Energy Efficient Software Update Mechanism for Networked Component-based IoT Devices

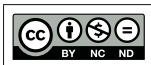
by

Ngoc Hai BUI

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT FOR A MASTER'S DEGREE
WITH THESIS IN TELECOMMUNICATIONS NETWORK
M.A.Sc.

MONTREAL, 12 DECEMBER 2019

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC



BOARD OF EXAMINERS

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

Mr. Kim Khoa Nguyen, Thesis Supervisor
Department of Electrical Engineering, École de technologie supérieure

Mr. Mohamed Cheriet, Co-supervisor
Department of Automated Manufacturing Engineering, École de technologie supérieure

Mr. Aris Leivadeas, President of the Board of Examiners
Department of Software Engineering and IT, École de technologie supérieure

Mr. Pascal Giard, Member of the jury
Department of Electrical Engineering, École de technologie supérieure

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

ON 22 NOVEMBER 2019

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to my supervisor, professor Kim Khoa Nguyen for giving me the opportunity to work with him and for his patient guidance and continuous support throughout the years. I also would like to give a special thanks to my co-supervisor, professor Mohamed Cheriet for his effective supervision, his assistance, and his encouragement during my master's project.

I extend a big thank you to my parents, my brother and my girlfriend for their moral support and encouragement for the completion of this thesis.

I would like to give my honest appreciation to my colleagues at Synchromedia for their great support. And finally, I also want to thank all my friends who gave me moral support during my master's degree. Thank you!

Mécanisme de mise à jour logicielle écoénergétique pour les appareils IoT à base de composants en réseau

Ngoc Hai BUI

RÉSUMÉ

En raison de problèmes de sécurité et des exigences supplémentaires des utilisateurs, les logiciels des appareils IoT doivent être changés fréquemment pour améliorer les fonctionnalités existantes ou pour corriger les bogues. La mise à jour de logiciels est devenue une tâche intégrale des systèmes IoT afin de maintenir des opérations efficaces. Récemment, l'architecture du système logiciel commun des périphériques IoT avancés est basée sur des composants qui peuvent être mis à jour au moment de l'exécution. Dans de tels réseaux IoT, les appareils peuvent télécharger des composants mis à jour à partir de nœuds voisins, permettant ainsi un déploiement rapide des mises à jour. Dans ce contexte, la distribution des composants logiciels doit prendre en compte deux problèmes principaux: i) comment fournir des mises à jour de tous les périphériques de manière écoénergétique, et ii) comment déployer rapidement des mises à jour pour éviter de longues périodes d'inactivité du réseau.

Dans ce mémoire, nous proposons un mécanisme qui planifie les mises à jour de tous les appareils d'un réseau de appareil IoT dans le but de minimiser la consommation d'énergie, en tenant compte de la contrainte du délai pour la mise à jour de l'ensemble du réseau. Contrairement aux études précédentes sur les mises à jour logicielles basées sur des composants IoT, qui traitent souvent de la manière dont un composant est remplacé dans le système d'exploitation, nous nous concentrons sur la distribution des composants dans le réseau et étudions le processus de mise à jour intervenu dans la mémoire flash d'un périphérique, dans lequel l'ordre de réécriture des composants dans la mémoire est déterminant pour la consommation d'énergie.

Nous introduisons un nouveau modèle énergétique du processus de mise à jour à l'intérieur d'un appareil en nous concentrant sur l'opération de réécriture du memoire flash, qui consomme une quantité d'énergie importante dans le processus de mise à jour. Ensuite, nous formulons un modèle d'optimisation mathématique pour le problème de la planification des mises à jour écoénergétiques.

En raison de la grande complexité du problème, nous proposons ensuite un algorithme appelé ESUS, qui se rapproche du ordonnancement optimal pour la mise à jour de tous les périphériques du réseau. Pour évaluer notre algorithme de planification, nous comparons les résultats d'ESUS aux solutions optimales données par un solveur mathématique. Les résultats de la simulation montrent l'efficacité de notre méthode qui est proche de la solution de planification optimale avec un temps d'exécution beaucoup plus court que celui du solveur.

Mots-clés: efficacité énergétique, mise à jour de logiciel, dispositif IoT, logiciel IoT à base de composants

Energy Efficient Software Update Mechanism for Networked Component-based IoT Devices

Ngoc Hai BUI

ABSTRACT

Due to security issues and incremental user requirements, software in IoT devices needs to be changed frequently to improve existing functionalities or to fix bugs. Software updates have become an integral task of IoT systems to maintain effective operations. Recently, the common software architecture in advanced IoT devices is component-based, in which components can be updated at run time. In such IoT networks, devices can download updated components from neighbor nodes, enabling quick deployment of updates. In this context, there are two main issues in the distribution of software components that needed to pay attention: i) how to deliver updates to all devices in an energy-efficient way, and ii) how to quickly deploy updates to avoid long network downtime.

In this thesis, we propose a mechanism that schedules updates on all devices in an IoT edge network with the goal to minimize the energy consumption, taking into account the deadline constraint for updating the entire network. Unlike previous studies on IoT component-based software update, which often focus on how a single component is replaced in the operating system, we focus on the distribution of components in the network and investigate the update process happened in the flash memory of a device, in which the order of re-written components into the memory is decisive for energy consumption.

We introduce a novel energy model of the update process inside a device, focusing on the flash re-writing operation which consumes a significant amount of energy in the update process. Then, we formulate a mathematical optimization model for the problem of energy efficient update scheduling.

Because of the high complexity of the problem, we then propose an algorithm called ESUS to approximate the optimal schedule for updating all devices in the network. To evaluate our scheduling algorithm, we compare the results of ESUS to the optimal solutions given by a mathematical solver. Simulation results show the efficiency of our method, which is close to optimal scheduling solution with much lower execution time compared to the solver.

Keywords: energy efficiency, software update, IoT device, component-based IoT software

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

LIST OF ALGORITHMS

Page

LIST OF ABREVIATIONS

CPU	Central Processing Unit
ECD	Efficient Code Dissemination
EEPROM	Electrically Erasable Programmable Read-only Memory
ELF	Executable and Linkable Format
ESUS	Energy-efficient Sofware Update Scheduling
IDE	Integrated Development Environment
INLP	Integer Non Linear Programming
I/O	input/output
IoT	Internet of Things
NP	Non-deterministic Polynomial-time
P2P	Peer to Peer
OPL	Optimization Programming Language
OS	Operating System
RAM	Random Access Memory
RFID	Radio-Frequency Identification
RISC	Reduced Instruction Set Computer
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver-Transmitter
UDP	User Datagram Protocol

USB	Universal Serial Bus
WSN	Wireless Sensor Networks

LISTE OF SYMBOLS AND UNITS OF MEASUREMENTS

KB/s	Kilobytes per second
KB	Kilobytes
s	Second
mJ	Milijoule

CHAPTER 1

INTRODUCTION

1.1 Context and motivation

The Internet of things (IoT) is a convergence of Internet with advanced wireless communications, sensor and smart objects, where everyday objects can interact with each another to access all kinds of real-world information and provide various intelligent services and applications (Al-Fuqaha, Guizani, Mohammadi, Aledhari & Ayyash, 2015). In order to adapt incremental user requirements of IoT applications, software in IoT devices need to be changed frequently to improve existing functionalities or to fix revealed bugs, and the need to update the running software of IoT devices arises. For this reason, software updates must become an integral part of IoT systems to maintain effective operations.

With the IoT boom, the number of smart devices is growing fast, and advanced functionalities are developed increasingly, which brings many challenges to deployment and management. Although IoT networks are often deployed in large scales, IoT devices are usually highly resource constrained, with small memory storage, low processing power and limited energy capacity, and they have to strictly follow low-cost requirements. The large scale of device networks, together with the limited communication bandwidth, the low capacity of every node, and the deployment in high access cost environments, makes the task of updating these systems extremely challenging. Therefore, in spite of the fact that software updates are common in all kinds of systems, updating IoT device networks comes with additional difficulties.

Various approaches have been developed to distribute and install new software in deployed IoT/wireless sensor systems. Each of them is suitable for one kind of IoT device software, including full system image replacement (Hui & Culler, 2004), image differencing approaches (Panta, Bagchi & Midkiff, 2011), virtual machines (Koshy & Pandey, 2005a), and runtime-loadable code modules as in Contiki and SOS (Hahm, Baccelli, Petersen & Tsiftes, 2016). Recently, the common execution environment in advanced IoT devices is component-based, such

as Contiki and SOS, in which software is partitioned into small blocks, so-called components, which can be added or updated at run-time. In such an environment, only parts of the entire software need to be changed during the update process, allowing to reduce the amount of data needed to be transferred. Therefore, there have been many studies investigating component-based software systems of IoT devices to improve the update process (Ruckebusch, De Poorter, Fortuna & Moerman, 2016),(Munawar, Alizai, Landsiedel & Wehrle, 2010), (Amjad, Sharif, Afzal & Kim, 2016).

Prior research on component-based software for IoT devices often focused on the ways a component is replaced and did not consider thoroughly how updates are distributed, especially when multiple components are required to be deployed at the same time. In this thesis, we investigate the case of distributing updates to the entire network, focusing on the typical type of IoT edge networks consisting of a number of devices with the same component-based software connected to a gateway, and a set of components needs to be updated to all devices. In this kind of networks, a peer-to-peer manner can help reduce the time to deliver the update to the entire network, enabling quick update deployment, since a device can download updated components from multiple neighbor nodes at the same time through cheap communication technologies (e.g., Bluetooth or WiFi) without having to rely on a more expensive communication with the gateway.

An example of such a network is presented in Fig. 1.1, where each device is running Contiki and has to send temperature information to a gateway every ten minutes. Device software consists of four components a, b, c and d with specific roles as follows: component a reads temperature data from sensors, component b processes the data, component c sends the data and component d is the main task control. When the programmer wants to change the data processing algorithm and the transport protocol (e.g., from UDP to TCP), he will generate only two new components b' and c' to replace b and c, respectively, instead of the entire new software as in legacy devices. These components are typically compiled as Executable and Linkable Format (ELF) files (Ruckebusch *et al.*, 2016). They can be downloaded and stored in a buffer such as an EEPROM, then the Contiki core will link them to existing components and

load them into the flash memory in run-time. As shown in Fig. 1.1, component b' is transferred from the gateway to devices 1 and 3, then it is sent to device 2 from 1. In contrast, component c' is transmitted from the gateway to 3, from 3 to 2 and from 2 to 1, consecutively.

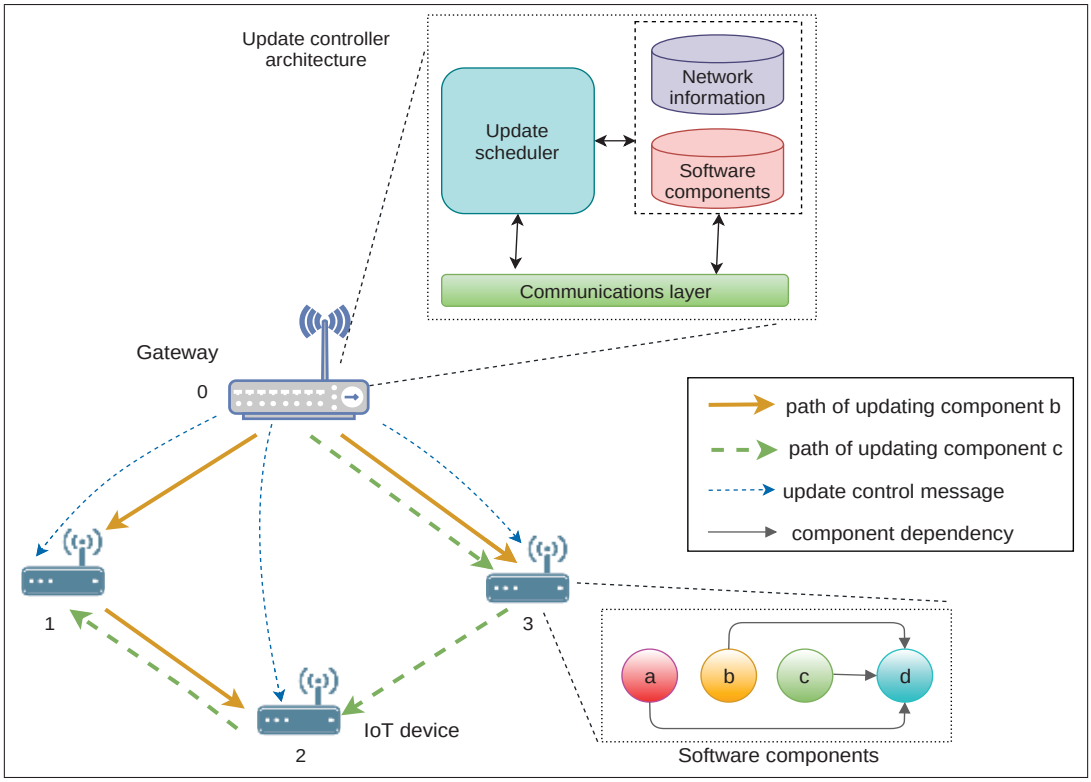


Figure 1.1 Example of an IoT network with a gateway which is responsible to update IoT devices

