosB* has the maximum size of a packet of 128 bytes² including its headers and CRC, which is demanded by the IEEE 802.15.4 PHY specification. Increasing the packet size will increase data throughput and RAM consumption, but will also increase the probability that interference will cause the packet to be destroyed and need to be re-transmitted.

On older platforms like *Mica2* which used *CC1000* radio chip some developers have reported successful transmissions with a size of over 200 bytes. This was caused by *CC1000* which is a bit radio, i.e., each bit is sent directly by the MCU through the radio and out on the air. This means a packet size can be increased up to 256 (the length field is only 8 bits).

¹Packet radio chip buffers the entire packet in the radio hardware itself before sending it over the air. This means that the number of payload bytes is limited to the size of the hardware buffer.

²The default value in *TinyOS 2.x* is 29+1 bytes.

7. Discussion and Future Directions

Normally, these limitations are not critical for protocols based on data exchange. The situation changes dramatically when we switch to mobile code implementations. As it was shown in Table 6.1, Section 6.1 some applications can easily produce packets exceeding the limits stated above. This would mean that such protocols are not operational in the first place, before they are compressed. We have solved this problem using the following trick:

1. Split code in multiple chunks.
2. Disseminate the chunks.
3. Re-assemble code again on a node.
4. Run compression without running the protocol itself up to the point when the packet size fits the platform limit.
5. Continue to run protocol and compression in parallel.

Obviously, this greatly limits the use of the method but currently we have no better solution.

Fully distributed compression: In this setting, briefly mentioned in Section 5.7, different nodes holding different programs run a sort of cooperative compression on a heterogeneous code set. There are two main questions raised by this process: 1) how to schedule compression iterations between multiple nodes, e.g., Round-Robin or priority-based according to the code size on each node,¹ and 2) how to coordinate information exchange between those nodes.² As can be seen, it is not a trivial scenario. Alternatively, in order to avoid a very complicated coordination between nodes, we could just combine multiple program images on one node, run the compression on the entire set and then disseminate the final result to the others. This case would be a reduced form of that described in Section 5.6.

Auxiliary Huffman-coding: As it was mentioned in Section 5.4.11 *Huffman*-coding could be used on top of our code compression scheme to better utilize the code space. As is well known, *Huffman*-coding does not reduce the size, i.e., it does not actually compress. It rather

¹Obviously, the node having more code can make better compression decisions since there is more statistics available.

²Each node after making a decision must announce it to the rest of the network. The others can agree or not if the new rule contradicts any previous assignments or if they are aware of a better decision. The change can be applied if only all nodes have confirmed it.

allows encoding more frequent codes with fewer bits. Our scheme could definitely benefit from it.

Corrupted code: Mobile code can be corrupted during transmissions. In most WSN OS there is a simple CRC check on each packet. If the CRC check fails the packet is simply dropped and requested for re-transmission. For time critical applications this might not be the case. In Section 4.5 we discussed how the robustness of mobile code could be improved. This should be investigated further. Having the code robust at the instruction and structural levels would make possible a full-grown self-recovery feature which still remains a hot topic in mobile code based applications.

Security: What is the definition of malicious code in ES? Can we trust the code we receive? How can we check code validity? These are the questions we have not touched in this work. There has been a lot of research done [BN08] on the protection of sensing and disseminating data in WSN from various types of attacks. These solutions include hardware- and software-based security architectures. Both could potentially be used to provide code validation in our system.

7.4 Future Directions

Besides the open questions from Section 7.3 we would like to emphasize some ideas which could bring this work up to the next research level. These ideas can mainly be grouped around two things: the tools we use and the compression method itself. We first take a look at the tools.

Sponge protocols: In the current implementation capsules or fraglets can only be combined with each other according to the pre-defined rules (`merge` instruction in *ChameleonVM*, or `match-es` in *FragletVM*) within the code only. This gives a very pre-deterministic behavior to the system and in terms of stack composition an expected stack structure. This is not bad but it requires different protocol parts to be aware of connection points between each other. In our opinion this could be done in a more elegant way by using the concept of prioritization, or attraction (gravity) levels between parts of the code. For example, in case of high noise inside a communication channel this would increase the attraction level of Reed-Solomon FEC module to the network-layer module rather than simple CRC-check. This would

7. Discussion and Future Directions

also allow start to build network protocols in a more probabilistic and proactive fashion.

In contrast to the **stack composition** mentioned in Section 1.5.3 with sponge protocols we assume finer granularity. Sponge protocols are supposed to be constructed out of single instructions. Those instructions may have very complex semantics though.

Another distinguishing feature of sponge protocols¹ would be an ability for self-organization according to some sort of specification listing a set of modules spread over the network. Current stack composition approaches assume that all modules are available locally. Sponge protocols would be able to detect which parts are missing, locate them in the network, deliver to a node and include them in the composition process. The concept of attraction levels would help here again as we would have to look only for the pieces we like at that moment based on the current conditions and express this feeling somehow to the rest of the network. The matching pieces would be gravitated towards us. Ideally, this process would be fully automated.

The model could be extended to physical nodes as well: a node would attract molecules with some properties according to the assigned role for this node. Taking a materially minded example, this would make a sink re-assignment (see Figure 7.1) in WSN an easy task. The node with a new role “sink” will attract capsules specific to this role, e.g., “collect”, “send to Internet”, etc. Those capsules will migrate from the old sink to the new one to satisfy the new role. Inversely, capsules for the role “regular node” will migrate (or replicate from the neighbors) to the old sink.

Distributed labels: This concept somehow extends the idea of capsules to building applications spread over multiple nodes. The application itself remains transparent at the code level as shown in Figure 7.2.

The program execution flow would switch between physical nodes as it goes on. Execution control instructions would force the system to execute code on different nodes. For instance, as shown in Figure 7.2, `jmp L2` calling on node 1 would pass control to node 2 at the position

¹Here we draw the analogy with animals of the phylum *Porifera* which “are known for regenerating from fragments that are broken off, although this only works if the fragments include the right types of cells”. This enables “sponges that have been squeezed through a fine cloth to regenerate”. [information taken from the Wikipedia article located at <http://en.wikipedia.org/wiki/Sponge>]

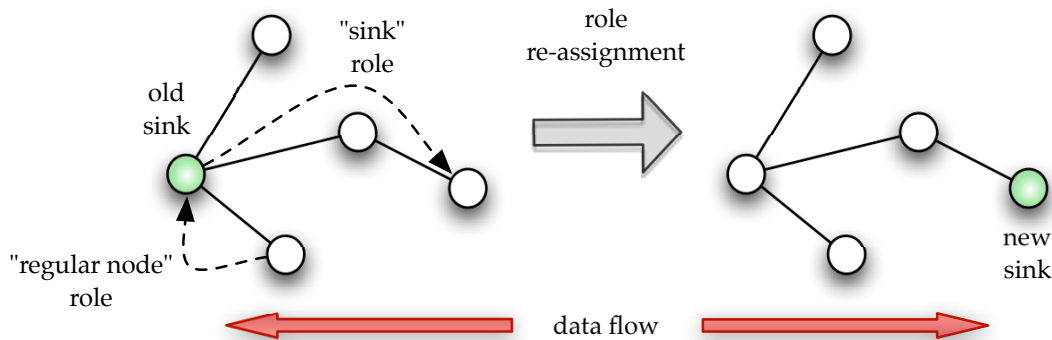


Figure 7.1: Sink Role Re-Assignment

marked with $L2$. This execution method would probably involve some non-trivial code localization tricks.

In-network data fusion: It is hard to believe but network steering still remains mainly manned. The concept of capsules could easily be adapted to provide a certain level of automatic control over network activities. We have shown one simple example with temporary traffic redirection in Section 6.4.2. This would bring us to the idea of “smart agents” sitting inside the network, observing the situation and taking some corrective actions according to the situation. Those actions could even be taken as a prophylaxis.

As an example, we can take the *X-Sense* (former *PermaSense*) project mentioned in Section 1.4. In [BBF⁺11] the authors state that “the data retrieved from the network will still contain artifacts”. This happens because of many factors: clocks drifts, packet duplication, packet loss, system misbehavior, etc. These factors are normally aggravated by harsh environmental conditions some systems, like WSN, are working in. In *X-Sense*, in order to meet qualitative and quantitative requirements, i.e., data integrity, a lot of data post-processing steps are involved in the reception and database side. These steps include: data cleaning/ordering, data conversion/mapping, domain user processing (filtering, aggregation). All this processing is done post-hoc. Our belief is that it could be done with smart agents using an injection of processing rules (meta-data) and units (agents) into the network. The job would be done in place and in time.

7. Discussion and Future Directions

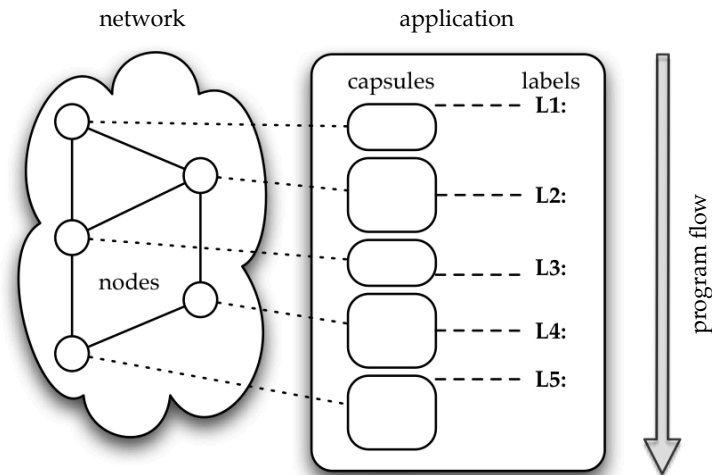


Figure 7.2: Distributed Labels

Identity management: As a side effect we could have information collected about existing profiles. This information could be used for the purpose of routing, address resolution and manipulating multiple name spaces within the network. This way, nodes performing a common task could be addressed collectively. Communication between nodes in the same profile could be optimized too, e.g., to use a short address version between “colleges”. This can be considered as a step towards “cognitive” networks.

Additionally, we have the following ideas in mind regarding our compression method:

Hardware acceleration: In order to improve performance for extremely time critical applications an FPGA-accelerator could be used for code (de-)compression. Here we can refer to the recent work [M10] on data stream processing on FPGA which adapts the WSN-based query system introduced by *SwissQM* [MAK07] to the world of FPGA. The author even creates a query-to-hardware compiler to translate SQL-like queries into FPGA primitives.

In our case the system would look like the one shown in Figure 7.3. The hardware accelerator would continuously grab pieces of code from the code pool, (de-)compress them according to the algorithm presented in Section 5.4, drop the (de-)compressed code back to the pool and update the dictionary. The in-memory code pool would require

some scheduling mechanism implemented by the FPGA side to decide which parts of code is to (de-)compress and when. Alternatively, the entire system could be organized inside a FPGA chip. In fact, in this case the VM would become a soft-core processor.