In this context, there are two issues that require our attention. The first one is the problem of energy consumption, which is the classical issue of IoT/sensor devices as well as of any embedded systems in general. Because most of edge devices are powered by battery power supply with very limited energy capacity and difficult to be replaced in the deployed environment, minimizing energy consumption is always a crucial task to prolong the operational lifetime of the network. Hence, deploying software updates in an energy efficient way is a significant requirement for every IoT operator. Although there have been many dissemination protocols proposed to deliver updates with low energy consumption, such as Varuna (Panta, Vintila & Bagchi, 2010) and Triva (Saginbekov & Jhumka, 2014), these protocols only con-

sider the communication cost between nodes and do not take into account the update process happening inside device memories. In a device running component-based software system, a key operation that consumes a significant amount of energy in the update process is flash re-writing (Panta *et al.*, 2011), in which the order of re-writing components into the memory is decisive for energy consumption, as will be explained in the following section. So, determining an optimal component update order is substantial for reducing energy consumption in the device software update operation.

The second issue we should consider is the time required to update the entire network. Due to the Quality of service(QoS) requirements of different IoT applications, the downtime of update operation should be minimized. Therefore, quickly deliver updates to the whole network is another important requirement that can bring benefits to both users and service providers.

The aforementioned issues raise a problem of update scheduling in which we can find a proper update schedule that minimizes the energy consumed during the update operation while satisfying a deadline constraint for updating the entire network. A schedule is a plan which specifies two decisions: First, each device should download a component from which node? And second, when each component can be downloaded?

1.2 Problem statement

We continue to describe our scheduling problem by illustrating more details with the network example in Fig. 1.1. Inside an IoT device, software components are written in a sequence in flash memory as shown in Fig. 1.2, from low addresses to high addresses. Each component may reside in several memory pages. When a component is updated (assume its size changes), its memory pages need to be re-written completely, and all the components placed next to it in the memory have to be shifted to other addresses (Dong et al., 2015). Therefore, all these components also need to be re-written. In that context, different orders could result in different numbers of re-written blocks, which leads to different amounts of energy consumption. For instance, two components *b* and *c* in Fig. 1.2, account for 3 and 2 pages in the flash, respec-

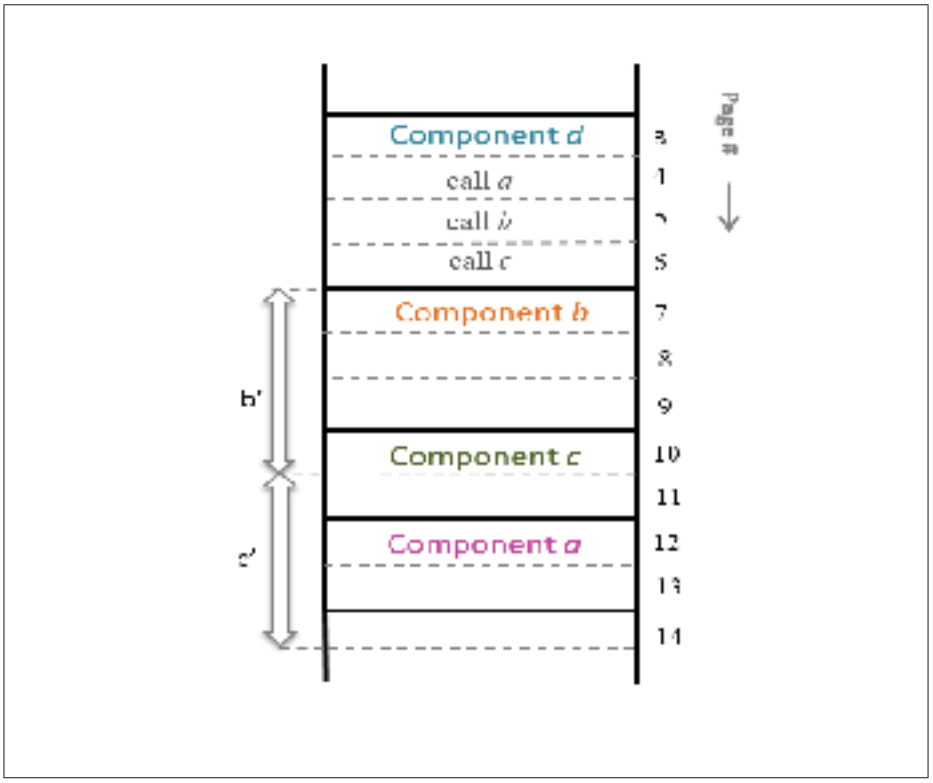


Figure 1.2 Software components in flash memory of an IoT device
b and *c* are updated by *b'* and *c'*

tively. We update both b and c by the new components b' and c' that have both 4-page size. In the first case, if the update order is (c,b), we have to re-write 4 pages of c' and pages of component a that is located after c, then 4 pages of b', 4 pages of c' and finally a has to be re-written again. Hence, the total number of re-written pages are 12 plus twice the size of a. This update order example is illustrated in Fig. 1.3a. Now, let consider a better way to do in this situation, as shown in Fig. 1.3b. If we update b first, we have to re-write 4 pages of b', only 2 pages of current size of c and pages of a, then 4 pages of c' and pages of a again, so the total pages are 10 plus twice the size of a, that is smaller than the first case.

Furthermore, in component-based software systems, some components may call the others during their execution (Ruckebusch *et al.*, 2016). This dependency leads to an update order constraint, in which a component can only be updated when the components it depends on had all been updated. Otherwise, an inconsistency error would be experienced. This constraint has

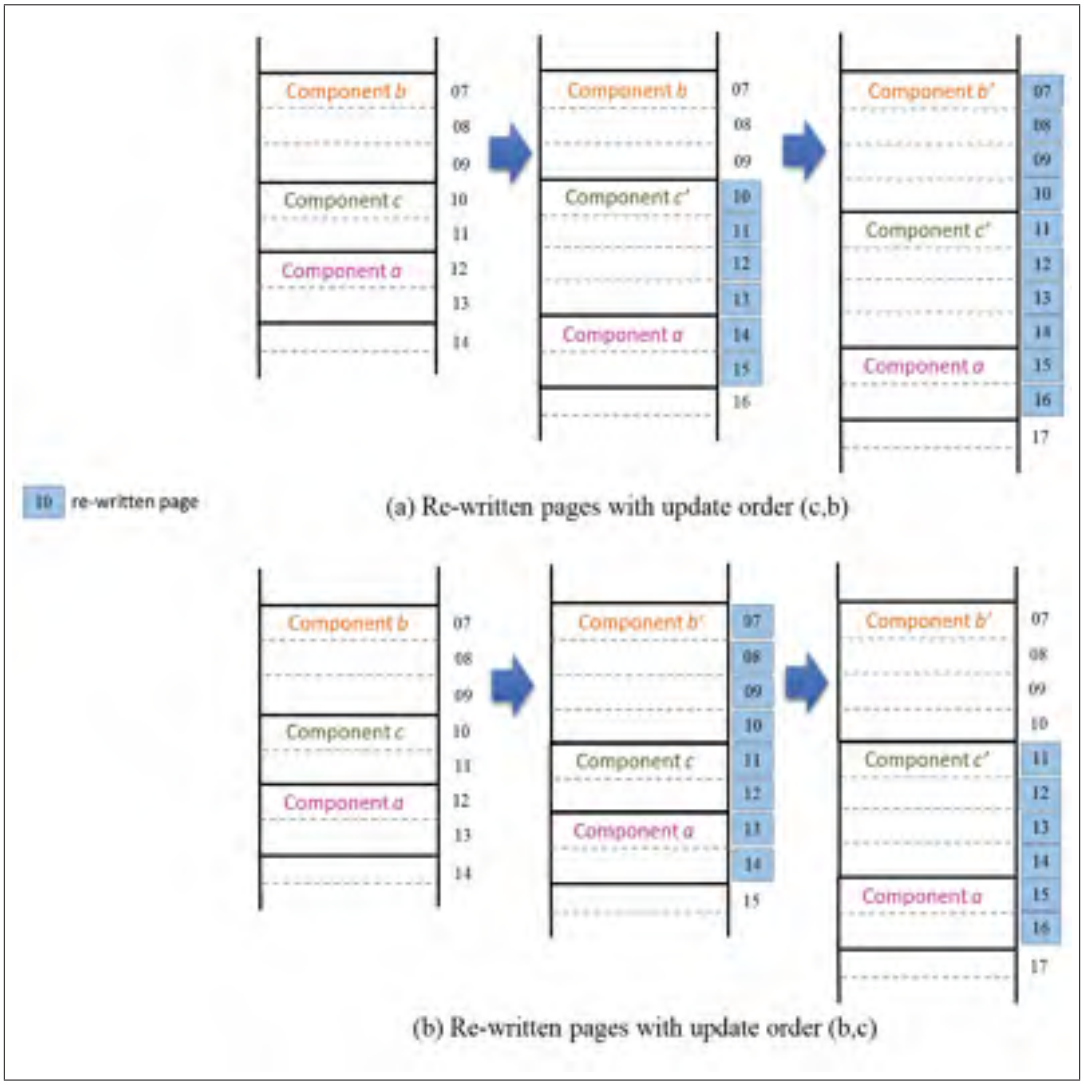


Figure 1.3 Different number of re-written pages with different component update orders

to be taken into account when we make an update schedule for the network. Some work (Dong, Chen, Bu & Huang, 2013a) also mentioned the update order constraint, however, the constraint and its impact on energy have not been well-considered in previous studies.

Fig. 1.4 shows two different examples of update schedules, in which the components *b* and *c* are distributed in the network with diverse paths and different download time. That leads to various component update orders in devices, which results in a difference in the total amounts of energy consumed.

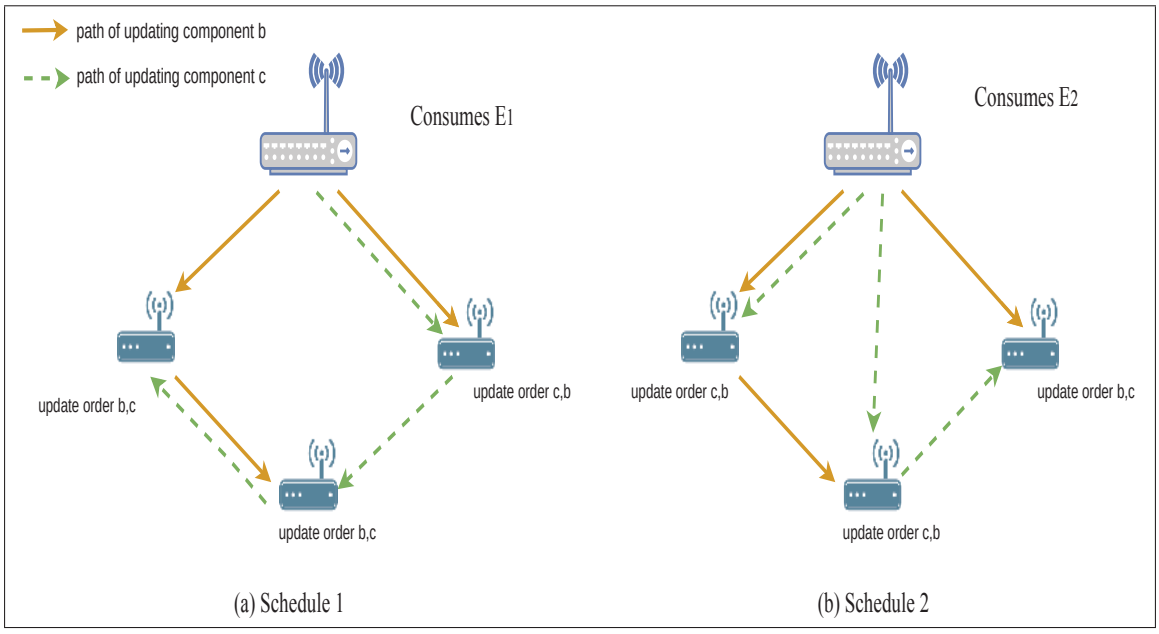


Figure 1.4 Different update schedules result in different amounts energy consumption

In our work, we propose a mechanism that determines update schedules for all devices with the goal to minimize the total energy consumption of the update process, taking into account the component update orders, the component dependencies and the deadline constraint for updating all the devices in the network. Our mechanism optimizes the update schedule while satisfying the five constraints:

- The dependency order of components.
- The topology constraint of the network.
- The constraint that a device can only send a component after having it.
- The constraint that a device can download at most one component from one source at a time.
- The deadline constraint of updating the entire network.