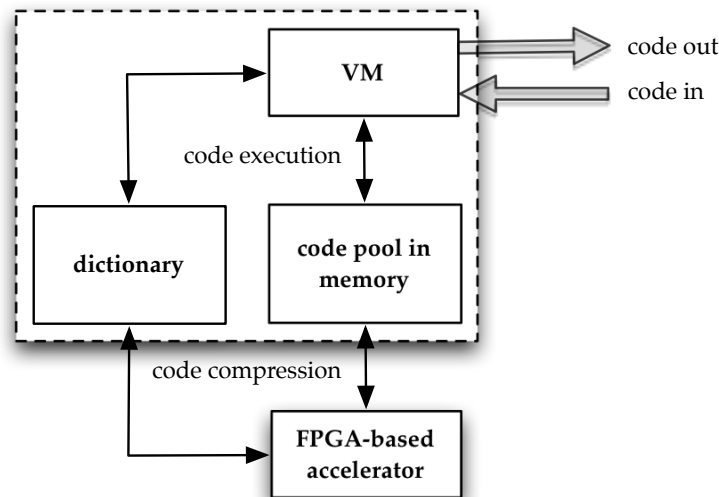


Figure 7.3: Hardware Acceleration for Code Compression

Deliverable semantics: The last thing we would like to mention in this section refers to the fact that so far we have considered code morphing based on pre-defined semantics only. Our ideal environment, e.g., based on *ChameleonVM*, would consist of an abstract execution engine only. This engine would initially know nothing about ISA, instruction specifications or how it should execute them. This would be coming along with a new code. The execution engine would be able to change the semantic rules so that when the code leaves the node and moves somewhere else it would carry the new semantics along. This process is shown in Figure 7.4.

The method of delivering semantics along with the code may have a huge implication on protocol design in the future. In this case, not only code but also code semantics can be re-written at run-time while the code is moving. This process will bring the system to a totally new level of flexibility. However, performance (semantics processing) and

7. Discussion and Future Directions

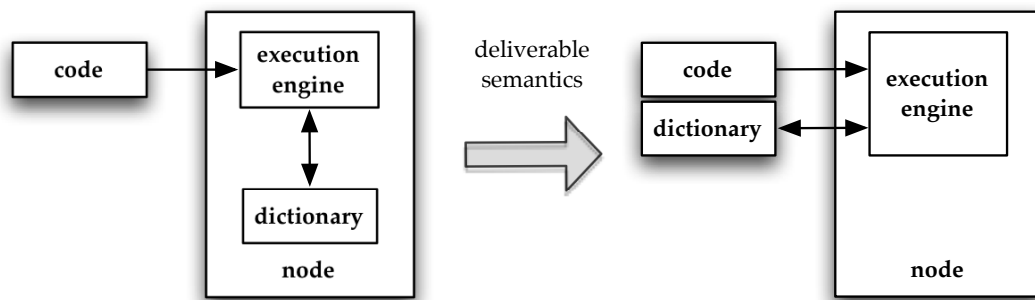


Figure 7.4: Deliverable Semantics

resource consuming (semantics delivery) issues might become critical for the system operation in this case.

7.5 Summary

During the discussion we have tried to assess the scientific significance of the proposed models and methods. This helped us to find the new potential application domains for the obtained results. It turned out that applicability is surprisingly wide, many fields can potentially benefit from using our work. We have found a number of open questions, which have to be answered first though. At the end we have outlined some future directions for our work which give us plenty of research opportunities: from using self-designed mobile code to perform tasks like in-network management and distributed computations to prototyping the proposed compression method in hardware or extending it with new architectural principles.

8

Conclusions

The final chapter of this work summarizes and assesses all our findings made so far. We try to clarify how the initial motivation and the stated goals from Chapter 1 correlate with what has been achieved at the end. Once again we are having a retrospective look at our contributions discussed briefly in Section 1.6 and re-estimate them from the perspective of the finished work.

8. Conclusions

At the beginning of this work in Chapter 1, we stated several goals which we would like to pursue in our research along with some premises. We are pleased to see that most of them have lived up to our expectations.

Tools for carrying out network tasks based on mobile code: In the center of our research we put the idea of **run-time task optimization in network ES**. In this context a task may come in different forms: a protocol stack (native code), an execution environment, or a middleware script (bytecode), a user-level abstraction (query). Initially, we did not want to limit our research to some specific form of task representation. Later we realized that certain forms have a significant advantage over the others. For instance, the bytecode form normally provides more possibilities to manipulate the code and modify it. This is in contrast to the machinery (binary) form or human-like query languages. Bytecode gives the necessary level of abstraction from hardware and OS-level complexity, at the same time keeping full, detailed control over program execution flow. This mix of simple programming abstractions and fine granularity of the resulting code led us to focus on this form of task encoding.¹ Having made this decision we built two **mobile bytecode execution environments**. The first one, *ChameleonVM*, employs the ideas of classic active networking and mobile code, the other, *FragletVM*, is an adaptation of the execution environment originally invented to support protocol design based on chemical reactions. Both solutions were tailored for use in the context of embedded network systems, WSN in particular. We were surprised how many minor questions arise, when well-established methodologies, developed with a different mindset, are being adapted to the resource-limited contexts.

Based on the gained experience we can say that using mobile code in embedded applications is feasible with some restrictions dictated by the limited available resources on most embedded platforms. Therefore, the two basic principles for such systems are: 1) code must meet the size requirements (storage/transmission), and 2) code must meet the time requirements (in-time execution). Moreover, using mobile code in a system would require a total rethink of how protocols are designed. The design of a mobile version of well-known protocols has been another interesting experience.

¹Moreover, two other forms can be translated to the bytecode easily.

A model of building task-specific WSN configurations: The developed mobile code frameworks allowed us to do system **re-tasking at run-time**. We later used it as a base for the newly created model of **building task-specific WSN configurations**. We truly believe that the future of continuously growing and complicating WSN technology lies in the area of building specific network configurations by customization of available common abstractions. We call this process **task-specific profiling**. As some requirements may not be known in advance, at design-time, the system must be able to adapt, to profile itself at run-time, when it is probably already deployed and up and running. The decision on what is needed for proper system operation and which parts of the software can be thrown out is a big challenge. We have not completely covered this topic in our research; we have rather proposed a set of policies and tools for doing it. In our examples, the decision-making is still done manually. We think that our model can satisfy most system scenarios available today. The recent research efforts show that deployment (programming and operating) of a network configuration across a wide variety of (unforeseen) circumstances is one of the hottest topics in research and development of ES. However, most projects still follow the traditional way of trying to predict and pre-define system behavior at design-time for as many situations as possible. From this point of view, our approach is different as we first deploy the system and only after that we learn the current conditions. The system is meant to be tweaked at run-time. We think this fundamental difference is the first step to creating a new generation of self-configuring and self-regulating ES.

Methods for dynamic code morphing: When the task-specific configuration is ready the next question arises. Since many unnecessary components are out at this stage, would it be possible to change the task representation (encoding) in a way to better utilize the released resources? And, if yes, how should we optimize the representation to achieve better results? The reason for doing that is simply defined by the context we are working in. In most ES energy budget is the main factor, which defines the system's lifetime. This budget is important for being able to provide unmanned operation over extended periods of time. In turn, the amount of information we have to exchange defines how much power we have to spend on transmissions. We have studied

8. Conclusions

various methods of optimizing the task encoding, from structural to algorithmic, and even bit-level coding. We were more interested in not how a task is programmed but rather how it is encoded. This led us to creating the next model as a part of our research plan which we called **dynamic code optimization** (or **dynamic code morphing**). This model includes various optimization steps from **code shrinking** by cutting off unneeded pieces as the program executes to **code compression** at the instruction level. As the experiments showed the system responds to the new model positively, code reduction was significant.

Mobile code robustness: Code morphing may not be necessarily intentional. Code may be corrupted due to an imperfect execution environment. This has a higher chance to happen if we deal with mobile code as communication channels normally give an extra bag of errors. We tried to analyze how **code robustness** could be increased in this case to prevent malicious code execution. We have not come up with an ultimate solution but rather have outlined the directions in which robustness can be improved. This highly depends on the application domain and many other factors (hardware support, software configuration, etc.).

A model of online code compression: As our research progressed we realized that code compression seems to be a prominent part of the optimization process. We have developed a model and a framework for a so-called **online code compression**. The last became an integral part of two of our mobile code execution environments mentioned earlier. We have analyzed the model in detail for various types of code streams and various network settings. Our compression scheme has shown higher compression rates compared with the existing solutions as we demonstrated with different models of mobile code: traditional active networking, chemical networking protocols and a model based on using mobile agents. We were glad to see that our model can support different code streams, which makes it possible to apply to many application domains.

The emerging trends in ES and applications mainly include adding in “smartness” in a form of new features and functions. But this “smartness” plays against the other principles of the embedded world:

low-power, small physical form factor/footprint, low radiation/emission/thermal dissipation, ruggedness in design, and robustness in operation. That is why the design of ES must be kept energy-aware in the newly appearing contexts to allow final products to adapt their energy resources to the continuously changing demands at run-time. In this work, we have examined the dynamic code morphing methods, which we hope will assist in achieving this goal.

List of Figures

1.1	Typical Hardware Architecture of a WSN Mote	3
1.2	PermaDAQ SIB Module	4
1.3	Typical WSN System Architecture	5
1.4	Software Suite on a Typical WSN Mote	9
1.5	PermaSense Installations Sites in the Swiss Alps	13
1.6	Virtual Segments and Network Profiles	15
3.1	Code Dissemination	37
3.2	Profiles Building and Switching	39
3.3	Network Level within Node’s Software Structure	45
3.4	A Network Protocol Implementation: Static Code and Packet Exchange vs Mobile Code	46
3.5	ChameleonVM: System Architecture	51
3.6	ChameleonVM: Functional Planes and Data Flows	52
3.7	ChameleonVM: Memory Structure	56
3.8	ChameleonVM: Capsule Structure	58
3.9	ChameleonVM: Merge Operation	59
3.10	ChameleonVM: Clone Operation	60
3.11	ChameleonVM: Split Operation	61
3.12	ChameleonVM: Instruction Data and Code Sources	64
3.13	ChameleonVM: Communication Model Between Nodes	70
3.14	ChameleonVM: Dictionary Structure	71
3.15	ChameleonVM: Example of Compound Dictionary Entry	72
3.16	ChameleonVM: “add new instruction” command	73
3.17	ChameleonVM: “delete existing instruction” command	73
3.18	ChameleonVM: “update existing instruction” command	74

List of Figures

3.19	ChameleonVM: “enable/disable protection from changes” command	74
3.20	ChameleonVM: Propagation of Dictionary Updates . .	75
3.21	FragletVM: Vessels	79
3.22	FragletVM: Fraglet Structure	83
3.23	FragletVM: Program Execution Look-Up Table	84
3.24	FragletVM: Sub-Vessels	85
3.25	Traditional vs CNP Programming Models	86
3.26	FragletVM: Original Selective Algorithm	88
3.27	FragletVM: Customized Selective Algorithm	89
3.28	FragletVM: Sending Fraglets Between Nodes	90
3.29	FragletVM: Forwarding Fraglets	91
3.30	FragletVM: Communication Model Between Nodes . .	92
3.31	FragletVM: Exchange of Dictionary Updates	93
4.1	“Case”-Branching vs “If”-Branching	98
4.2	Code Adaptation Process	112
4.3	Stack Operations for the “Sense-Calculate-Mean” Example	114
4.4	4-bit Hamming Binary Hypercube	120
4.5	Instruction Pair Occurrence Rate	122
5.1	Characteristic of Online Compression in Time and Re- source Domains	136
5.2	Online Compression Scheme: System Architecture . . .	137
5.3	Online Compression Scheme: Sliding Window	138
5.4	Online Compression Scheme: Compression Tree Exam- ple (for the code in Listing 4.2)	140
5.5	Online Compression Scheme: Compression Tree Exam- ple (for the code in Listing 4.1)	141
5.6	Online Compression Scheme: Data and Code Mix . . .	144
5.7	Online Compression Scheme: Dictionary-Profile Interlink	145
5.8	Online Compression Scheme: Dictionary Sub-Classing	146
5.9	Online Compression Scheme: Dictionary Update Models	147
5.10	Online Compression Scheme: Single-Node Model . . .	150
5.11	Single-Node Compression Model, Continuous Stream: CF vs PS	151
5.12	Single-Node Compression Model, Continuous Stream: CF vs AS	152

5.13 Single-Node Compression Model, Continuous Stream: CF vs FCS	153
5.14 Single-Node Compression Model, Fragmented Stream: CF vs FN	154
5.15 Single-Node Compression Model, Fragmented Stream: CF vs FL	155
5.16 Single-Node Compression Model, Fragmented Stream: CF vs AS	156
5.17 Single-Node Compression Model, Fragmented Stream: CF vs FCS	157
5.18 Online Compression Scheme: Group Model	158
5.19 Online Compression Scheme: Example of Profile-Oriented Network	160
5.20 Cloud Compression Model: Continuous Stream	161
5.21 Cloud Compression Model: Fragmented Stream	162
5.22 Cloud Compression Model: Quasi-Real Setting	163
5.23 Single-Node Compression Model, Continuous Stream: Convergence vs PS	165
5.24 Single-Node Compression Model, Fragmented Stream: Convergence vs FN	166
5.25 Single-Node Compression Model, Fragmented Stream: Convergence vs FL	167
5.26 Single-Node Compression Model: Convergence vs AS	168
5.27 Single-Node Compression Model: Convergence vs FCS	169
5.28 Cloud Compression Model, Continuous Stream: Con- vergence vs PS	170
5.29 Cloud Compression Model, Fragmented Stream: Con- vergence vs FN	171
5.30 Cloud Compression Model, Fragmented Stream: Con- vergence vs FL	172
5.31 Cloud Compression Model: Convergence vs AS	173
5.32 Cloud Compression Model: Convergence vs FCS	174
6.1 Code Compression Process: “Hello World!” Application	189
6.2 Route Discovery	190
6.3 Code Compression Process: Route Discovery	193
6.4 Building Spanning Tree Algorithm	195
6.5 Code Compression Process: Spanning Tree Builder	196

List of Figures

6.6	Network Size Estimation using Capsules	198
6.7	Code Compression Process: Network Size Estimator . . .	199
6.8	Code Compression Process: Node ID Assignment	202
6.9	SBTSP: Working Principles	204
6.10	SBTSP: Implementation	205
6.11	SBTSP: 5 min sync interval, almost fully mesh topology, first 14 hours of the run shown	208
6.12	Code Compression Process: SBTSP	209
6.13	Data Collection Application: Code Build-Up	214
6.14	Code Compression Process: Data Collection Application	214
6.15	Disperser Protocol: 4-nodes topology with fixed links .	217
6.16	Disperser Protocol: Convergence	218
6.17	Code Compression Process: Disperser Protocol	220
6.18	Fire Tracker Possible Topologies	222
6.19	Code Compression Process: Fire Tracker	223
6.20	Compression Factor of Test Applications	225
7.1	Sink Role Re-Assignment	237
7.2	Distributed Labels	238
7.3	Hardware Acceleration for Code Compression	239
7.4	Deliverable Semantics	240
B.1	Block-Diagram for Fire-Tracking using Agilla	268

List of Tables

1.1	List of WSN Hardware Platforms	7
3.1	Classification of VM for WSN	42
3.2	ChameleonVM: Support Calls for Existing Protocols . .	77
5.1	Online Code Compression: Simulation Test Setting for Random Code Streams	149
5.2	Single-Node Compression Model: Simulation Overview	150
5.3	Convergence Speed: Simulation Overview	164
6.1	Characteristics of Test Applications: Program Size . . .	180
6.2	Radio Characteristics of TelosB and TinyNode WSN Plat- forms	182
6.3	Characteristics of Test Applications: Message Exchange Intensity (ChameleonVM)	185
6.4	Characteristics of Test Applications: Message Exchange Intensity (FragletVM)	186
6.5	Characteristics of Test Applications: Message Exchange Intensity (Agilla)	186
6.6	ChameleonVM: Source Code Notations	187
6.7	Example: Address Assignments During Route Discov- ery Process	192
6.8	SBTSP: Comparison of Static and Mobile Versions . . .	207
6.9	Code Size Estimation: Data Collection Application . . .	213
6.10	Data Collection Application: Dissemination Frequency Setting	215

List of Tables

6.11	Agilla: Agent Manipulation Instructions	222
6.12	Dictionary Size and Point of Convergence of Test Applications	225
6.13	Compression Factor of Test Applications	226
C.1	ChameleonVM's Basic Instruction Set: Stack Operations	269
C.2	ChameleonVM's Basic Instruction Set: Arithmetic Operations	269
C.3	ChameleonVM's Basic Instruction Set: Binary/Logic Operations	270
C.4	ChameleonVM's Basic Instruction Set: Flow Control . .	270
C.5	ChameleonVM's Basic Instruction Set: Capsule Manipulation	271
C.6	ChameleonVM's Basic Instruction Set: Communication	271
C.7	ChameleonVM's Basic Instruction Set: Memory Access	271
C.8	ChameleonVM's Basic Instruction Set: Miscellaneous .	272
C.9	ChameleonVM's Basic Instruction Set: Aliases	272
C.10	ChameleonVM's Basic Instruction Set: System Variables	272
D.1	FragletVM's Reduced Instruction Set: General Format .	273
D.2	FragletVM's Reduced Instruction Set: Data Stack Operations	273
D.3	FragletVM's Reduced Instruction Set: Null Operations	274
D.4	FragletVM's Reduced Instruction Set: Arithmetic Operations	274
D.5	FragletVM's Reduced Instruction Set: Conditional Operations	274
D.6	FragletVM's Reduced Instruction Set: Single Symbol Manipulations	275
D.7	FragletVM's Reduced Instruction Set: Split/Merge Operations	275
D.8	FragletVM's Reduced Instruction Set: Communication	275
D.9	FragletVM's Reduced Instruction Set: Inspection	276
D.10	FragletVM's Reduced Instruction Set: Extras	276

List of Listings

3.1	Simple Alarm Sensor in ChameleonVM	53
3.2	Simple Alarm Sensor in FragletVM	82
4.1	Example of “Case”-Branching	96
4.2	Example of “If”-Branching	97
4.3	Code Space-Polymorphism Example	113
4.4	Capsule’s Control Directives	116
5.1	General Cloud Model Setting Map File	160
6.1	WSN Version of “Hello World!”	184
6.2	Alternative WSN Version of “Hello World!”	188
6.3	Route Discovery	190
6.4	Spanning Tree	194
6.5	Network Size Estimator: Traveling Marker (node)	197
6.6	Network Size Estimator: Collector (sink)	198
6.7	Node ID Assignment: Traveling Capsule (node)	200
6.8	Node ID Assignment: “Multiply-With-Carry” Random Number Generator	201
6.9	Data Collection Application: sense and send measure- ments to the sink	211
6.10	Data Collection Application: collect and store measure- ments in a sink’s buffer	212
6.11	Disperser Protocol: 4-nodes topology with fixed links	216
6.12	Disperser Protocol: unknown topology	218
6.13	Neighbor Discovery	219

List of Listings

A.1	Pseudo-Code for SBTSP	255
A.2	TinyOS Code for SBTSP	256
A.3	SBTSP: Initialization Capsule	260
A.4	SBTSP: Resident Capsule	261
A.5	SBTSP: Beacon Capsule	262
A.6	SBTSP: Export Function	262
B.1	Assembler-Code for Fire-Tracking using Agilla	265

Appendix A

Pseudo- and Program-Code for SBTSP

```
1  CONSTANTS:
2
3  GUARD - guard interval [ms]
4  SLOT - number of slots (nodes)
5  SYNCWINDOW - sync interval [ms]
6  DATA - data exchange interval [ms]
7  SLEEP - offline interval [ms]
8
9  VARIABLES:
10
11 integer refTime
12 integer currentSkew
13 bool validSkewFlag
14 circBuffer skewTable
15 "skew measurement variables"
16 "received skew variables"
17 "constants (upper-case)"
18
19 EVENTS:
20
21 at wake-up time (= refTime):
22     reset skew measurement variables
23     reset received skew variables
24     start timer for sending beacon at "reftime + GUARD + myNodeId
25     * SLOT"
```

A. Pseudo- and Program-Code for SBTSP

```
26
27 at beacon b=<id,validS,s> reception:
28   accumulate measured skew = now() - ("reftime + GUARD + b.id
29     * SLOT")
30   if b.validS then
31     accumulate received b.s
32   fi
33
34 at beacon send time:
35   send beacon <myNodeId, validSkewFlag,currentSkew>
36   start timer for end-of-wake at "reftime + 2*GUARD + SYNCWINDOW
37     + DATA"
38
39 at end-of-wake time:
40   if any beacons received then
41     skewTable.add(average(measuredSkews))
42   else
43     if missedTooManyBeacons then
44       validSkewFlag = FALSE skewTable.reset()
45     fi
46   fi
47   if skewTable.isFull() then
48     currentSkew = skewTable.getAverage()
49     validSkewFlag = TRUE
50   fi
51   if any valid skews received AND validSkewFlag then
52     skewTable.adjust(diff/2)
53     diff = currentSkew - average(receivedSkews)
54     currentSkew = currentSkew + diff/2
55     reftime = reftime + diff/2
56   fi
57   reftime = reftime + "GUARD + SYNCWINDOW + GUARD + DATA
58     + SLEEP"
59   sleepUntil(reftime)
```

Listing A.1: Pseudo-Code for SBTSP

```
1 // time-sync exchange message
2 typedef nx_struct SkewMsg
3 {
4   nx_uint16_t nodeid;
5   nx_int16_t  skew;
6 } SkewMsg;
7
8 // constants
9 enum
10 {
```