In terms of complexity, for a single device, finding the best component update order is equivalent to finding the best topological order of the component dependency graph (as in Fig. 4.1b),

which is known to be NP-hard in general. Hence, with a large number of devices in the network, and with the aforementioned additional constraints, our scheduling problem would be more complex. For this reason, finding the optimal update schedule is not a trivial task.

1.3 Research questions

In order to make an optimal schedule for delivering updates to all devices in an IoT network, so that the total energy consumption is minimized while satisfying the update order constraint of software components as well as the deadline for updating the entire network, we have to deal with the following research questions:

- **RQ1**. How can we model the energy consumption of the update process in a component-based IoT device, according to the computing resource consumption?

  With the goal of minimizing the energy consumption of updating all devices in the network, the first step is calculating the energy consumed in each device, and the parameters that affect the energy need to be defined. We should consider the effect of the component update order and the component dependency on the energy consumption of the update process. The proposed model has to take into account the specific characteristics of IoT devices such as hardware architecture, program memory and program loader.

- **RQ2**. How can we optimize the total energy consumption of the update process in the entire network?

  The purpose is to formulate an optimization model with the objective function is the total energy to update the entire network. The model needs to take into account all the constraints of the system, including network topology, bandwidth and component dependency.

- **RQ3**. How to compute the energy efficient update schedule in polynomial time?

  In IoT context, fast deployment is required to quickly adapt incremental user requirements or to promptly solve revealed issues. So the computation time of an exact optimal method is not practical because a solver often needs hours or even days to find the optimal solution.

Hence, It is necessary to design a fast algorithm to give near-optimal solutions in order to achieve faster computation time.

1.4 Objectives

The main objective of this thesis is to propose a mechanism that schedules updates on all devices to minimize the energy consumption, taking into account the component dependencies and the deadline constraint for updating the entire network. This objective can be divided into three specific objectives based on previous research questions.

- **SO1**. Build an energy model of the update process in a component-based IoT device.

  A model is necessary to understand the sources of energy consumption and how parameters affect the energy consumed during the update process. From the energy model, we can find a way to reduce the energy of each device as well as of the entire network.

- **SO2**. Build an optimization model for the energy efficient update scheduling problem.

  In order to bring energy efficiency to the update process, we need to formulate our optimization problem with a clear objective function and constraints.

- **SO3**. Propose an efficient algorithm to solve our optimization problem.

  In our system model, the update controller is deployed in a gateway, it can take very long time (days) to find an optimal schedule with a solver, that leads to a waste of time and computing resources. Moreover, in some cases, application providers want to deploy new applications as soon as possible to adapt to new user requirements. Finding solution in short time can help quickly adapt to new requirements of IoT applications.

1.5 Plan

The thesis is divided into five main chapters, followed by a conclusion and perspectives for further research, the chapters are organized as follows:

- The first chapter is the introduction. We first present the context and motivation of this study, then, the problem statement, the research questions and objectives are presented.
- The second chapter summarizes the technical background related to our research. In this chapter, we discuss IoT networks, device hardware architecture, and run-time technologies of IoT devices. In terms of run-time technologies, we focus more on devices with component-based software architecture, which is investigated in this research.
- The third chapter discusses the related work. We present a review of the prior research in IoT device update, which is divided into three sub topics, update dissemination, data minimization and run-time environments.
- The fourth chapter presents to the methodology. According to our objectives, the first part of this chapter is dedicated to the system modeling, in which we describe the software system model of each device and the IoT network system under consideration. In the second part, we introduce the proposed energy model and the optimization model. Then, we present the ESUS algorithm to solve the optimization model in the last part of this chapter.
- The fifth chapter presents the experimental setup and simulation scenarios, and then discusses the simulation results.

CHAPTER 2

BACKGROUND

In this chapter, we present the technical background related to our study. We first introduce the concepts of IoT edge networks and the existing update mechanisms of IoT devices. Then, we present the typical hardware architecture of devices in IoT edge networks to give some basic ideas about the update process and how energy is consumed during the process. Finally, we introduce the existing device's software run-time technologies, which are crucial for the ways to update IoT devices.

2.1 IoT edge networks

2.1.1 Overview of IoT edge networks

An IoT edge network typically involves IoT devices such as sensors, actuators and gateways that connect and communicate with each other and with the IoT platform (Leukert, Kubach, Eckert, Tsutsumi, Crawford & Vayssiere, 2016). The network scale ranges from small deployment with a few sensor devices directly connected to an IoT platform running on the cloud to a large factory with all production tools with extensive communication components. An edge network can include a separate local network or networks where devices connect by various protocols and via several routers to an edge gateway. The network can use different topologies for internal connections between devices as well as connections from devices to the gateway, such topologies can be either star, in which all communications within the local network go through the edge gateway, or mesh, in which some IoT devices have routing ability.

In our work, we consider the type of edge network including a local network connected to a gateway. The gateway works as a broker between the local network of devices and a wide area network (WAN) which connect to the platform. It is responsible for managing devices of the local network and isolating the edge devices from the WAN. IoT devices may directly communicate to the edge gateway or connect through routers, and there may be routes of

connections between devices that do not pass through the gateway. Fig. 2.1, which is taken from (Leukert *et al.*, 2016), presents a typical IoT edge network.

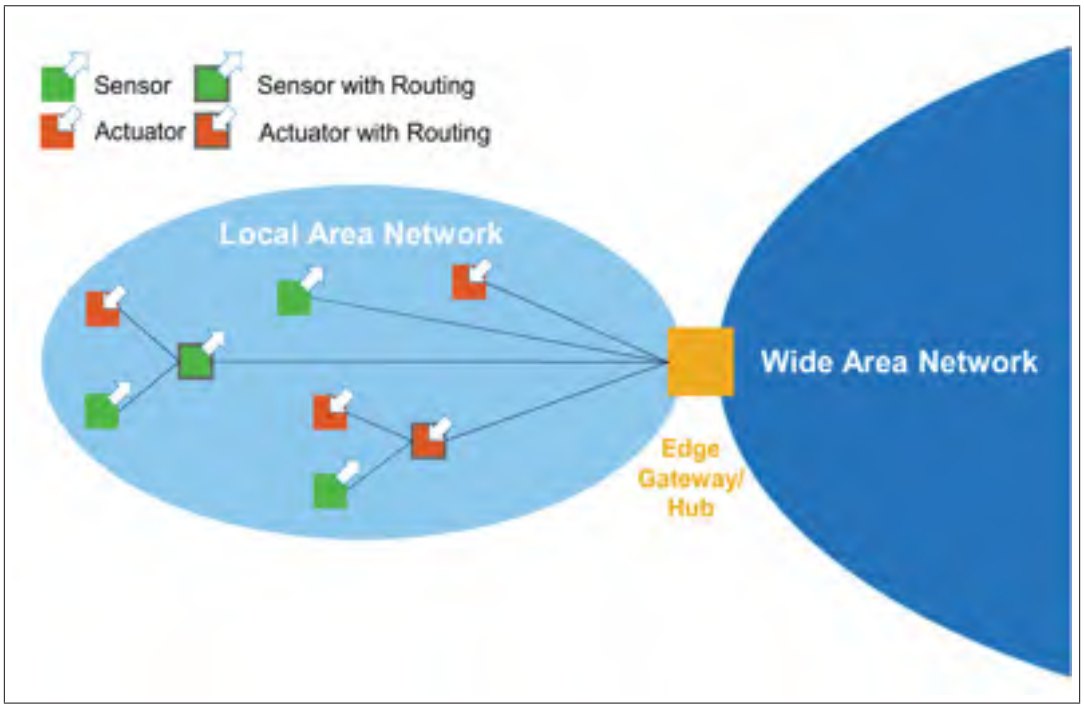


Figure 2.1 Example of an IoT edge network

Devices in an edge network consist of distributed sensors and actuators. An IoT device is a small embedded system that are typically equipped with one or more sensors to carries out some individual tasks (such as measuring temperature or humidity, or turning on/off a light or a machine), performs at low power so that it can run on battery or employ energy harvesting. IoT devices are often small and low cost, with very restricted computing/storage capacity and limited energy resources, so they can be deployed in a large scale.

2.1.2 IoT device hardware

In this subsection, we introduce the common IoT device's hardware architecture, in order to give understanding about the software update process. We focus on the processing unit, which is the main part that executes the software update operation.

2.1.2.1 Hardware architecture of IoT sensor devices

Most sensor devices are typically very small, low energy consumption, autonomous and adaptive to the environment. A device often has four basic components which are sensing unit, processing unit, transceiver unit and energy unit (McGrath & Scanaill, 2013). In this research, we investigate the update process which happens in the processing unit. The processing unit stores application code and data, it has the duty to manage, process data and control other components in the sensor device. This unit is typically a microcontroller that includes a processor and some small storage memories, microcontrollers are often used for sensor devices because they are low cost, easy to program, and consumes little energy.

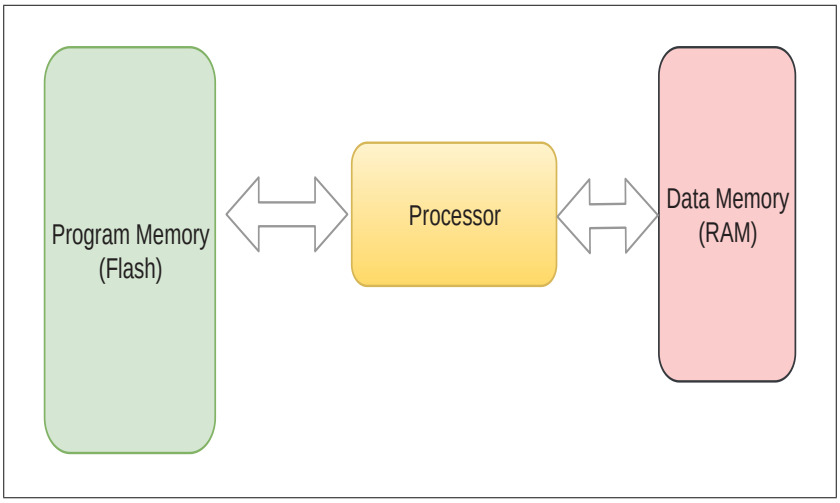


Figure 2.2 Overview of the Harvard architecture

Most microcontrollers of IoT sensor devices employ Harvard architecture (Healy, Newe & Lewis, 2008) with memories usually include two independent memory areas that are Flash and RAM. Flash memory is used for storing installed application code and it is the main location of the update process, while RAM is used for storing input/output data and temporary computations. The overview of Harvard architecture in sensor devices is presented in Fig. 2.2. In our problem, we assume that all IoT devices in the network have the same Harvard architecture of microcontrollers, which is seen in many common IoT device platforms such as Arduino family (Arduino.cc, 2019) and Mica sensor family (Karray, Jmal, Garcia-Ortiz, Abid & Obeid, 2018).

2.1.2.2 Flash memory of sensor devices

A flash memory is a non-volatile memory component that can store saved data even when no power supply is available. The flash memory of an IoT device works like the hard disk to store installed applications, this memory supports at least 10,000 erase/write cycles. Unlike RAM, where the smallest memory block of read and write operations can be a byte or a word, the basic unit of such operations in flash memory is a page, which is the smallest granularity of data addressable by the flash. A flash page is continuous memory space, typical size ranging from 512$Bytes$ to 4 $Kilobytes(KB)$, this is the smallest flash unit that can be erased and written (Park, Kim, Urgaonkar, Lee & Seo, 2011).

Writing into flash is performed in a page-by-page manner, because partial writing or erasing a page is not accepted, that means any modification of any byte in a page will result in the entire page needs to be re-written. Data for every single page have to be stored in a temporary buffer before writing to the flash. In addition, if a page already contains some data, it must be erased before being re-written again because flash memory does not provide the overwriting feature. Flash re-writing operation is performed by the bootloader as the following (Koshy & Pandey, 2005b). First, the bootloader stores the new content of the page in a buffer such as RAM or EEPROM. Second, the page is erased and finally, the content is written to the blank page. Flash re-writing consumes more energy than other operations, this feature was mentioned in many previous studies such as (Panta *et al.*, 2011), (Koshy & Pandey, 2005b) and (Heo, Gu, Eo, Kim & Jeon, 2010), it needs to be minimized as much as possible to reduce the energy consumption during program loading/updating process.

2.2 Existing update mechanisms for IoT edge networks

In an IoT edge network, there are two traditional approaches to update devices: centralized and Peer-to-Peer (P2P) (Brown & Sreenan, 2013).

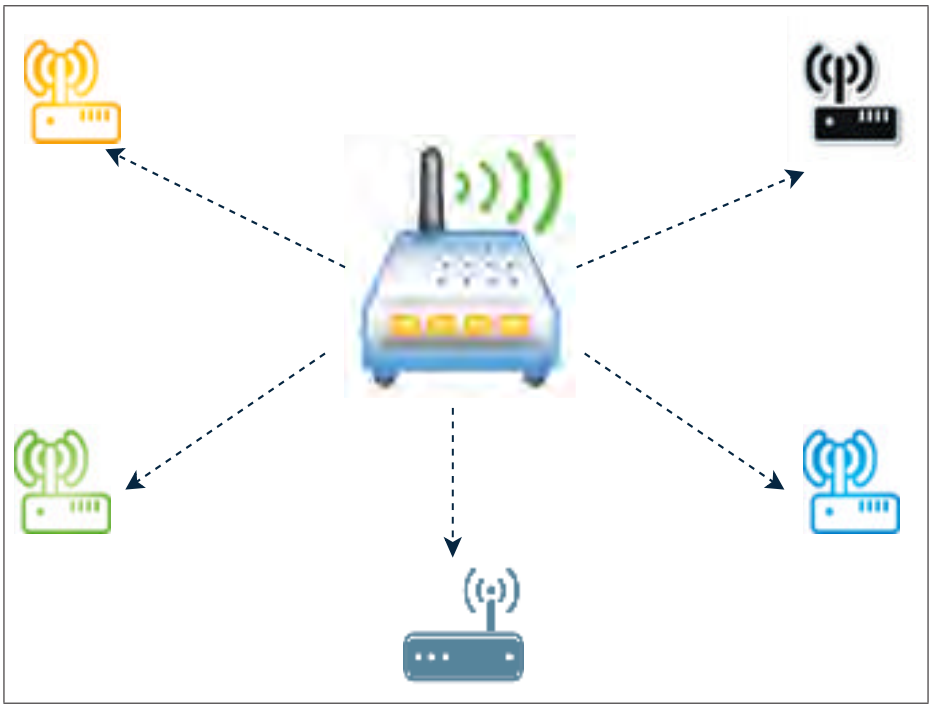


Figure 2.3 Centralized update mechanism

2.2.1 Centralized mechanism

The centralized model, also called client-server, is a traditional update delivery mechanism of IoT networks as well as of the Internet, in which every node in the network connects to a server to get updates, as shown in Fig. 2.3. The centralized server is responsible for managing and delivering the new software. It typically employs some complex algorithms and protocols to perform the update process. It stores the updates, collects information about states of devices (e.g. software version), pushes data to devices when new updates are available or schedule the time to update for devices (Kolomvatsos, 2018). A server can be a local gateway or a cloud node running in a datacenter on the Internet. To fulfill the task, the server should be aware of all connected nodes in the network and monitor statuses of nodes. With the centralized mechanism, the number of nodes is a challenge, because it is difficult for the server to manage and serve all the nodes in a huge network. On the device side, an update agent is used in each device to execute the update, the agent has duties to receive updates from server, apply the update and reset the device (if needed).

2.2.2 Peer-to-Peer mechanism

Another typical update mechanism of IoT/sensor networks is peer-to-peer(P2P), that is illustrated in Fig. 2.4, in which devices directly distribute updates to each other. In this mechanism, nodes are more autonomous compared to the centralized model. Originally, software is also stored on a server (also called base station, or sink node), when a new update is available, the server broadcast an advertisement message to the network, some nodes receive the message and check their current version, then contact the server to get new code. After that, nodes also broadcast their own messages to others, to advertise about the software versions they have. By receiving messages, a node compares the version of it with others, then decides to get the new version or not. In case a node receives multiple advertisement messages of the same version, it will select the source to download the update based on a certain strategy. The way nodes contact each other to distribute updates is called dissemination protocol, it is a broad topic in IoT networks as well as in Wireless Sensor Networks (Taherkordi, Loiret, Rouvoy & Eliassen, 2013).

In the literature, update dissemination protocols have been designed with the goal to optimize the update process of the device network. These protocols need to satisfy two key requirements. The first is energy efficiency. Because wireless communication is high energy consuming, and most devices are powered by very limited energy sources which are difficult to be replaced in the operational environment, energy efficiency is the most important requirement of IoT/sensor networks, and minimizing energy consumption is an integral task to extend the operational lifetime of the system. The second requirement of any protocol is dissemination latency. While the new software is being distributed in the network, devices may have erroneous and useless states which cause downtime of the entire system, as cooperating nodes may have different software versions running. In this case, the update time is useless, which must be minimized. Therefore, an advantageous dissemination protocol should also distribute updates quickly in order to reduce the downtime.

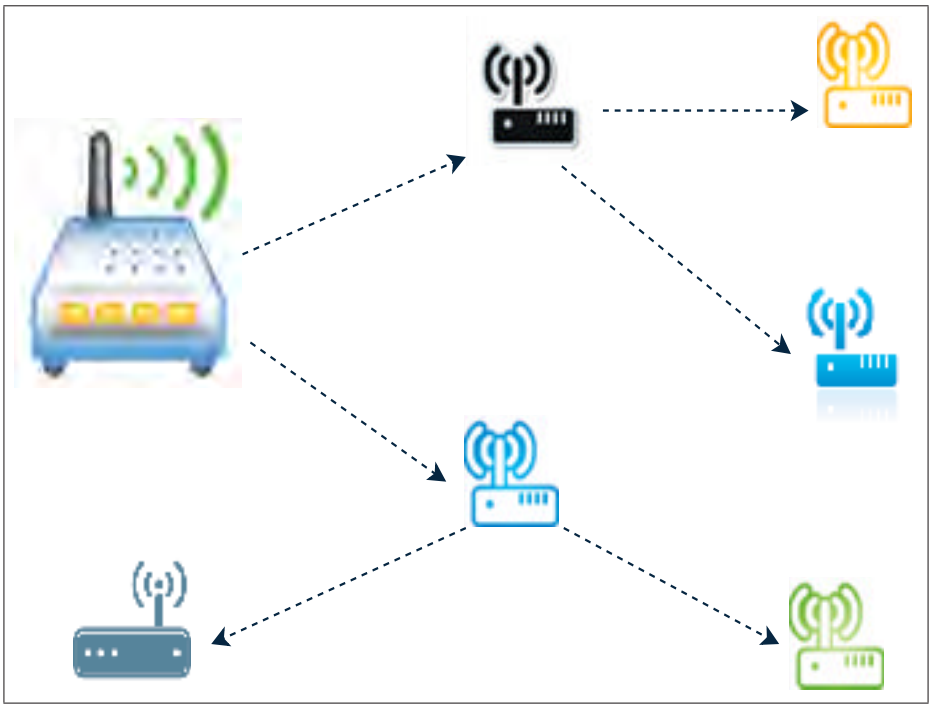


Figure 2.4 Peer-to-Peer update mechanism

2.3 IoT software run-time technologies

Many different execution environments have been developed to run on IoT sensor devices, ranging from virtual machines to component-based systems. Some run-time environments come with the purpose to facilitate programming (Boulis, Han, Shea & Srivastava, 2007), others are motivated by the potentiality of reducing energy costs for updating applications (Khan, Belqasmi, Glitho, Crespi, Morrow & Polakos, 2015). The choice of the run-time environment directly impacts on the format and size of data needed to transfer to an IoT device as well as the way software in this device can be updated. In this section we discuss four different mechanisms for executing program code in IoT sensor devices, including script languages, virtual machines, image-based and component-based systems.

2.3.1 Script environments

Script run-time environment is the kind of environments in which programs are loaded as scripts (e.g Python, Unix bash) and interpreted at run-time. These devices allow the execution of script languages by employing an interpreter to run the statements of scripts. There are many script run-time environments for embedded devices that have been introduced. For instance, Python is a well-known language which has been ported to microcontrollers (Norris, 2016). SensorWare (Boulis *et al.*, 2007) is another example, this framework provides a scripting environment and supports programming using a script language called TCL. In addition, the LiteOS operating system (Cao, Abdelzaher, Stankovic & He, 2008) provides a lightweight version of the Unix bash environment that has designed for sensor IoT platforms. However, Due to the string representation of script statements, script runtime environments require a significant amount of memory and CPU resources Ruckebusch *et al.* (2016).

2.3.1.1 Software update in script environment

Thanks to the runtime interpretation, in this kind of system, software can be updated after installation by adding or altering the scripts. Some frameworks like SensorWare also provides an algorithm to distribute the new scripts to IoT nodes. Furthermore, scripts can be inserted by external users and can contain replicate commands that allow a script to be replicated on other devices.

2.3.2 Virtual machines

Virtual machines are a kind of software systems which provides a run-time environment to execute intermediate code, that is high level CPU independent instructions. In such systems, software is written in intermediate code and translated to machine code at run-time by the virtual machine. Because intermediate code is in a high level of abstraction, the program code for virtual machines is often smaller than the native program code for physical machines, which reduces the data needed to transfer when deploying new applications to the devices.

However, due to the code translation at run-time, virtual machines present a substantial higher resource consumption compared to performing native machine code. In addition, because of the limitation of device memory, virtual machines are often optimally designed for specific applications rather than supporting many different application domains.

2.3.2.1 Software update for virtual machine

Virtual machines are a common approach in Wireless Sensor Network as well as in IoT devices, to reduce the cost of distributing new application code in the situation that the cost of data transmission is high. Because for devices running such environments, the new program code needed to transfer when updating is more compact than the physical machine code. Hence, many solutions have been proposed for IoT sensor devices that provide code updating using virtual machines such as Maté (Levis & Culler, 2002), Agilla (Fok, Roman & Lu, 2009) and VM Star (Razzaque, Milojevic-Jevric, Palade & Clarke, 2015).

Maté is one of the most well-known virtual machines for wireless sensor devices, which runs on top of TinyOS Amjad *et al.* (2016). Maté program code is divided into 24 code capsules, bigger programs can be provided using subroutine capsules. With Maté, updated code capsules are marked as self-forwarding and distributed to the network by Trickle dissemination protocol (Levis, Patel, Culler & Shenker, 2004). Each node broadcasts the information of its capsule version to its neighbours using a random timer. If a node hears a same version, it ignores the message and does not send any information; if it hears an older version than itself, it will broadcast the newer code capsules to the others. This broadcasting continues even after the whole network is updated.

2.3.3 Image-based software systems

This software system is an execution environment in which all source code, including software applications together with the operating system, is compiled into a single image and then installed on devices. In such systems, not like in virtual machines, native machine code can

be executed directly by the micro-controller of the constrained devices and no interpretation is required at run-time, allowing to avoid the high run-time overhead of both virtual machines and script execution. For this reason, image-based is preferred in the kind of devices with limited hardware and energy resources. However, installing native machine code on a device is more complex than loading code for a virtual machine because the native code uses physical addresses which typically need to be updated before the program can be executed.