A. Pseudo- and Program-Code for SBTSP

```
11  AM_SKEWMSG          = 42,
12
13  STATE_SLEEPING      = 0,
14  STATE_SYNCWINDOW    = 1,
15  STATE_STOPPING      = 2,
16  STATE_SENSING       = 3,
17  STATE_BOOTING       = 10,
18
19  FLAG_VALIDSKEW      = 1<<15,
20  FLAG_HUNTMODE       = 1<<14,
21  FLAG_MASK           = (FLAG_VALIDSKEW | FLAG_HUNTMODE),
22
23  SKEW_MAXENTRIES     = 8,
24
25  SY_NUMBER_OF_NODES  = 10,
26  SY_SLEEP_DURATION   = 3*1024,    // 3 s
27  SY_GUARD_DURATION   = 50,        // 50 ms
28  SY_SLOT_DURATION    = 50,
29  SY_SYNC_DURATION    = SY_NUMBER_OF_NODES * SY_SLOT_DURATION,
30  SY_SENSE_DURATION   = 500,
31  SY_CYCLE_DURATION   = 2 * SY_GUARD_DURATION + SY_SYNC_DURATION
32                      + SY_SENSE_DURATION + SY_SLEEP_DURATION,
33  SY_BOOT_DURATION    = SY_CYCLE_DURATION,
34 };
35
36 // variables
37 uint8_t    state = STATE_SLEEPING;
38 uint16_t   nodeid;
39 uint32_t   reftime;
40
41 message_t  sndpacket;
42
43 int16_t    skew;
44 int16_t    lastmeasured;
45 int16_t    adjSum;
46 int32_t    lastAdj;
47
48 int32_t    msrdSkewSum;
49 uint8_t    msrdSkewCnt;
50 int32_t    rcvdSkewSum;
51 uint8_t    rcvdSkewCnt;
52
53 int16_t    skewArray[SKEW_MAXENTRIES];
54 uint8_t    skewNext, skewCnt, skewGap;
55
56 // initial state
57 command void init()
```

A. Pseudo- and Program-Code for SBTSP

```
58 {
59     state = STATE_BOOTING;
60     nodeid = call AMPpkt.address();
61     call MilliTimer0.startOneShot(SY_BOOT_DURATION);
62 }
63
64 // calculate the skew adjustment using linear regression
65 // algorithm
66 int32_t getSkewAdjust()
67 {
68     uint8_t i;
69     int32_t tmp = 0;
70
71     if(skewCnt > 0)
72     {
73         skewGap++;
74         if(skewGap >= SKEW_MAXENTRIES/2)
75             skew = skewCnt = skewGap = lastAdj = 0;
76     }
77
78     if(msrdSkewCnt == 0) return lastAdj;
79
80     skewGap = 0;
81     skewArray[skewNext] = msrdSkewSum / msrdSkewCnt;
82     skewNext = (skewNext+1) % SKEW_MAXENTRIES;
83     if(skewCnt < SKEW_MAXENTRIES)
84     {
85         skewCnt++;
86         return 0;
87     }
88
89     for(i = 0, tmp = 0; i < SKEW_MAXENTRIES; i++)
90         tmp += skewArray[i];
91     skew = tmp / SKEW_MAXENTRIES;
92
93     if(rcvdSkewCnt == 0) return lastAdj;
94
95     tmp = (skew - rcvdSkewSum / rcvdSkewCnt) / 2;
96     for(i = 0; i < SKEW_MAXENTRIES; i++) skewArray[i] -= tmp;
97     skew -= tmp;
98
99     adjSum += tmp;
100    lastAdj = tmp;
101    return tmp;
102 }
103
104 // upon sending a message to the neighbors - do nothing
```

A. Pseudo- and Program-Code for SBTSP

```
105 event void SkewSend.sendDone(message_t *bufPtr,
106                               error_t error)
107 {
108 }
109
110 // upon reception of a new message - correct the times
111 event message_t *SkewRecv.receive(message_t *bufPtr,
112                                   void *payload,
113                                   uint8_t len)
114 {
115     SkewMsg *sm = (SkewMsg *) payload;
116
117     if(state == STATE_BOOTING)
118     {
119         sm->nodeid &= ~FLAG_MASK;
120         state = STATE_SENSING;
121         reftime = call MilliTimer0.getNow() - (SY_GUARD_DURATION +
122                                               sm->nodeid*SY_SLOT_DURATION);
123         call MilliTimer0.stop();
124         call MilliTimer0.startOneShotAt(reftime, 2*SY_GUARD_DURATION
125                                       + SY_SYNC_DURATION);
126     }
127     else if(state != STATE_SLEEPING)
128     {
129         lastmeasured = call MilliTimer0.getNow() - (reftime +
130                                                    SY_GUARD_DURATION + (sm->nodeid &
131                                                                    ~FLAG_MASK)*SY_SLOT_DURATION);
132         msrdSkewSum += lastmeasured;
133         msrdSkewCnt++;
134         if(sm->nodeid & FLAG_VALID_SKEW)
135         {
136             rcvdSkewSum += sm->skew;
137             rcvdSkewCnt++;
138         }
139     }
140
141     return bufPtr;
142 }
143
144 // when timer fires - make a decision what to do
145 event void MilliTimer0.fired()
146 {
147     SkewMsg *sm;
148
149     switch(state)
150     {
151     case STATE_SLEEPING:
```

A. Pseudo- and Program-Code for SBTSP

```
152     state = STATE_SYNCWINDOW;           // now is "reftime"
153     rcvdSkewCnt = msrdSkewCnt = msrdSkewSum = rcvdSkewSum = 0;
154     call MilliTimer0.startOneShotAt(reftime,
155                                     SY_GUARD_DURATION +
156                                     nodeid*SY_SLOT_DURATION);
157     break;
158 case STATE_BOOTING:
159     reftime = call MilliTimer0.getNow() - (SY_GUARD_DURATION +
160     nodeid*SY_SLOT_DURATION);
161 case STATE_SYNCWINDOW:
162     state = STATE_SENSING;
163     call MilliTimer0.startOneShotAt(reftime,
164                                     2*SY_GUARD_DURATION +
165                                     SY_SYNC_DURATION);
166     sm = (SkewMsg *) call SkewPack.getPayload(&sndpacket,
167     NULL);
168     sm->nodeid = nodeid;
169     if(skewCnt == SKEW_MAXENTRIES)
170     {
171         sm->skew = skew;
172         sm->nodeid |= FLAG_VALIDSKEW;
173     }
174     else sm->skew = 0;
175     call SkewSend.send(AM_BROADCAST_ADDR, &sndpacket,
176     sizeof(SkewMsg));
177     break;
178 case STATE_SENSING:
179     state = STATE_STOPPING;
180     call MilliTimer0.startOneShotAt(reftime,
181                                     2*SY_GUARD_DURATION +
182                                     SY_SYNC_DURATION +
183                                     SY_SENSE_DURATION);
184     // do some sensing and probably data exchange...
185     break;
186 case STATE_STOPPING:
187     state = STATE_SLEEPING;
188     reftime += SY_CYCLE_DURATION + getSkewAdjust();
189     call MilliTimer0.startOneShotAt(reftime, 0);
190 }
191 }
```

Listing A.2: TinyOS Code for SBTSP

```
1 { .code.init
2     SHMEM[SY_NUMBER_OF_NODES] = 20;
3     SHMEM[SY_GUARD_DURATION] = 50;           // 50 ms
4     SHMEM[SY_SLEEP_DURATION] = 5*60*1024; // ~5 min
```


A. Pseudo- and Program-Code for SBTSP

```
5  SHMEM[SY_SLOT_DURATION] = 50;           // 50 ms
6  SHMEM[SY_SYNC_DURATION] = SHMEM[SY_NUMBER_OF_NODES] *
7                               SHMEM[SY_SLOT_DURATION],
8  SHMEM[SY_CYCLE_DURATION] = 2 * SHMEM[SY_GUARD_DURATION] +
9                               SHMEM[SY_SYNC_DURATION] +
10                              SHMEM[SY_SENSE_DURATION] +
11                              SHMEM[SY_SLEEP_DURATION];
12  SHMEM[SY_BOOT_DURATION] = SHMEM[SY_CYCLE_DURATION];
13
14  SHMEM[SKEW_MAXENTRIES] = 8;
15
16  SHMEM[state] = STATE_BOOTING;
17  call Timer0.startOneShot(SHMEM[SY_BOOT_DURATION]);
18 }
```

Listing A.3: SBTSP: Initialization Capsule

```
1 { .code.timer0
2   switch(SHMEM[state]) {
3     case STATE_SLEEPING:
4       SHMEM[state] = STATE_SYNCWINDOW;
5       call Timer0.startOneShotAt(SHMEM[reftime] +
6         SHMEM[SY_GUARD_DURATION] +
7         getNodeid()*SHMEM[SY_SLOT_DURATION]);
8       break;
9     case STATE_BOOTING:
10      if the beacon capsule is installed {
11        SHMEM[state] = STATE_SYNCWINDOW;
12        SHMEM[reftime] = call Timer0.getNow() -
13          (SHMEM[SY_GUARD_DURATION] +
14            getNodeid()*SHMEM[SY_SLOT_DURATION]);
15        call "timer0" handler of the beacon capsule
16        break;
17      }
18      else {
19        call Timer0.startOneShot(SHMEM[SY_BOOT_DURATION]);
20        break;
21      }
22     case STATE_STOPPING:
23       SHMEM[state] = STATE_SLEEPING;
24       getSkewAdjust();
25       SHMEM[reftime] += SHMEM[SY_CYCLE_DURATION] +
26         BUFS[adjustment];
27       call Timer0.startOneShotAt(SHMEM[reftime]);
28   }}
```

Listing A.4: SBTSP: Resident Capsule

A. Pseudo- and Program-Code for SBTSP

```
1 { .code.init
2   BUFS[msrdSkewSum] = BUFS[msrdSkewCnt] =
3   BUFS[rcvdSkewCnt] = BUFS[rcvdSkewSum] = 0;
4   if(SHMEM[state] == STATE_BOOTING) {
5     BUFC[nodeid] &= ~SHMEM[FLAG_MASK];
6     SHMEM[state] = STATE_STOPPING;
7     SHMEM[reftime] = call Timer0.getNow() -
8     (SHMEM[SY_GUARD_DURATION] +
9     BUFC[nodeid]*SHMEM[SY_SLOT_DURATION]);
10    call Timer0.startOneShotAt(SHMEM[reftime] +
11    2*SHMEM[SY_GUARD_DURATION] + SHMEM[SY_SYNC_DURATION]);
12  }
13  else if(SHMEM[state] != STATE_SLEEPING) {
14    BUFS[msrdSkewSum] += call Timer0.getNow() -
15    (SHMEM[reftime] + SHMEM[SY_GUARD_DURATION] +
16    (BUFC[nodeid] &
17    ~SHMEM[FLAG_MASK])*SHMEM[SY_SLOT_DURATION]);
18    BUFS[msrdSkewCnt]++;
19    if(BUFC[nodeid] & SHMEM[FLAG_VALID_SKEW]) {
20      BUFS[rcvdSkewSum] += BUFC[skew];
21      BUFS[rcvdSkewCnt]++;
22    }
23  }
24
25  .code.timer0
26  if(SHMEM[state] == STATE_SYNCWINDOW) {
27    SHMEM[state] == STATE_STOPPING;
28    call Timer0.startOneShotAt(SHMEM[reftime] +
29    2*SHMEM[SY_GUARD_DURATION] + SHMEM[SY_SYNC_DURATION]);
30    if(BUFS[skewCnt] == SHMEM[SKEW_MAXENTRIES]) {
31      BUFC[nodeid] = call getNodeId() |
32      SHMEM[FLAG_VALID_SKEW];
33    }
34    else {
35      BUFC[nodeid] = call getNodeId();
36      BUFC[skew] = 0;
37    }
38    send "BEACON capsule" to ALL
39  }}
```

Listing A.5: SBTSP: Beacon Capsule

```
1 int32_t getSkewAdjust() {
2   // read out in-capsule fields
3   int32_t *msrdSkewSum = read(0x11, 0, 4);
4   uint8_t *msrdSkewCnt = read(0x11, 4, 1);
5   int32_t *rcvdSkewSum = read(0x11, 5, 4);
```

A. Pseudo- and Program-Code for SBTSP

```
6  uint8_t *rcvdSkewCnt = read(0x11, 9, 1);
7
8  static uint8_t skewNext, skewGap;
9  uint8_t *skewCnt = read(0x11, 10, 1);
10
11 static int32_t lastAdj;
12 int16_t *skew = read(0x11, 11, 2);
13
14 static int16_t skewArray[SHMEM[SKEW_MAXENTRIES]];
15
16 uint8_t i;
17 int32_t tmp;
18
19 if(*skewCnt > 0) {
20     skewGap++;
21     if(skewGap >= SHMEM[SKEW_MAXENTRIES]/2) *skew =
22         *skewCnt = skewGap = lastAdj = 0;
23 }
24
25 if(*msrdSkewCnt == 0) {
26     write_bufs(0x11, 13, &lastAdj, 4);
27     exit();
28 }
29
30 skewGap = 0;
31 skewArray[skewNext] = *msrdSkewSum / *msrdSkewCnt;
32 skewNext = (skewNext+1) % SHMEM[SKEW_MAXENTRIES];
33 if(*skewCnt < SHMEM[SKEW_MAXENTRIES]) {
34     *skewCnt++;
35     write_bufs(0x11, 13, NULL, 4);          # write 0s
36     exit();
37 }
38
39 for(i = 0, tmp = 0; i < SHMEM[SKEW_MAXENTRIES]; i++)
40     tmp += skewArray[i];
41 *skew = tmp / SHMEM[SKEW_MAXENTRIES];
42
43 if(*rcvdSkewCnt == 0) {
44     write_bufs(0x11, 13, &lastAdj, 4);
45     exit();
46 }
47
48 lastAdj = (*skew - *rcvdSkewSum / *rcvdSkewCnt) / 2;
49 for(i = 0; i < SHMEM[SKEW_MAXENTRIES]; i++)
50     skewArray[i] -= lastAdj;
51 *skew -= lastAdj;
52
```

A. Pseudo- and Program-Code for SBTSP

```
53 // write down in-capsule fields
54 write_bufs(0x11, 0, msrdSkewSum, 4);      # int32_t
55 write_bufs(0x11, 4, msrdSkewCnt, 1);     # uint8_t
56 write_bufs(0x11, 5, rcvdSkewSum, 4);    # int32_t
57 write_bufs(0x11, 9, rcvdSkewCnt, 1);    # uint8_t
58 write_bufs(0x11, 10, skewCnt, 1);        # uint8_t
59 write_bufs(0x11, 11, skew, 2);          # int16_t
60 write_bufs(0x12, 0, &lastAdj, 4);        # int32_t
61 }
```

Listing A.6: SBTSP: Export Function

Appendix B

Pseudo- and Program-Code for Fire-Tracking using Agilla

The algorithm and program code given below are borrowed from the demo application presented at http://mobilab.cse.wustl.edu/projects/agilla/Examples/single_agent_fire_detection/index.html. We use these algorithm and code “as is”.

```
1 BEGIN          pushn det          // check whether a detector
2                pushc 1          // is already here
3                rdp
4                rjumpc DIE
5 OUT_DETECTOR   pushn det          // detector not here, OUT a
6                pushc 1          // detector tuple
7                out              // atomic with respect to
8                rjump REGISTER_RXN // the rdp b/c of min # of
9 DIE            halt              // instructions (fix me)
10 RXN_FIRED     pushc 9
11              putled            // turn off green and yellow
12              pushn det        // LEDs
13              pushc 1
14              inp              // remove fire detector
15              halt            // tuple
16 REGISTER_RXN  pushn fir
17              pushc 1
```

B. Pseudo- and Program-Code for Fire-Tracking using Agilla

```
18          pushc RXN_FIRED
19          regrxn          // register fire reaction,
20 CHECK_NEIGHBORS pushc 28          // die when reaction fires
21          putled          // toggle yellow LED
22          pushn fir
23          pushc 1
24          rrdpg          // check if any neighbors
25          rjumpc FORM_BARRIER // are on fire
26 RANDOM_MOVE   pushn det
27          pushc 1
28          inp          // remove the detector tuple
29          pushc 9
30          putled          // turn off yellow and green
31          randnbr          // LEDs
32          wmove          // weak move to random
33          halt          // neighbor
34 FORM_BARRIER pushc 2
35          putled          // turn on the green LED
36          // (turn everything else
37          // off)
38          pushc 0          // for each neighbor whose
39          // dist <= 2 of fire,
40          // wclone to it
41          setvar 9          // heap[9] = neighbor
42 BARRIER_LOOP getvar 9          // counter, init=0
43          numnbrs
44          ceq
45          rjumpc BARRIER_DONE // done checking all
46          getvar 9          // neighbors
47          getnbr          // get the i'th neighbor
48          vicinity
49          rjumpc BARRIER_FIRE
50          pushcl BARRIER_NXT2 // not close to fire, skip
51          jumps
52 BARRIER_DONE rand
53          pushc 63
54          land
55          pushc 20
56          add
57          sleep          // sleep between 5/8-20/8s
58          pushc CHECK_NEIGHBORS
59          jumps
60 BARRIER_FIRE pushn det          // check if neighbor is on
61          pushc 1          // fire
62          getvar 9
63          getnbr
64          rrdp          // check if neighbor has a
```

B. Pseudo- and Program-Code for Fire-Tracking using Agilla

```
65                                     // detector
66     pushcl BARRIER_NXT // jump to BARRIER_NXT if
67     jumpc      // neighbor is on fire
68     pushn fir
69     pushc 1
70     getvar 9
71     getnbr
72     rrdp                                     // check if neighbor is on
73     rjumpc BARRIER_NXT // fire
74     getvar 9
75     getnbr
76     wclone                                     // clone self on neighbor
77     rjumpc BARRIER_NXT2
78 BARRIER_NXT clear // clear the op stack
79 BARRIER_NXT2 getvar 9
80     inc
81     setvar 9 // proceed to next neighbor
82     pushcl BARRIER_LOOP
83     jumps // proceed to next neighbor
```

Listing B.1: Assembler-Code for Fire-Tracking using Agilla

B. Pseudo- and Program-Code for Fire-Tracking using Agilla

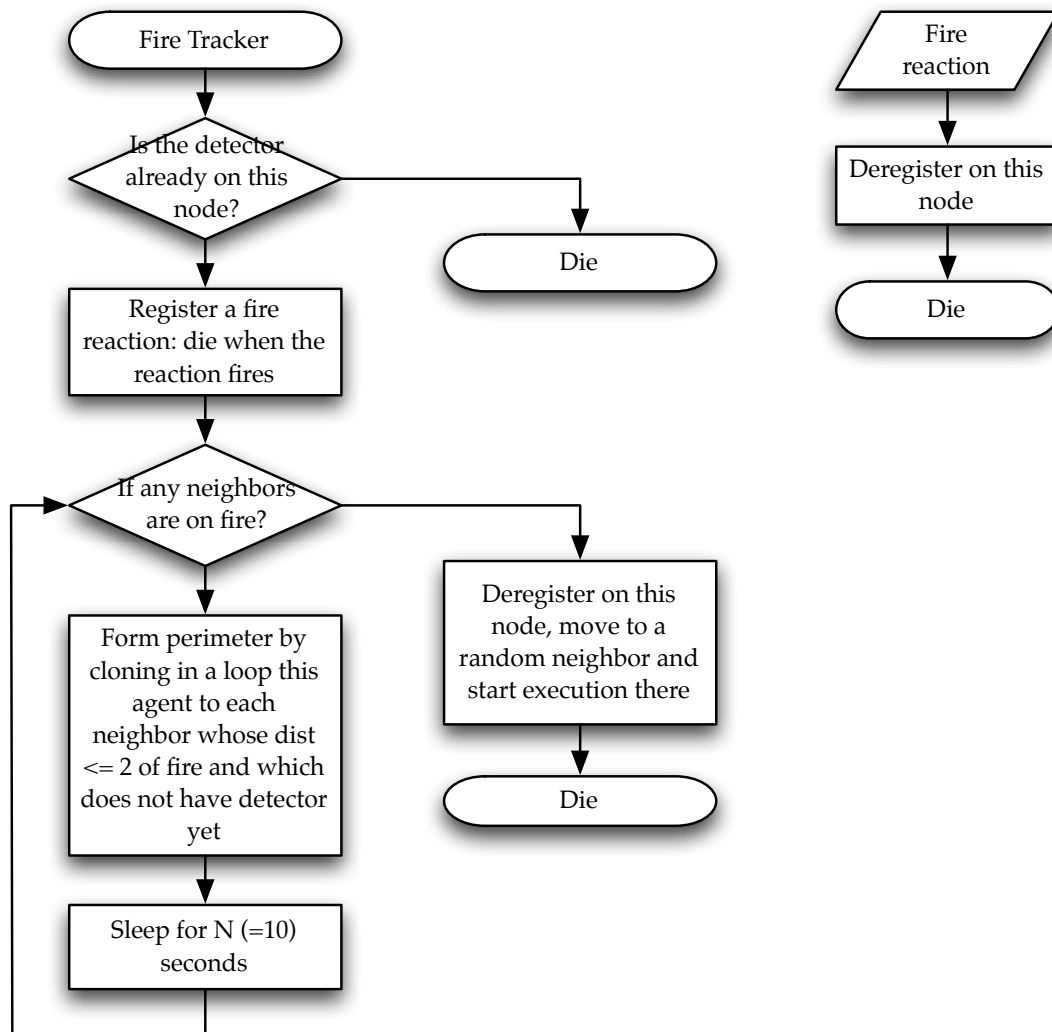


Figure B.1: Block-Diagram for Fire-Tracking using Agilla

Appendix C

ChameleonVM's Basic Instruction Set

The tables below contain the *ChameleonVM*'s basic instruction set. In the used syntax $*$ represents the rest of the stack.