2.3.3.1 Software update in image- based software systems

The update method for image-based systems is replacing the software image. Originally, the common way to replace images in image-based sensor devices is to compile a complete new image and overwrite the existing one, such as in Deluge (Hui & Culler, 2004). Full image replacement does not require extra processing of the new software image before it is loaded into the device, since the new image will be placed at the same physical address in the flash memory with the previous one.

Recently, instead of creating and distributing the whole new image, binary differencing techniques are often used to make the image replacement process more efficient. In these techniques, the delta file – the difference between the existing image and the new one is computed and delivered to devices, allowing to reduce the update size. When a device receives the delta, it processes and constructs the new image based on the delta and the old one, then starts the updated system. Differencing techniques are very effective for small updates in the software system. However, this approach often requires additional processing at the devices. There have been many image differencing methods are proposed in the literature, such as Hermes (Panta & Bagchi, 2012) and R3 (Dong et al., 2013b).

2.3.4 Component-based software systems

Recently, the common execution environment in advanced IoT devices is component-based, in which software is partitioned into a set of loadable components, which can be added or

updated at run-time. With component-based systems, only parts of the entire software need to be replaced during the update process, enabling to reduce the amount of data needed to transfer. In addition, the downtime is also lower because component-based approach does not require system reboot and the running state can be maintained during updates.

Typically, updating software components require support from the operating system in the devices. In such systems, the software always has a static part (e.g core OS kernel), and a dynamic part that includes a set of loadable components. A component includes native machine code and represents a functional module that provides a specific task of the overall system. So, the installation of new functionality or a bug fix is usually limited to a single or a small number of components. Nevertheless, the disadvantage of this execution environment is, it requires to disseminate symbol tables and relocation tables together with the component itself for linking and relocating steps (see 2.3.4.2), which can increase the amount of transferred data.

In this research project, we focus on the type of devices with component-based software system, because this kind of software is more and more popular in modern IoT devices.

2.3.4.1 Component dependency

In component-based systems, there are natural dependencies between software components in which some components may call the others during their execution. It is an important feature that needed to consider when updating every component, because an inconsistency error could occur if we update a component when the components it depends on have not all been updated. Hence, the order of updating components needs to satisfy all the dependencies.

2.3.4.2 Software update process in component based software systems

Run-time loadable software components are typically compiled and distributed as Executable and Linkable Format (ELF) files (Ruckebusch *et al.*, 2016). An ELF file normally includes the compiled code and data section of a component. During the update process, spaces in flash and RAM memory must be allocated for the new component, the relative addresses in the code

and data need to be replaced by the real physical address by relocation activity. If the code and data contain some undefined symbols (e.g. calling to functions or data declared in other components), a linking step is also required and each undefined symbol will be linked to the exact physical address.

A software program is written in Flash memory of devices as shown in Fig. 1.2 in Introduction chapter, components are arranged from low addresses high addresses. Each component accounts for a number of pages, which is the smallest unit that can be erased and written. It means that, even if only some bytes in a page need to be modified, the entire page needs to be re-written. When the device receives a component, it buffers the component in EEPROM (or RAM), then the flash pages are erased, and the corresponding new pages are transferred from EEPROM to the flash. When a component is updated, normally, the component size grows or shrinks, making a code shift that all the components lie after it in the flash need to be moved to other addresses. So not only the updated one, all those components have to be re-written despite the fact that they do not change anything in their functionalities. This code shift problem often consumes significant amounts of energy (Reijers & Langendoen, 2003).

2.4 Conclusion

This chapter introduced the background related to our work in this thesis. We have presented the concepts of IoT edge networks and the existing update mechanisms of IoT device, the typical hardware architecture of devices in IoT edge networks and the existing device's software run-time technologies.

CHAPTER 3

LITERATURE REVIEW

In this chapter, we summarize some previous work on software update for sensor/IoT networks, which can be categorized into three kinds that are dissemination protocols, data minimization and software run-time environments (Brown & Sreenan, 2013).

3.1 Update dissemination protocols

Data dissemination protocols focus on the ways to deliver software updates in the network, this topic often employs peer-to-peer communication between IoT devices, in which devices receive updates and transfer to others. Various protocols for update dissemination have been developed with the goal to minimize energy consumption of the entire network update process. The typical pattern of a dissemination protocol includes three steps: (i) the advertisement of new software ; (ii) the selection of download sources; and (iii) the downloading of the target nodes.

In (Dong & Yu, 2015), the authors propose an Adaptive Code Dissemination Protocol (ACDP) that employs random linear coding to reduce unnecessary computation and transmission cost. This protocol distributes the whole updated software image to sensor devices. A neighbor discovery scheme is proposed together with a source selection strategy, that allows a device to explore its neighbors to exchange data. Moreover, the protocol also provides a network coding technique to minimize the amount of data needed to transfer.

Before disseminating packets into the network, a node randomly generates a number N coefficients and calculates the linear combination of N packets, with N is defined as the size of coding window. An IoT sensor device gets a sufficient number (which is greater than or equal to N) of encoded packets and computes the original packets. ACDP reduces traffic by using the adaptive coding window in which a device dynamically chooses the size of its coding window based on the number of neighbors. The optimal value of N is a function of the network density.

The protocol also provides an effective load balancing feature that helps extend not only the lifetime of the entire network, but also the lifetime of each individual sensor device.

Triva (Saginbekov & Jhumka, 2014) is an other update dissemination protocol which focuses on event-based networks where data are sent in the network only when an event is detected. This protocol also aims to optimize the communication between devices to perform the update process with the goals to save both time and energy. Conceptually, Triva is a combination of Trickle (Levis *et al.*, 2004) and Varuna (Panta *et al.*, 2010) protocols. It works in such a way so as to enable nodes to update their code quickly, very much like in Trickle. However, the difference is it does not consume much energy in the steady state, like in Varuna, when there is no new update in the network.

In Triva, when a node n_1 completes downloading the new software, it tries to quickly deliver the update to its neighbors. n_1 broadcasts advertisement messages at random time in a given period. If, during this period, a neighbor node n_2 requests the new update, n_1 sends it to n_2. If n_1 obtains an advertisement with the same software version from a node n_2, it will save the ID of n_2 in its neighborhood table. After broadcasting advertisement messages for a given period, the node stops broadcasting and change to the steady state to save energy, like in Varuna. The steady period is when all the nodes in the network have the same software version and no dissemination is performed.

3.2 Update minimization methods

Besides software dissemination protocols, data minimization is also an important topic in IoT device updating. Data minimization focuses on reducing the size of updates, it has a direct impact on the communication and processing energy used, and therefore it both helps extend sensor network lifetime, and decrease time for new software deployment. Many methods are proposed based on the idea of transferring only delta files, the differences between the old and the new software versions, to reduce the transmitted data.

The authors in (Panta *et al.*, 2011) introduce a software update scheme called Zephyr. It decreases the delta size by using a function table for indirect function calls, that mitigates the impact of code shifts in flash memory and increases the similarity between the two software versions. Then it compares the two versions at the byte level to generate a small delta file. Zephyr improves data size minimization by performing the modifications of the update software on application-level. Zephyr employs a modified version of the Rsync algorithm (Panta *et al.*, 2011) to create delta files, which is the differences between the old and new software versions. The overview of Zephyr is described in Fig. 3.1, which is from (Panta *et al.*, 2011).

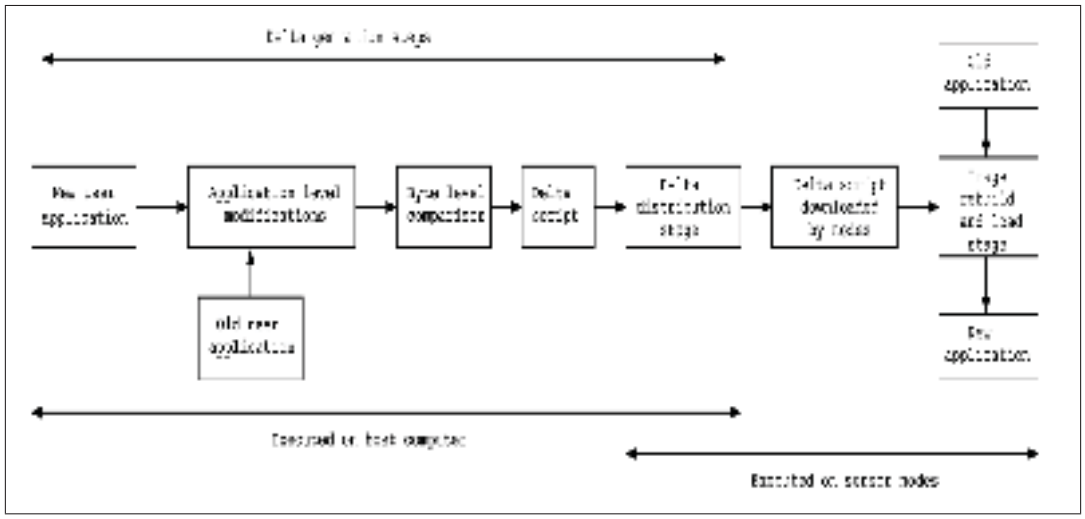


Figure 3.1 Overview of the Zephyr update scheme

Dissemination in Zephyr has two stages: first all the sensor devices are requested to reboot the component that need to be updated, then a dissemination protocol is used to deliver the update to all nodes. After the update has been applied to construct the new image, and this new image is loaded into the flash memory, all indirect function calls are replaced by direct ones to improve performance. The devices then need to reboot to start the new image. Zephyr does not specifically support for autonomous update, but it emphasizes a very significant reduction in data size that makes software updates are more flexible in general.

Hermes (Panta & Bagchi, 2012) is another update scheme built over Zephyr. Not only focusing on reducing impacts of code shifts, Hermes also mitigates the effects of data shifts in RAM

by first fixing variables to the same locations by source code level modifications, and then propose two different approaches for reducing data shifts. This scheme scans through the program source code before calling the compiler, it puts initialized and uninitialized variables into assigned structures, so that their order is preserved when the compiler creates the two sections .data (for initialized variables) and .bss (for uninitialized variables). The overview of Hermes is described in Fig. 3.2 (Panta & Bagchi, 2012), with the dashed rectangles are presenting the new features of Hermes compared to Zephyr.

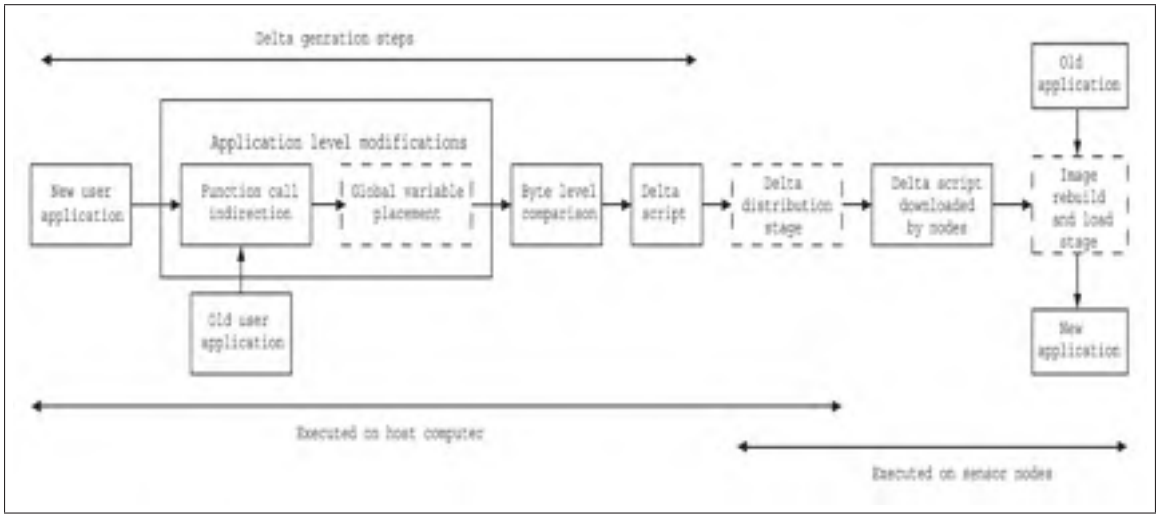


Figure 3.2 Overview of the Hermes scheme

We note that both Zephyr and Hermes can be used in component-based software architecture of IoT devices. Similar to Zephyr, Hermes performs software comparison on byte level to generate the delta files. However, different from Zephyr, Hermes also provides a transparent update feature, in which a new version of a software component can run in parallel with the old one until it collects enough states to transparently take over the operation. This feature provides a smooth update process to software components, which support autonomous update.

3.3 Software update in different component-based execution environments

The execution environment such as virtual machine (Kovatsch, Lanter & Duquennoy, 2012), image-based and component-based (Taherkordi *et al.*, 2013), also has a significant impact on

how the software in an IoT device can be updated. Recently, there are many studies (Ruckebusch *et al.*, 2016), (Munawar *et al.*, 2010) investigate on component-based software systems, to improve the update process.

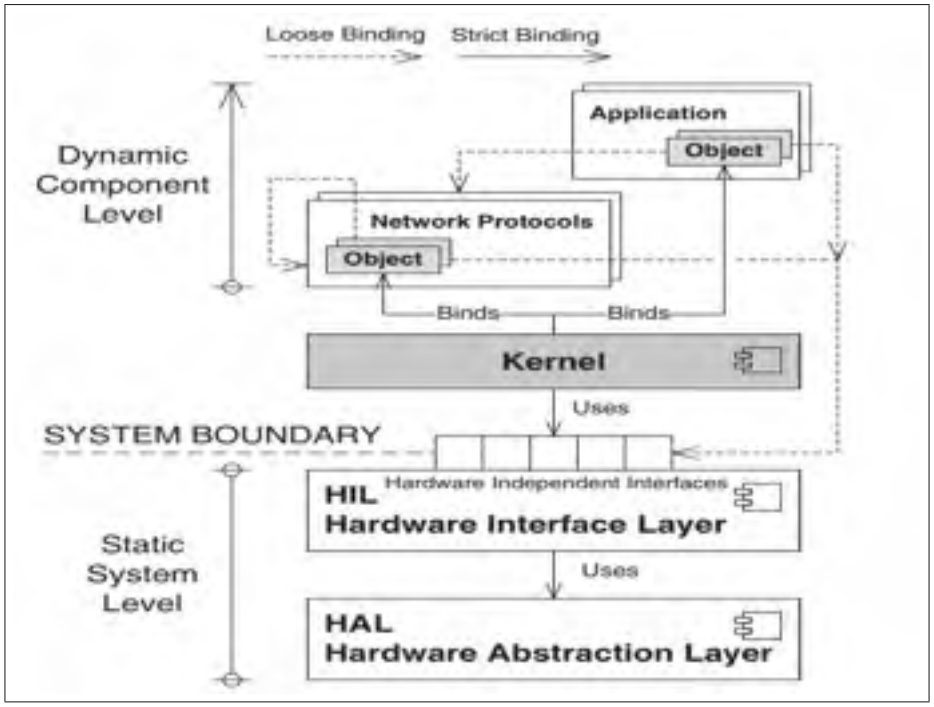


Figure 3.3 Overview of the Gitar architecture

Gitar (Ruckebusch *et al.*, 2016) is an example, this architecture is built on top of Contiki OS to reduce operation overhead during updating components. Gitar also enables software in both application and network levels is updated in an efficient way. As shown in Fig. 3.3 (taken from (Ruckebusch *et al.*, 2016)), Gitar architecture includes three levels: system level, kernel level and component level. The system level is partitioned into two layers, a hardware abstraction (HAL) layer and a hardware interface (HIL) layer, in order to improve software portability. This level involves the operating system and hardware drivers is static and can only be updated by replacing the entire software image. On the other hand, the software components at the component level (i.e. network protocol and application components) are flexible and can be dynamically updated at runtime. Finally, the kernel level is the middle level between the system and component levels. Gitar uses a *loosely coupled binding model* in which compo-

nents are called indirectly through their references, that allows each component can be updated separately without affecting other components.

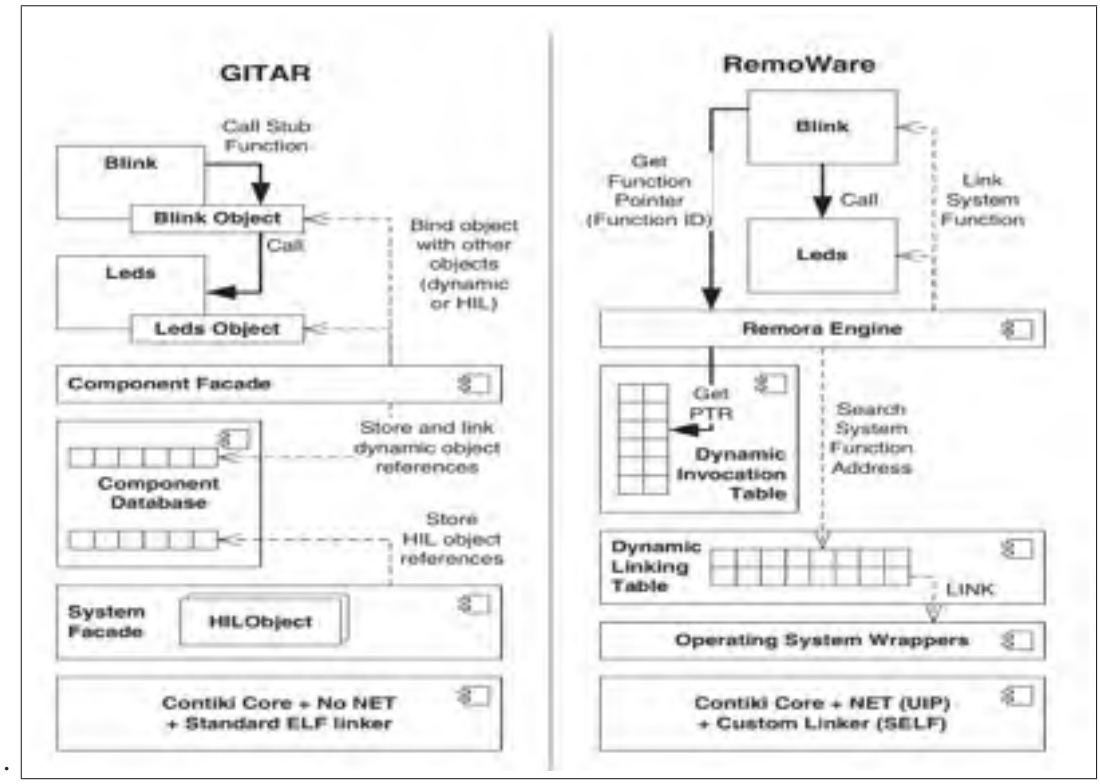


Figure 3.4 Comparison between Gitar and Remoware

In another work (Taherkordi *et al.*, 2013), a component-based middleware for IoT sensor devices named RemoWare is proposed. This middleware mitigates the cost of software update deployment on devices by the notion of in situ reconfiguration and provides a component-based programming abstraction in order to facilitate the development of IoT applications. It has rich features to support dynamic software update including component distribution and runtime linking that allows to make changes only in individual components, thus saving resource usage overhead and energy consumption. The main difference between RemoWare and Gitar is the way of function calls, which is illustrated in Fig. 3.4 (Ruckebusch *et al.*, 2016).

In RemoWare, all the function pointers are stored in a dynamic invocation table (DIT) with a one-to-one mapping relation with function IDs. The calls to the function pointers are actually

performed by the kernel. RemoWare combines a strict binding model for system level and a loosely coupled binding model in the dynamic component level, the combination of both models cause a fixed memory overhead for operating the system functions regardless of the number of software components.

3.4 Discussion

In this section, we present a brief summary of the relevant studies mentioned above and then compare the characteristics of those studies to our research. The table 3.1 highlights the main differences between these searches and their limitations.

Unlike the prior studies above, our work focuses on optimizing the total energy consumption of the update process in the entire network, considering the case that multiple software components are updated. We focus on the flash re-writing of the update process inside a device, which is a main energy consuming operation (Koshy & Pandey, 2005b) (Heo *et al.*, 2010), and take into account the component update order in every IoT device.

Table 3.1 Comparison of previous work

Research work	**Objective**	**Methodology**	**Limitations**
ACDP (Dong & Yu, 2015)	Minimize unnecessary computation and communication cost of update process in the device network.	A neighbor discovery scheme and a source selection strategy, together with a random linear network coding method to reduce unnecessary computation and transmission cost.	Devices need to communicate and exchange a lot of messages. Does not consider the update process inside a device.
Triva (Saginbekov & Jhumka, 2014)	Minimize both the time and energy consumption of the update process in even-based networks.	A combination of Trickle and Varuna protocols to reduce energy consumed in the steady phase.	Devices still need to exchange advertisement messages. Does not consider the update process inside a device.
Zephyr (Panta *et al.*, 2011)	Reduce size of delta files, mitigate impact of code shifts in flash memory.	Comparing the two software versions at the byte level to increase the similarity and generate a small delta file.	Only the software in a single device is considered. Does not consider updating multiple components.
Hermes (Panta & Bagchi, 2012)	Reduce size of delta files, mitigate impact of code shifts in flash memory and data shifts in RAM.	Based on Zephyr, but it fixes variables to the same locations by source code level modifications.	Only the software in a single device is considered. Does not consider updating multiple components.
Gitar (Ruckebusch *et al.*, 2016)	Facilitate the replacing of a component, reduce operation overhead.	Using a loosely coupled binding model in which components are called indirectly through their references.	Does not optimize the entire network. Does not consider the flash re-writing. Does not consider updating multiple components.
RemoWare (Taherkordi *et al.*, 2013)	Facilitate the replacing of a component, reduce operation overhead.	Combining a strict binding model for system level and a loosely coupled binding model in the dynamic component level.	Does not optimize the entire network. Does not consider the flash re-writing and the case of updating multiple components.

CHAPTER 4

METHODOLOGY

This chapter presents the methodology of our research project. We first introduce the system description, which includes the illustrations of the component-based software system in IoT devices and the IoT networks under consideration. Then, the energy model of the update process inside a device is presented, followed by the optimization formulation of our scheduling problem. Finally, we introduce our proposed algorithm, called ESUS, to approximate the optimal update schedule in polynomial time.

4.1 System description

4.1.1 Assumptions

In this subsection, we summarize the assumptions that are made in our study. We consider the case in which a gateway downloads software updates from a server running on the cloud, and then sends to a number of devices of the same type (i.e., having the same hardware and software configuration) which employ the component-based software architecture. A device does not need to update all new components at a time, but one by one. During the update period, the device can maintain operation with both old and new components, in other words, at a moment, some components are completely updated, and some others are still keeping the old version. Since a component may call some others, their dependency causes the order constraints that need to be satisfied by the update schedule, in which a component can only be updated when the components it depends on had all been updated.

A device receives components from both the gateway and other devices in a P2P manner. It can download from or send to multiple nodes at the same time, but can only download at most one component from one corresponding node at a time. A device can only send a component to other devices after it completes downloading this component. We assume that the network is stable during the update period, it means that no new nodes come and the connections are

unchanged. The installation time is constant for each component, and therefore this time can be skipped and is not considered in our model. In the update process in the flash memory of a device, we assume the energy consumption in a flash re-writing operation is much bigger than other operations, and is proportional to the number of re-written pages in this operation.

For simplicity, we also assume that all the devices use the same low-energy communication technology (e.g. Zigbee) in the same environment conditions, and the distances between them are not much different (such as in smart homes). Therefore, the energy consumed during data transmission is not significantly affected by the distances and is proportional to the amount of data. Hence, the total communication energy cost of the network can be considered as constant because the total amount of transmitted data is fixed and does not depend on the update schedule. The more realistic assumptions will be taken into account in our future work.

4.1.2 IoT component-based software model

The component-based software system in each IoT device can be considered as a directed acyclic graph $D =< V_D, A_D >$ with V_D is the set of components and A_D is the set of arcs which presents component dependencies. The graph D can be represented by a matrix $\mathscr{M}_D = \{c_{m,n}\}$ where each binary entry $c_{m,n}$ $(m,n \in V_D)$ with value 1 denotes an arc $(m,n) \in A_D$, means that a component m is called by component n. An example of such a graph is presented in Fig. 4.1b. In this example, component d calls three components a, b and c, and the device can update component d only after completing the updates of three others.