Instruction	Semantics	Stack state / Remarks
push value on the stack	push x	$*$ \Rightarrow $*, x$
pop value from the stack	pop	$*, x \Rightarrow *$
swap the top two values on the stack	swap	$*, y, x \Rightarrow *, x, y$
duplicate the top value on the stack	dup	$*, x \Rightarrow *, x, x$
execute command from the data stack	exec	$*, c \Rightarrow *$; execute c as command

Table C.1: ChameleonVM's Basic Instruction Set: Stack Operations

Instruction	Semantics	Stack state / Remarks
sum two top values on the stack	add	$*, y, x \Rightarrow *, (y + x)$
subtract two top values on the stack	sub	$*, y, x \Rightarrow *, (y - x)$
multiply two top values on the stack	mult	$*, y, x \Rightarrow *, (y * x)$
divide (quotient) two top values on the stack	div	$*, y, x \Rightarrow *, (y / x)$
divide (get the remainder) two top values on the stack	rem	$*, y, x \Rightarrow *, (y \% x)$

Table C.2: ChameleonVM's Basic Instruction Set: Arithmetic Operations

C. ChameleonVM's Basic Instruction Set

Instruction	Semantics	Stack state / Remarks
binary "and" of the two top values on the stack	and	$*, y, x \Rightarrow *, (y \& x)$
binary "or" of the two top values on the stack	or	$*, y, x \Rightarrow *, (y x)$
binary "not" of the two top values on the stack	not	$*, x \Rightarrow *, (!x)$
binary "xor" of the two top values on the stack	xor	$*, y, x \Rightarrow *, (y \wedge x)$
binary right shift	rsh	$*, x, shift \Rightarrow *, (x >> shift)$
binary left shift	lsh	$*, x, shift \Rightarrow *, (x << shift)$

Table C.3: ChameleonVM's Basic Instruction Set: Binary/Logic Operations

Instruction	Semantics	Stack state / Remarks
jump if equal	jmqeq	$*, y, x, l \Rightarrow *; \text{if } x == y \text{ then } PC = l$
jump if less	jmqlt	$*, y, x, l \Rightarrow *; \text{if } x < y \text{ then } PC = l$
jump if less or equal	jmqle	$*, y, x, l \Rightarrow *; \text{if } x \leq y \text{ then } PC = l$
jump	jmp	$*, l \Rightarrow *; PC = l$

Table C.4: ChameleonVM's Basic Instruction Set: Flow Control

C. ChameleonVM's Basic Instruction Set

Instruction	Semantics	Stack state / Remarks
merge two capsules	<code>merge</code>	$*$, $\#ID$, $ABOVE/BELLOW/ALL$, $BEFORE/AFTER \Rightarrow *$; see Section 3.2.2
split the capsule	<code>split</code>	does not use the stack; see Section 3.2.2
clone the capsule	<code>clone</code>	same as <code>merge</code> ; see Section 3.2.2
replace the capsule	<code>replace</code>	$*$, $\#ID \Rightarrow *$; see Section 4.4
re-schedule the current capsule	<code>execute</code>	does not use the stack
free memory resources	<code>die</code>	does not use the stack
suspend execution	<code>exit</code>	does not use the stack
remove the capsule	<code>kill</code>	$*$, $\#ID \Rightarrow *$; see Section 4.4
erase a part of the capsule's code explicitly	<code>erase</code>	$*$, $TOP/BOTTOM/UP/DOWN \Rightarrow *$; see Section 4.1
reschedule execution of this capsule	<code>execap</code>	does not use the stack; see Section 3.2.2
synchronize the versions	<code>bind</code>	does not use the stack

Table C.5: ChameleonVM's Basic Instruction Set: Capsule Manipulation

Instruction	Semantics	Stack state / Remarks
send over radio	<code>send</code>	$*$, $ME/PACK/CAP/\#ID$, $addr/ALL/ANY/GROUP/UP/DOWN \Rightarrow *$; see Section 3.2.6
send over radio using destination	<code>sendd</code>	$*$, $ME/PACK/CAP/\#ID$, $addr/ALL/ANY/GROUP/UP/DOWN$, $addr \Rightarrow *$

Table C.6: ChameleonVM's Basic Instruction Set: Communication

Instruction	Semantics	Stack state / Remarks
move data between stack and memory	<code>mov</code>	$*$, $where$, $what \Rightarrow *$
read from the memory	<code>read</code>	$*$, $addr \Rightarrow *$, x ; see Section
write to the memory	<code>write</code>	$*$, $addr$, $x \Rightarrow *$; see Section
append to the buffer	<code>append</code>	$*$, $BUFC/BUFS/SHMEM \Rightarrow *$

Table C.7: ChameleonVM's Basic Instruction Set: Memory Access

C. ChameleonVM's Basic Instruction Set

Instruction	Semantics	Stack state / Remarks
set the system variable	set	$*$, NID , $x \Rightarrow *$
get the system variable	get	$*$ $\Rightarrow *$, $NID/GROUP/NUMN/CHIL/CHILS/PAR$
explicitly switch dictionaries	dict	$*$, $\#ID \Rightarrow *$; see Section
switch LED on/off	led	$*$, $how \Rightarrow *$
sets up the timer	timer or delay	$*$, $type$, $when \Rightarrow *$; timer assumes absolute value whereas delay is relative (offset from NOW)
sense the value	sense	$*$, $what \Rightarrow *$, x
print debug info	print	print debug into out through the local interface
reset the state of the VM	reset	all stack and heap pointers, error register, etc. are reset to their default values

Table C.8: ChameleonVM's Basic Instruction Set: Miscellaneous

Instruction	Semantics	Stack state / Remarks
sort the buffer in ascending/descending order	sort	$*$, $BUFC/BUFS/SHMEM$, $ASC/DESC \Rightarrow *$
generate a random number	rand	$*$ $\Rightarrow *$, x

Table C.9: ChameleonVM's Basic Instruction Set: Aliases

System Variable	Meaning
NOW	System clock (32-bit)
ME	This capsule
PACK	The packet being processed
CAP	The capsule being processed
$*.ID$	Capsule/packet's ID
$*.TO$	Next hop
$*.FROM$	Previous hop
$*.SRC$	Source address
$*.DST$	Destination address
BUFC	In-capsule data buffer
BUFS	On-node data buffer with exclusive access
SHMEM	On-node data buffer with shared access
ERREG	System error register
CONDREG	Condition register
NULL	Null packet

Table C.10: ChameleonVM's Basic Instruction Set: System Variables

Appendix D

FragletVM's Reduced Instruction Set

Below we present the limited subset of Fraglets instructions for embedded applications.

Instruction	Immediate version	Stack version
<i>instr</i>	[instr \$T \$1 ... \$n @TAIL] → [\$T op(\$1,...,\$n)]	[sinstr @TAIL \$n ... \$1] → [@TAIL op(\$1,...,\$n)]

Table D.1: FragletVM's Reduced Instruction Set: General Format

Instruction	Immediate version	Stack version
<i>push</i>	—	[spush \$1 @TAIL] → [@TAIL \$1]
<i>pop</i>	—	[spop \$T @TAIL \$1] → [\$T \$1 @TAIL]

Table D.2: FragletVM's Reduced Instruction Set: Data Stack Operations

D. FragletVM's Reduced Instruction Set

Instruction	Immediate version	Stack version
<code>nop</code>	<code>[nop @TAIL] → [@TAIL]</code>	—
<code>nul</code>	<code>[nul @TAIL] → []</code>	—

Table D.3: FragletVM's Reduced Instruction Set: Null Operations

Instruction	Immediate version	Stack version
<code>sum</code>	<code>[sum \$T \$1 \$2 @TAIL] → [\$T (\$1+\$2) @TAIL]</code>	<code>[ssum @TAIL \$2 \$1] → [@TAIL (\$2+\$1)]</code>
<code>diff</code>	<code>[diff \$T \$1 \$2 @TAIL] → [\$T (\$1-\$2) @TAIL]</code>	<code>[sdiff @TAIL \$2 \$1] → [@TAIL (\$2-\$1)]</code>
<code>mult</code>	<code>[mult \$T \$1 \$2 @TAIL] → [\$T (\$1*\$2) @TAIL]</code>	<code>[smult @TAIL \$2 \$1] → [@TAIL (\$2*\$1)]</code>
<code>div</code>	<code>[div \$T \$1 \$2 @TAIL] → [\$T (\$1/\$2) @TAIL]</code>	<code>[sdiv @TAIL \$2 \$1] → [@TAIL (\$2/\$1)]</code>
<code>mod</code>	<code>[mod \$T \$1 \$2 @TAIL] → [\$T (\$1%\$2) @TAIL]</code>	<code>[smod @TAIL \$2 \$1] → [@TAIL (\$2%\$1)]</code>
<code>pow</code>	<code>[pow \$T \$1 \$2 @TAIL] → [\$T (\$1^\$2) @TAIL]</code>	<code>[spow @TAIL \$2 \$1] → [@TAIL (\$2^\$1)]</code>

Table D.4: FragletVM's Reduced Instruction Set: Arithmetic Operations

Instruction	Immediate version	Stack version
<code>eq</code>	<code>[eq \$1 \$2 \$3 \$4 @TAIL] → (\$3==\$4?[\$1 \$3 \$4 @TAIL]:[\$2 \$3 \$4 @TAIL])</code>	<code>[seq @TAIL \$2 \$1] → [@TAIL (\$2==\$1?1:0)]</code>
<code>lt</code>	<code>[lt \$1 \$2 \$3 \$4 @TAIL] → (\$3<\$4?[\$1 \$3 \$04 @TAIL]:[\$2 \$3 \$4 @TAIL])</code>	<code>[slt @TAIL \$2 \$1] → [@TAIL (\$2<\$1?1:0)]</code>
<code>gt</code>	<code>[gt \$1 \$2 \$3 \$4 @TAIL] → (\$3>\$4?[\$1 \$3 \$4 @TAIL]:[\$2 \$3 \$4 @TAIL])</code>	<code>[sgt @TAIL \$2 \$1] → [@TAIL (\$2>\$1?1:0)]</code>
<code>if</code>	—	<code>[sif \$3 \$2 @TAIL \$1] → (\$1!=0?[\$3 @TAIL]:[\$2 @TAIL])</code>

Table D.5: FragletVM's Reduced Instruction Set: Conditional Operations

D. FragletVM's Reduced Instruction Set

Instruction	Immediate version	Stack version
<code>exch</code>	<code>[exch \$T \$1 \$2 @TAIL] → [\$T \$2 \$1 @TAIL]</code>	<code>[sexch @TAIL \$2 \$1] → [@TAIL \$1 \$2]</code>
<code>dup</code>	<code>[dup \$T \$X \$1 @TAIL] → [\$T \$1 \$1 @TAIL]</code>	<code>[sdup \$X @TAIL \$1] → [@TAIL \$1 \$1]</code>
<code>del</code>	<code>[del \$T \$X @TAIL] → [\$T @TAIL]</code>	<code>[sdel @TAIL \$X] → [@TAIL]</code>

Table D.6: FragletVM's Reduced Instruction Set: Single Symbol Manipulations

Instruction	Immediate version	Stack version
<code>fork</code>	<code>[fork \$T1 \$T2 @TAIL] → [\$T1 @TAIL] + [\$T2 @TAIL]</code>	<code>[sfork @TAIL \$2 \$1] → [\$1 @TAIL] + [\$2 @TAIL]</code>
<code>split</code>	<code>[split @T1 * @T2] → [@T1] + [@T2]</code>	—
<code>match</code>	<code>[match \$1 @T1] + [\$1 @T2] → [@T1 @T2]</code>	<code>[smatch @R \$1] + [\$1 @T2] → [@T1 @T2]</code>
<code>matchp</code>	<code>[matchp \$1 @T1] + [\$1 @T2] → [matchp \$1 @T1] + [@T1 @T2]</code>	—
<code>matchs</code>	<code>[matchs \$1 @T1] + [\$1 @T2] → [@T1 @T2] + [\$1 @T2]</code>	—
<code>matchps</code>	<code>[matchps \$1 @T1] + [\$1 @T2] → [matchps \$1 @T1] + [\$1 @T2] + [@T1 @T2]</code>	—