As we mentioned in the previous chapters, each component occupies a number of memory pages (Fig. 4.1a), which is the smallest unit that can be erased and written. The modification of any byte in a page will result in the entire page needs to be re-written. We describe an re-writing flash operation by an example in Fig. 4.2, a set of components a, b, c and d that are located in a sequence in the flash. When c is updated, suppose that the new size of c increases compared to the previous size (c' is bigger than c), it leads to a code shift in which all the components lie after c in the memory will have to be shifted to higher addresses. Thus, even

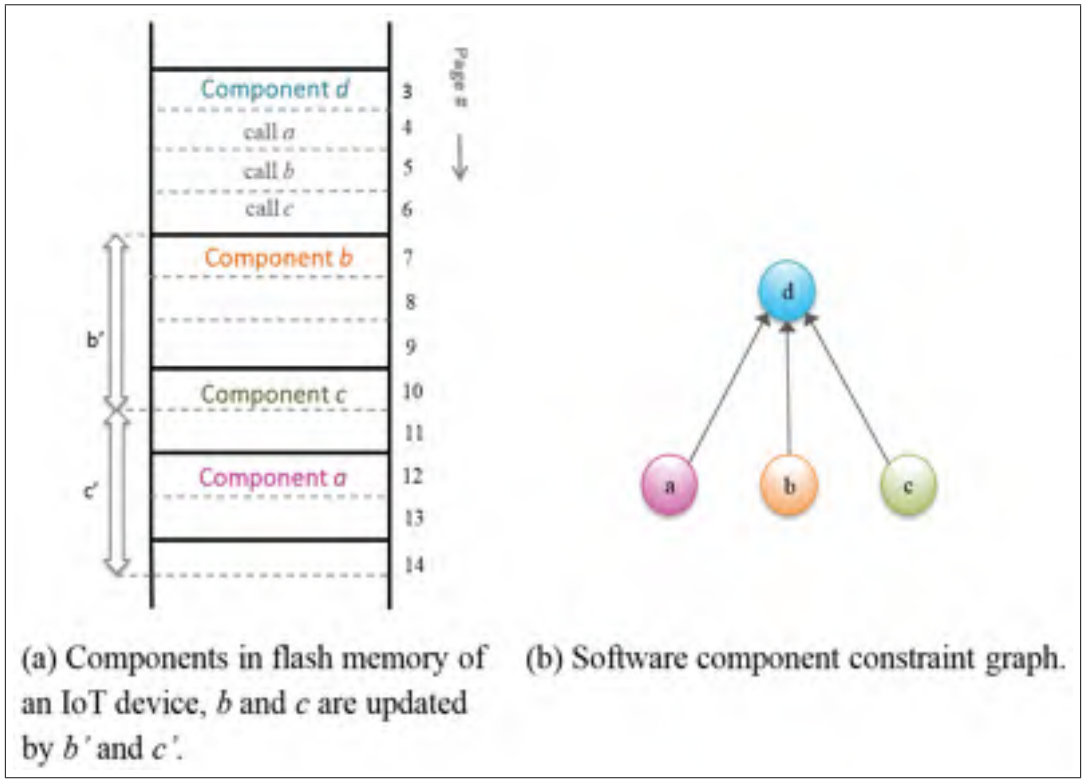


Figure 4.1 Software components in flash memory of an IoT device and the corresponding component dependency graph

if *a* is not in the update list, it will be re-written in a new location. There is a call from *d* to *a*, the address of this call instruction needs to be altered and the corresponding page - the page number 4 in Fig. 4.2 has to be re-written. By minimizing code shifts - the number of flash pages need to be re-written, we can reduce the energy consumed in the update process.

4.1.3 System model

We focus on a model of an IoT edge network including a number of connected component-based IoT devices and a gateway. A software update is a set of components which is distributed from the gateway to all devices in a P2P manner, that is devices can exchange components with others after receive from the gateway. The gateway manages IoT devices and is responsible for scheduling updates for the devices. By caching the update in the gateway (Brown & Sreenan, 2013), devices do not have to get the new software from the Internet.

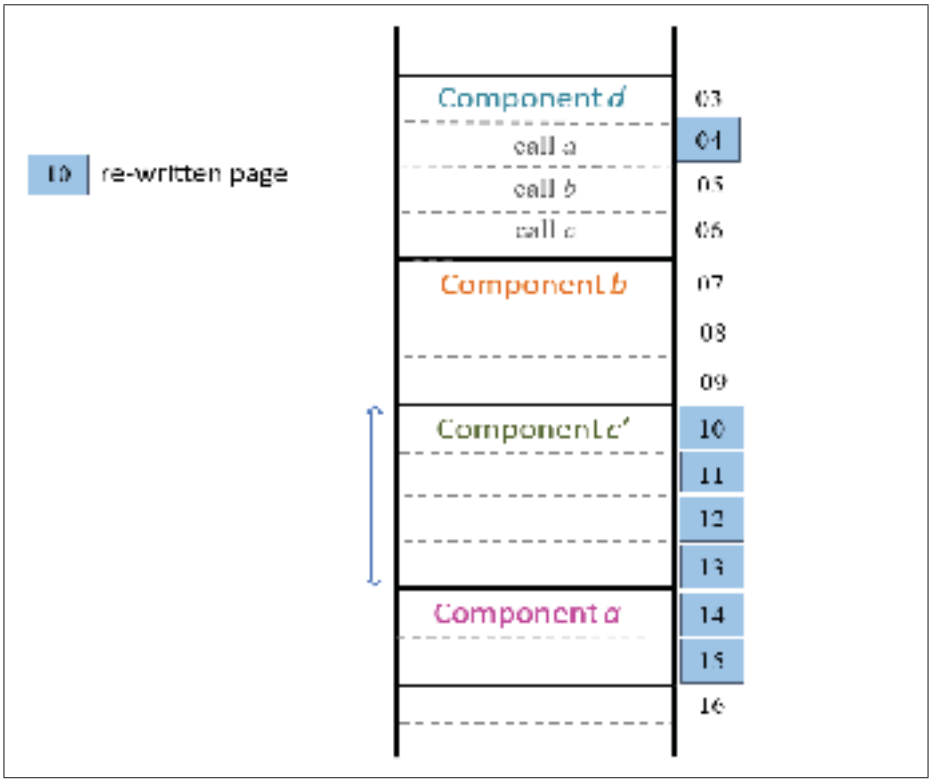


Figure 4.2 Re-written pages when updating component c

We illustrate the network system by a graph $G =< V_G, E_G >$ in which both the gateway and devices are considered as "nodes", with V_G is the set of vertices and E_G is the set of edges representing nodes and links, respectively. Let $V_G = \{i|\ i = 0, 1, \ldots |V_G|\}$, in which $i = 0$ represents the gateway, and IoT devices are corresponding to $i > 0$. We represent G by a symmetric matrix $\mathscr{M}_G = \{b_{i,j}\}$ where each entry $b_{i,j}$ specifies the bandwidth of the link between two nodes i and j. We denote by $b_{i,j} = 0$ if there is no link between the two nodes. An example of such graph with the corresponding matrix is shown in Fig. 4.3.

The update process of the entire network is implemented as follows: At the beginning, the gateway stores all components, it calculates the schedule and follows this schedule to control the update process. At each scheduled time, the gateway sends a message to each assigned device to specify that the device can download which component from which source. The process finishes when every node has all the new components. Since the total update time is

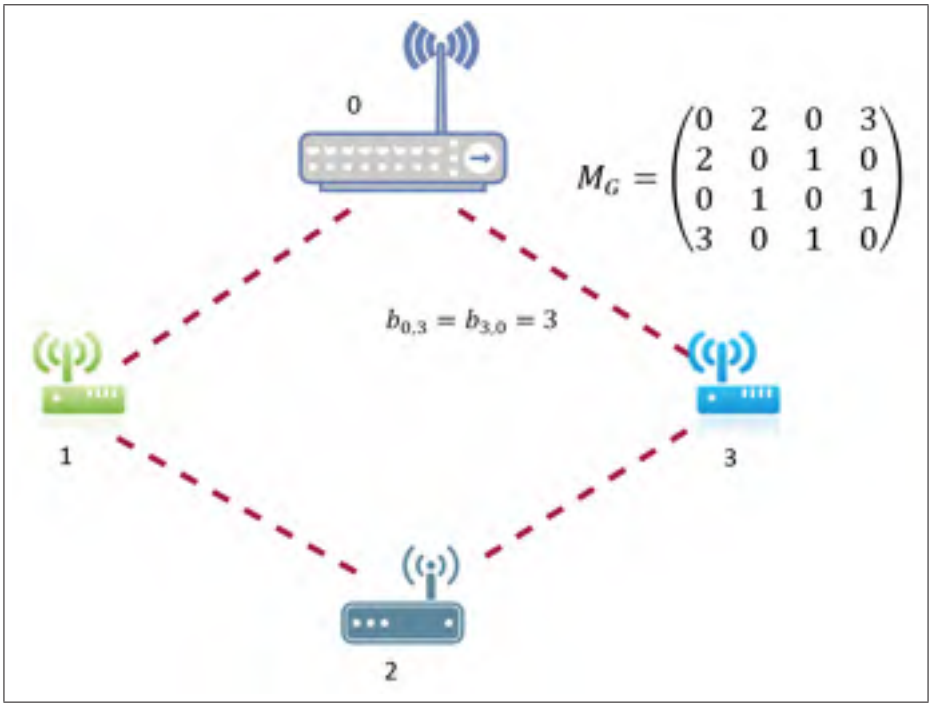


Figure 4.3 A graph presenting an IoT edge network with the corresponding matrix

also important, it is necessary to limit the amount of time to perform the update in the entire network by a deadline T_{max}.

In our scheduling problem, we need to find an optimal update schedule for the update process in the entire network, which minimizes the total energy consumption while satisfying the five constraints: (i) the dependency order of components, (ii) the network topology, (iii) a device can only send a component after having it, (iv) a device can download at most one component from one source at a time, and (v) the deadline constraint of updating the entire network. Such an optimized schedule is computed by a centralized controller running in the gateway (Barcelo, Correa, Llorca, Tulino, Vicario & Morell, 2016), as presented in Fig. 1.1 in chapter 1.

4.2 Problem formulation

In this section, we present the energy model for the update process in a component-based IoT device and the optimization model of our energy-efficient software-update scheduling problem.

4.2.1 Decision variables

We define two sets of decision variables used in our optimization model. Let $a_{i,j,m}$ be a binary variable that equals to 1 if device i downloads component m from gateway/device j, and let $x_{i,m}$ be the start time at which device i downloads component m. The update schedule of each device i is characterized by the sets $\{a_{i,j,m}\}$ and $\{x_{i,m}\}$.

Fig. 4.4 shows a part of an update schedule corresponding to the downloading of two components of device 3. The device downloads components b and c from the gateway 0, so $a_{3,0,b}$ and $a_{3,0,c}$ are both equal to 1. The start times to download are 4 and 0, respectively, then we have $x_{3,b} = 4$ and $x_{3,c} = 0$.