Table D.7: FragletVM's Reduced Instruction Set: Split/Merge Operations

Instruction	Immediate version	Stack version
<code>send</code>	<code>\$N1[send \$N2 @TAIL] → \$N1[send \$N2 @TAIL]</code>	<code>\$N1[ssend @TAIL \$N2] → \$N1[ssend @TAIL \$N2]</code>

Table D.8: FragletVM's Reduced Instruction Set: Communication

D. FragletVM's Reduced Instruction Set

Instruction	Immediate version	Stack version
<code>nodeid</code>	$\$N[\text{nodeid } \$T \$X \text{ @TAIL}] \rightarrow \$N[\$T \$N \text{ @TAIL}]$	$\$N[\text{snodeid } \$X \text{ @TAIL}] \rightarrow \$N[\text{@TAIL } \$N]$
<code>length</code>	$[\text{length } \$T \$X \text{ @TAIL}] \rightarrow [\$T \text{L}(\text{@TAIL}) \text{ @TAIL}]$	$[\text{slength } \$X \text{ @TAIL}] \rightarrow [\text{@TAIL } \text{L}(\text{@TAIL})]$
<code>rnd</code>	$[\text{rnd } \$T \$X \text{ @TAIL}] \rightarrow [\$T \text{ rnd}() \text{ @TAIL}]$	$[\text{srnd } \$X \text{ @TAIL}] \rightarrow [\text{@TAIL } \text{rnd}()]$
<code>sense</code>	$[\text{sense } \$T \$P \text{ @TAIL}] \rightarrow [\$T \$S_p \text{ @TAIL}]$	$[\text{ssense } \$P \text{ @TAIL}] \rightarrow [\text{@TAIL } \$S_p]$
<code>print</code>	print out debug info via serial interface	—
<code>erreg</code>	$[\text{erreg } \$T \$X \text{ @TAIL}] \rightarrow [\$T \$E \text{ @TAIL}]$	$[\text{serreg } \$X \text{ @TAIL}] \rightarrow [\text{@TAIL } \$E]$

Table D.9: FragletVM's Reduced Instruction Set: Inspection

Instruction	Immediate version	Stack version
<code>delay</code>	$[\text{delay } \$T \$D \text{ @TAIL}] \rightarrow [\$T \text{ @TAIL}]$	$[\text{sdelay } \text{@TAIL } \$D] \rightarrow [\$D \text{ @TAIL}]$
<code>newinstr</code>	$[\text{newinstr } \$T \$1 \$2 \text{ @TAIL}] \rightarrow [\$T \$1\$2 \text{ @TAIL}]$	$[\text{snewinstr } \text{@TAIL } \$2 \$1] \rightarrow [\$1 \text{ @TAIL } \$2]$
<code>blink</code>	blink a LED $[\text{blink } \$L \text{ @TAIL}] \rightarrow [\text{@TAIL}]$	—

Table D.10: FragletVM's Reduced Instruction Set: Extras

Bibliography

- [19596] RFC 1951. DEFLATE compressed data format specification version 1.3. 1996. 106, 128
- [32202] RFC 3229. Delta encoding in HTTP. 2002. 103
- [BA03] Kenneth Barr and Krste Asanovic. Energy aware lossless data compression. 2003. 36
- [BBF⁺11] Jan Beutel, Bernhard Buchli, Federico Ferrari, Matthias Keller, Lothar Thiele, and Marco Zimmerling. X-Sense: Sensing in extreme environments. 2011. 12, 237
- [BCD⁺05] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. MantisOS: An embedded multi-threaded operating system for wireless micro sensor platforms. 2005. 8
- [Beu06] Jan Beutel. Fast-prototyping using the BTnode platform. Proc. Design, Automation and Test in Europe (DATE 2006), 2006. 8
- [BGH⁺09] J. Beutel, S. Gruber, A. Hasler, R. Lim, A. Meier, C. Plessl, I. Talzi, L. Thiele, C. Tschudin, M. Woehrle, and M. Yuecel. PermaDAQ: A scientific instrument for precision sensing and data recovery under extreme conditions. 2009. 4, 11
- [BHR⁺06] Rahul Balani, Chih-Chieh Han, Ram Kumar Rengaswamy, Ilias Tsigkogiannis, and Mani Srivastava. Multi-level software re-configuration for sensor networks. 2006. 38
- [BHS04] A. Boulis, C.-C. Han, and M. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. 2004. 44
- [BJT⁺10] Ghazi Bouabene, Christophe Jelger, Christian Tschudin, Stefan Schmid, Ariane Keller, and Martin May. The autonomic network architecture (ANA). 2010. 25
- [Ble10] Guy E. Blelloch. Introduction to data compression. 2010. 103
- [BN08] David Boyle and Thomas Newe. Securing wireless sensor networks: Security architectures. 2008. 235
- [BS06] S. Brown and C.J. Sreenan. Updating software in wireless sensor networks: A survey. 2006. 37
- [BSE06] Bernd Burgstaller, Bernhard Scholz, and Anton Ertl. An embedded systems programming environment for C*. 2006. 128
- [BSTW86] J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive data compression scheme. 1986. 128
- [BvRW07] Nicolas Burri, Pascal von Rickenbach, and Roger Wattenhofer. Dozer: Ultra-low power data gathering in sensor networks. 2007. 194
- [BW94] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. 1994. 104, 128
- [CH87] Gordon Cormack and Nigel Horspool. Data compression using dynamic markov modelling. 1987. 105

Bibliography

- [CM99] K.D. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. 1999. 129
- [CPS07] Nuno Costa, Antonio Pereira, and Carlos Serodio. Virtual machines applied to WSN's: The state-of-the-art and classification. 2007. 42
- [CSCM00] L.R. Clausen, U.P. Schultz, C. Conzel, and G. Muller. Java bytecode compression for low-end embedded systems. 2000. 129
- [CW84] J.G. Cleary and I.H. Witten. Data compression using adaptive coding and partial string matching. 1984. 104
- [DBK⁺07] Matthias Dyer, Jan Beutel, Thomas Kalt, Patrice Oehen, Lothar Thiele, Kevin Martin, and Philipp Blum. Deployment support network: A toolkit for the development of WSNs. 2007. 38
- [DE02] S. Debray and W. Evans. Profile-guided code compression. 2002. 128
- [Deu96] P. Deutsch. GZIP file format specification version 4.3. 1996. 128
- [DFEV06a] Henri Dubois-Ferrière, Deborah Estrin, and Martin Vetterli. Packet combining in sensor networks. 2006. 118
- [DFEV06b] Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. 2006. 8, 40
- [DGV04] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki – a lightweight and flexible operating system for tiny networked sensors. *IEEE Emnets*, 2004. 8, 40
- [DOH07] A. Dunkels, F. Österlind, and Z. He. An adaptive communication architecture for wireless sensor networks. 2007. 50
- [DOM06] Suresh Upandra Donapudi, Christian Olesen Obel, and Jan Madsen. Extending lifetime of wireless sensor networks using forward error correction. 2006. 118
- [DOTH07] A. Dunkels, F. Österlind, N. Tsiftes, and Z. He. Software-based online energy estimation for sensor nodes. 2007. 184
- [Dun07] Adam Dunkels. *Programming Memory-Constrained Networked Embedded Systems*. PhD thesis, 2007. 38
- [EEF⁺97] J. Ernst, W.S. Evans, C.W. Fraser, S. Lucco, and T.A. Proebsting. Code compression. 1997. 129
- [EGE02] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. 2002. 203
- [ERR05] A. Eswaran, A. Rowe, and R. Rajkumar. Nano-RK: An energy-aware resource-centric RTOS for sensor networks. 2005. 8
- [Fal03] Kevin Fall. A delay-tolerant network architecture for challenged internets. 2003. 29
- [Fan49] R.M. Fano. The transmission of information. 1949. 109
- [FFMM06] Henri D. Ferrière, Laurent Fabre, Roger Meier, and Pierre Metrailler. TinyNode: A comprehensive platform for wireless sensor network applications. 2006. 182
- [FRL09a] C.-L. Fok, G.-C. Roman, and C. Lu. Agilla: A mobile agent middleware for self-adaptive wireless sensor networks. 2009. 24, 38, 41
- [FRL09b] C.-L. Fok, G.-C. Roman, and C. Lu. Enhanced coordination in sensor networks through flexible service provisioning. 2009. 44
- [FRWZ07] Elena Fasolo, Michele Rossi, Jörg Widmer, and Michele Zorzi. In-network aggregation techniques for wireless sensor networks: A survey. 2007. 216

-
- [Gag94] Philip Gage. A new algorithm for data compression. 1994. 107
- [Gil77] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. 1977. 87
- [GKS03] Saurabh Ganeriwal, Ram Kumar, and Mani B. Srivastava. Timing-sync protocol for sensor networks. 2003. 203
- [GLvB⁺03] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to network embedded systems. 2003. 6, 41
- [Gol66] S.W. Golomb. Run-length encodings. 1966. 110
- [Gra10] André Graf. PacketScript a Lua scripting engine for in-kernel packet processing. Master's thesis, 2010. 68
- [GVVL06] Sergio Gonzalez-Valenzuela, Son Vuong, and Victor C.M. Leung. A mobile code platform for distributed task control in wireless sensor networks. 2006. 44
- [HATW99] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R.V.D. Wiel. A code compression system based on pipelined interpreters. 1999. 128
- [HC04] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. 2004. 37, 40
- [HKSS05] C.-C. Han, R. Kumar, R. Shea, and M. Srivastava. Sensor network software update management: A survey. 2005. 37
- [HM06] Salem Hadim and Nader Mohamed. Middleware: Middleware challenges and approaches for wireless sensor networks. 2006. 127
- [HRS⁺05] Chih-Chieh Han, Ram Kumar Rengaswamy, Roy Shea, Eddie Kohler, and Mani Srivastava. SOS: A dynamic operating system for sensor networks. Proceedings of the Third International Conference on Mobile Systems, Applications, and Services (Mobisys), 2005. 8, 40
- [Huf52] D. Huffman. A method for the construction of minimum-redundancy codes. 1952. 108
- [IGE00] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. 2000. 26, 31
- [IT10] Pierre Imai and Christian Tschudin. Practical online network stack evolution. 2010. 16, 232
- [JE03] Jaemin Jeong and Cheng Tien Ee. Forward error correction in sensor networks. 2003. 118
- [KBX⁺04] Porlin Kang, Cristian Borcea, Gang Xu, Akhilesh Saxena, Ulrich Kremer, and Liviu Iftode. Smart messages: A distributed computing platform for networks of embedded systems. 2004. 44
- [KFC04] Sukun Kim, Rodrigo Fonseca, and David Culler. Reliable transfer on wireless sensor networks. 2004. 118
- [KH01] S. Kwong and Y.F. Ho. A statistical lempel-ziv compression algorithm for personal digital assistant. 2001. 107
- [KHM⁺08] Ariane Keller, Theus Hossmann, Martin May, Ghazi Bouabene, Christophe Jelger, and Christian Tschudin. A system architecture for evolving protocol stacks. 2008. 231
- [KMMV02] D. Korn, J. MacDonald, J. Mogul, and K. Vo. The VCDIFF generic differencing and compression data format. 2002. 128

Bibliography

- [Kol65] A.N. Kolmogorov. Three approaches to the quantitative definition of information. 1965. 126
- [KYB09] M. Keller, M. Yucel, and J. Beutel. High resolution imaging for environmental monitoring applications. 2009. 11
- [LBCM97] C. Lefurgy, P. Bird, I. C. Chen, and T. Mudge. Improving code density using compression techniques. 1997. 129
- [LC02] Philip Levis and David Culler. M \acute{a} te: A tiny virtual machine for sensor networks. 2002. 38, 112, 221
- [LF03] M. Latendresse and M. Feeley. Generation of fast interpreters for huffman compressed bytecode. 2003. 128
- [LGC04] Philip Levis, David Gay, and David Culler. Bridging the gap: Programming sensor networks with application specific virtual machines. 2004. 41, 72, 133
- [LGC05] Philip Levis, David Gay, and David Culler. Active sensor networks. 2005. 41, 72, 112, 133
- [LM98] C. Lefurgy and T. Mudge. Code compression for DSP. 1998. 129
- [LM03] T. Liu and M. Martonosi. Impala: A middleware system for managing autonomic, parallel sensor systems. 2003. 44
- [LMP⁺04] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. TinyOS: An operating system for sensor networks. 2004. 6
- [Lou06] Max Loubser. Delay tolerant networking for sensor networks. 2006. 29
- [LPCS04] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. 2004. 38, 41
- [LSW09] Christoph Lenzen, Philipp Sommer, and Roger Wattenhofer. Optimal clock synchronization in networks. 2009. 203
- [LW99] H. Lekatsas and W. Wolf. SAMC: A code compression algorithm for embedded processors. 1999. 129
- [M \ddot{u} 10] René Müller. *Data Stream Processing on Embedded Devices*. PhD thesis, 2010. 238
- [Mac03] David J. C. MacKay. Information theory, inference, and learning algorithms. 2003. 110
- [Mah05] M. Mahoney. Adaptive weighing of context models for lossless data compression. 2005. 105
- [MAK07] René Müller, Gustavo Alonso, and Donald Kossmann. SwissQM: Next generation data processing in sensor networks. 2007. 238
- [Mey10] Thomas Meyer. *On Chemical and Self-Healing Networking Protocols*. PhD thesis, 2010. 27, 78, 85, 89, 121, 215, 216
- [MKSL04] Miklòs Marì, Branislav Kusy, Gyula Simon, and Àkos Lèdeczi. The flooding time synchronization protocol. 2004. 203
- [Mos09] Clemens Moser. *Power Management in Energy Harvesting Embedded Systems*. PhD thesis, 2009. 4
- [MP09] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. 2009. 41
- [MSTY08] Thomas Meyer, Daniel Schreckling, Christian Tschudin, and Lidia Yamamoto. Robustness to code and data deletion in autocatalytic quines. 2008. 118

-
- [MT09] Thomas Meyer and Christian Tschudin. Chemical networking protocols. 2009. 28, 82
- [MT11] Thomas Meyer and Christian Tschudin. Flow management in packet networks through interacting queues and law of mass action scheduling. 2011. 87, 216
- [MuuR07] Bojan Mihajlović, Željko Žilic, and Katarzyna Radecka. Compression and encryption of self-test programs for wireless sensor network nodes. 2007. 128
- [MYT08a] Thomas Meyer, Lidia Yamamoto, and Christian Tschudin. An artificial chemistry for networking. 2008. 28
- [MYT08b] Thomas Meyer, Lidia Yamamoto, and Christian Tschudin. A self-healing multipath routing protocol. 2008. 102
- [NGK99] Erik L. Nygren, Stephen J. Garland, and M. Frans Kaashoek. PAN: A high-performance active network node supporting multiple mobile code systems. 1999. 23
- [NK08] Chauhan Durg Singh Nitin and Sehgal Vivek Kumar. Two $\mathcal{O}(n^2)$ time faulttolerant parallel algorithm for Inter NoC communication in NiP. 2008. 231
- [NM79] G. Nigel and N. Martin. Range encoder range encoding: An algorithm for removing redundancy from a digitized message. 1979. 109
- [Phi05] Lane A. Phillips. *Aqueduct: Robust and Efficient Code Propagation in Heterogeneous Wireless Sensor Networks*. PhD thesis, 2005. 111
- [PSC05] Joseph Polastre, Robert Szewczyk, and David E. Culler. Telos: Enabling ultra-low power wireless research. 2005. 182
- [Pug99] W. Pugh. Compressing Java class files. 1999. 129
- [QR07] Saad Bin Qaisar and Hayder Radha. OPERA: An optimal progressive error recovery algorithm for wireless sensor networks. 2007. 118
- [RÖ1] Kay Römer. Time synchronization in ad hoc networks. 2001. 203
- [Ris76] Jorma Rissanen. Generalized kraft inequality and arithmetic coding. 1976. 109
- [RL79] J.J. Rissanen and Jr. G.G. Langdon. Arithmetic coding. 1979. 109, 128
- [RMYT08] Juan J. Ramos-Muñoz, Lidia Yamamoto, and Christian Tschudin. Serial experiments online. 2008. 232
- [RN10] Prakash Ranganathan and Kendall Nygard. Time synchronization in wireless sensor networks: A survey. 2010. 203
- [RP71] R. F. Rice and R. Plaunt. Adaptive variable-length coding for efficient compression of spacecraft television data. 1971. 110
- [RS04] Montserrat Ros and Peter Sutton. A hamming distance based VLIW/EPIC code compression technique. 2004. 129
- [Rya80] B. Ya. Ryabko. Data compression by means of a "book stack". 1980. 104
- [RZS⁺08] Michele Rossi, Giovanni Zanca, Luca Stabellini, Riccardo Crepaldi, Albert F. Harris III, and Michele Zorzi. SYNAPSE: A network reprogramming protocol for wireless sensor networks using fountain codes. 2008. 119
- [Sal97] David Salomon. Data compression the complete reference. 1997. 107
- [SBB05] Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. Link-time binary rewriting techniques for program compaction. 2005. 127

Bibliography

- [Sha48] C.E. Shannon. A mathematical theory of communication. 1948. 109
- [SHE03] Thanos Stathopoulos, John Heidemann, and Deborah Estrin. A remote code update mechanism for wireless sensor networks. 2003. 38
- [SM06] C. Sadler and M. Martonosi. Data compression algorithms for energy-constrained devices in delay tolerant networks. 2006. 128
- [SSM⁺01] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli. Addressing the System-on-a-Chip interconnect woes through communication-based design. 2001. 231
- [SW73] D. Slepian and J. Wolf. Noiseless coding of correlated information sources. 1973. 111
- [SW09] Philipp Sommer and Roger Wattenhofer. Gradient clock synchronization in wireless sensor networks. 2009. 203
- [Tal08] Igor Talzi. Time synchronization with post-hoc calibration in small-scaled intermittently connected wireless sensor networks. 2008. 30, 203, 204, 207
- [TDV08a] Nicolas Tsiftes, Adam Dunkels, and Thiemo Voigt. Efficient sensor network reprogramming through compression of executable modules. 2008. 102, 111
- [TDV08b] Nicolas Tsiftes, Adam Dunkels, and Thiemo Voigt. Efficient sensor network reprogramming through compression of executable modules. 2008. 128
- [THGT07] I. Talzi, A. Hasler, S. Gruber, and C. Tschudin. PermaSense: Investigating permafrost with a WSN in the Swiss Alps. 2007. 10
- [Tsc03] Christian Tschudin. Fraglets – a metabolic execution model for communication protocols. 2003. 28, 78
- [TST08] I. Talzi, S. Schönborn, and C. Tschudin. Providing data integrity in intermittently connected wireless sensor networks. 2008. 29
- [Tur04] Jim Turley. Code compression under the microscope. 2004. 130
- [vGR03] Jana van Greunen and Jan Rabaey. Lightweight time synchronization for sensor networks. 2003. 203
- [Vit87] J. S. Vitter. Design and analysis of dynamic huffman codes. 1987. 108
- [WC92] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. 1992. 129
- [WDHN03] Jian Wu, Stefan Dulman, Paul Havinga, and Tim Nieberg. Multipath routing with erasure coding for wireless sensor networks. 2003. 118
- [WGT98] David Wetherall, John Guttag, and David Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. 1998. 23
- [WLLP01] Brett Warneke, Matt Last, Brian Liebowitz, and Kristofer S.J. Pister. Smart dust: Communicating with a cubic-millimeter. 2001. 4
- [WNC87] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. 1987. 109
- [XK08] Cao Xiaomin and M. Kuijper. A distributed source coding framework for multiple sources. 2008. 111
- [XLC04] Z. Xiong, A.D. Liveris, and S. Cheng. Distributed source coding for sensor networks. 2004. 111
- [ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. 1977. 106

Curriculum Vitae

Period	Activity
June 1, 1982	Born in Leningrad (Saint-Petersburg), USSR (Russian Federation)
1988–1997	Secondary school, St-Petersburg, Russia
1997–1999	Lyceum #239 specialized in mathematics and physics, ^a St-Petersburg, Russia
1999–2003	Study of “Computer and Information Science” at <i>St-Petersburg State Polytechnic University</i> , Faculty of Engineering Cybernetics, Department of Automatics and Computer Engineering ^b
02/2002– 05/2002	Embedded software tester at JSC “ <i>Astronautics Kearfott Electroavtomatika (AKE)</i> ,” ^c St-Petersburg, Russia
2003	Dipl. Bachelor of Science (BSc)
2003–2005	Study of “Computers, Systems and Networks” at <i>St-Petersburg State Polytechnic University</i> , Faculty of Engineering Cybernetics, Department of Automatics and Computer Engineering
07/2003– 06/2004	Firmware and FPGA design/verification engineer at JSC “ <i>Advanced System Design</i> ,” ^d St-Petersburg, Russia
2005	Dipl. El.-Ing. (equals MSc)
04/2005– 02/2006	Trainer and technical support engineer at “ <i>SWD Software Ltd</i> ,” ^e St-Petersburg, Russia
2006–2011	PhD student in the group of Prof. Dr. Christian Tschudin, <i>University of Basel</i> , Department of Mathematics and Computer Science, Computer Networks Group, ^f Switzerland

^a<http://www.239.ru/>

^b<http://kspt.ftk.spbstu.ru/>

^c<http://www.ake.spb.ru/>

^d<http://www.a-sys-d.com/> <http://www.actel.ru/>

^e<http://www.swd.ru/>

^f<http://cn.cs.unibas.ch/>