4.2.2 Energy consumption model

This sub-section is dedicated to addressing the **SO1**, we give details about the energy consumption model of the update process. As mentioned in the *Introduction* chapter, each new component is buffered in a dedicated space in EEPROM and then written to the flash. We denote the size of the update of a component $m \in V_D$ by s_m^{new}, and the size of m before update by s_m^{old}. The duration of device i to completely download component m can be calculated as follows:

$$t_{i,m} = \begin{cases} 0, & i = 0, \\ \dfrac{s_m^{new}}{\sum\limits_{j \in V_G} a_{i,j,m} b_{i,j}}, & i > 0. \end{cases} \tag{4.1}$$

In (4.1), $t_{i,m}$ is 0 if device i is the gateway, otherwise $t_{i,m}$ is calculated by dividing the size of m by the corresponding bandwidth. The amount of energy consumed when a device i updates a

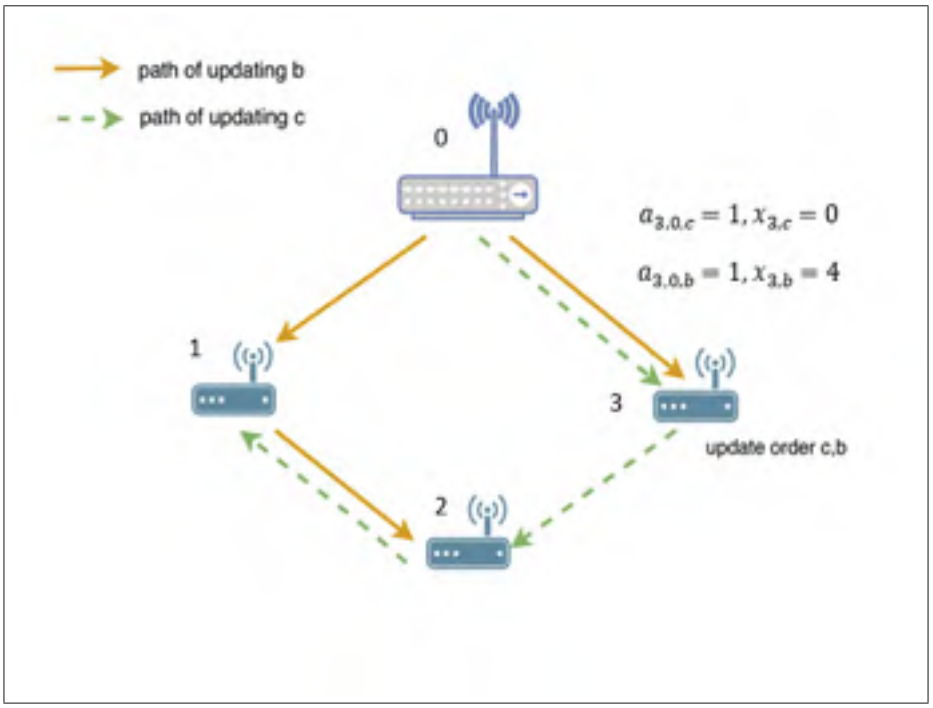


Figure 4.4 Decision variables correspond to downloading 2 components of a device

component m can be calculated by multiplying the energy for writing one flash page with the number of re-written pages, as follows:

$$E_{i,m} = e \times \left(\frac{s_m^{new}}{\rho} + \lambda_m \left(\sum_{h \in \alpha(m)} \frac{size(h)}{\rho} + \sum_{h \in \alpha(m)} \sum_{k \in \beta(m)} c_{h,k} \right) \right), \tag{4.2}$$

where e is the energy consumption for writing one page, ρ is the size of one page, λ_m is a binary indicator that equals to 1 if $s_m^{new} \neq s_m^{old}$, because if m does not change its size ($s_m^{new} = s_m^{old}$), we do not need to shift the following components. $\alpha(m)$ is the set of components lie after m and $\beta(m) = V_D \setminus (\alpha(m) \cup m)$ is the set of components lie before m in the flash memory. The binary indicator $c_{h,k}$ is an entry in the matrix $\mathscr{M}_D$ that equals to 1 if the arc $(h,k) \in A_D$, means that k depends on (calls) h; in this case, when shifting h to new address, we need to re-write the (one) page in k that contains the instruction calling h. And $size(h)$ is the size of component h at the moment of updating m, i.e., $size(h)$ is s_h^{new} if h is updated before m, otherwise $size(h)$ is s_h^{old}.

We calculate $size(h)$ as:

$$size(h) = s_h^{new}\delta_{h,m} + s_h^{old}(1-\delta_{h,m}), \tag{4.3}$$

where the variable $\delta_{h,m}$ indicates that h is updated before m or not:

$$\delta_{h,m} = \begin{cases} 1 & x_{i,h} + t_{i,h} < x_{i,m} + t_{i,m}, \\ 0 & otherwise. \end{cases} \tag{4.4}$$

Given a device with a fixed number of components, we can see that the quantity $\sum_{h\in\alpha(m)}\sum_{k\in\beta(m)} c_{h,k}$ in equation (4.2) is constant and does not depend on the update order. Since we want to find an optimal update order to reduce the number of re-written pages, we can skip this quantity without affecting our scheduling solutions. Also, with the assumption that component sizes always change, means that $s_m^{new} \neq s_m^{old}, \forall m \in V_D$, so λ_m is always 1, then we can have the simplified form of $E_{i,m}$ as:

$$\bar{E}_{i,m} = \frac{e}{\rho}\left(s_m^{new} + \sum_{h\in\alpha(m)}\left(s_h^{new}\delta_{h,m} + s_h^{old}(1-\delta_{h,m})\right)\right). \tag{4.5}$$

The value of $\bar{E}_{i,m}$ depends on each component $h \in \alpha(m)$ is updated before or after updating m.

The energy E_i consumed when device i updates all new components is:

$$E_i = \sum_{m\in V_D} \bar{E}_{i,m}. \tag{4.6}$$

In Eq. 4.6, E_i is a function of $\{a_{i,j,m}\}$ and $\{x_{i,m}\}$.

4.2.3 Optimization model

In this sub-section, we tackle the sub-objective **SO2** by illustrating our optimization model for the energy efficient scheduling problem. For the convenience of discussion, we summarize the mathematical notations used in our model in table 4.1.

Table 4.1 Notation

Notation	Description
V_G	Set of nodes (gateway and IoT devices)
V_D	Set of software components
s_m^{new}	New size of component m
s_m^{old}	Current size of component m
T_{max}	The deadline for all devices complete updating
$t_{i,m}$	Duration that a node i completely downloads component m
$b_{i,j}$	Bandwidth of the link between two devices i and j
$c_{m,n}$	Binary indicator indicating component n calls component m
Decision variables	
$a_{i,j,m}$	Binary variable equals to 1 if device i downloads component m from device/gateway j
$x_{i,m}$	Start time device i downloads component m

In our model, the objective function aims to minimize the total energy consumption of all the devices during the update process, as in the following expression:

$$\min \sum_{i=1}^{|V_G|} E_i. \tag{4.7}$$

Constraint (4.8) indicates that each start time needs to be greater or equal to 0.

$$x_{i,m} \geq 0, \quad \forall i \in V_G, i > 0, m \in V_D. \tag{4.8}$$

The gateway gets new software from the cloud, and then it acts as a source to distribute the updated software to the whole network. It means that the gateway does not download from any nodes in the network, as presented in condition (4.9).

$$x_{0,m} = 0, \quad \forall m \in V_D. \tag{4.9}$$

During the update process, a device only downloads each component m once from another node, this specification is indicated in Eq. (4.10).

$$\sum_{j \in V_G} a_{i,j,m} = 1, \quad \forall i \in V_G, i > 0, m \in V_D. \tag{4.10}$$

Constraint (4.11) is the network topology constraint, a device i can download from device/-gateway j only if there is a link (i, j);

$$a_{i,j,m} \leq \phi(b_{i,j}), \quad \forall i, j \in V_G, i > 0, m \in V_D. \tag{4.11}$$

where $\phi(b_{i,j}) = 1$ if $b_{i,j} > 0$, means that link (i, j) exists, otherwise $\phi(b_{i,j}) = 0$ if $b_{i,j} = 0$.

A device i can only download a component from a device j after j completes downloading this component, this constraint is described in condition (4.12)

$$a_{i,j,m}(x_{i,m} - (x_{j,m} + t_{j,m})) \geq 0, \quad \forall i, j \in V_G, i > 0, m \in V_D. \tag{4.12}$$

where the download duration $t_{j,m}$ is calculated by formula (4.1).

In our problem, we suppose that in a certain link (i, j), there is at most one component is transferred at any moments.In other words, a device can only download one component from each other node at a time, it is stated in constraint (4.13)

$$a_{i,j,m} a_{i,j,n}(x_{i,m} - x_{i,n} - t_{i,n})(x_{i,n} - x_{i,m} - t_{i,m}) \leq 0, \quad \forall i, j \in V_G, m \neq n \in V_D. \tag{4.13}$$

For the software component dependency, as mentioned before, a device can update a component m if and only if all the component called from m are already updated. So, we have the constraint (4.14) indicates that the component download order of each device needs to satisfy

the component dependency graph.

$$c_{m,n}(x_{i,n} - (x_{i,m} + t_{i,m})) \geq 0, \quad \forall i \in V_G, m, n \in V_D. \tag{4.14}$$

And finally, condition (4.15) is the deadline constraint, means that all the component downloads need to complete before a deadline T_{max}.

$$x_{i,m} + t_{i,m} \leq T_{max}, \quad \forall i \in V_G, m \in V_D. \tag{4.15}$$

Due to the constraints (4.12), (4.13) and the discrete objective function, our optimization problem is an Integer Non Linear Programming (INLP) problem. Since the complexity is very high, finding the optimal solution with a solver is very time consuming, we design an algorithm to solve the problem in the next section.

4.3 Proposed Algorithm

This section addresses the sub-objective **SO3**, which is about building a fast algorithm for our scheduling problem. We design an algorithm called ESUS, which stands for Energy-efficient Software Update Scheduling, to approximate the optimal solution of our optimization problem. Our proposed algorithm employs a procedure P_1 to generate an initial update schedule without taking the deadline constraint T_{max} into consideration. In case T_{max} is violated by the initial solution given by P_1, ESUS uses another procedure, called P_2, to properly adjust the schedule to reduce the overall update time. These two steps are repeated in a number of iterations with the purpose to find a near-optimal solution that satisfies the deadline.

4.3.1 Procedure P_1

The outline of P_1 is described in Algorithm 4.1, this procedure is based on the idea of dividing the schedule into steps. At a single step, each device i maintains a list of downloadable components $\mathscr{L}_i$, it is the set of components that the device can update at this moment; and a list of possible sources $\mathscr{S}_i$ (other devices or the gateway), where the device can get those components.

The two lists can be represented as a bipartite graph B_i with each edge join a component m in $\mathscr{L}_i$ with a node j in $\mathscr{S}_i$, indicating that i can download component m from j. A matching of B_i represents an assignment of sources-to-components at this step, that is corresponding to a set of values of $a_{i,j,m}$.

Algorithm 4.1 Procedure P_1 - Generate an initial schedule

Input: *Network matrix $\mathscr{M}_G$, software component matrix $\mathscr{M}_D$, software component sizes $\{s_m^{new}\}$ and $\{s_m^{old}\}$*
Output: *final set of download assignments $\{a_{i,j,m}\}$, final set of download time$\{x_{i,m}\}$*
repeat
  Each step, **do**
  for *each device i* **do**
    Construct the bipartite graph $\mathscr{B}_i$;
    Do *Matching* the bipartie graph $\mathscr{B}_i$;
    With $\{m\}$ is the set of downloaded components given by *Matching*, set each $x_{i,m}$ is the finishing time of the *previous step*, then adjust $\{x_{i,m}\}$ so that $\{x_{i,m}+t_{i,m}\}$ has the order as in the flash;
  end
  Calculate the finishing time of this step;
until *all nodes complete downloading all components*;

Algorithm 4.2 Matching algorithm

Input: *Bipartite graph $\mathscr{B}_i$ with list of downloadable components $\mathscr{L}_i$ and list of available source $\mathscr{S}_i$*
Output: *a matching of $\mathscr{B}_i$ with maximum number of downloaded component*
Sort component list in ascending order by number of available sources
for *each component m in the list $\mathscr{L}_i$* **do**
  if *There are some sources that are other devices* **then**
    Choose the best source (highest bandwidth) between these devices and match with this component;
  end
end
if *There are some components that can only be downloaded from gateway* **then**
  Randomly **choose** one among these components;
end

P_1 uses a function to find a matching of B_i with the aim to maximize the number of components can be downloaded in the step. To do that, the *Matching* function sequentially chooses the

component that has the smallest number of sources, then randomly assigns a source to this component and updates source lists of other components, as described in Algorithm 4.2. After matching, P_1 calculates $x_{i,m}$ for each downloaded component m so that the order of download complete time (that is $x_{i,m} + t_{i,m}$) is same as the order of components in the flash, that helps reduce the number of re-written pages. This idea is based on the update order example in Introduction chapter.

An example of a bipartite graph is shown in Fig. 4.5, at this step, device 2 has three downloadable components a, b, and c that lie in its flash as in Fig. 4.1a. With this graph, it can get all the components by downloading a from the gateway, b from device 1 and c from device 3. In this case, P_1 adjusts $x_{2,a}$, $x_{2,b}$ and $x_{2,c}$ so that the device 2 completes downloading b first, then c and a, according to the order in the flash.

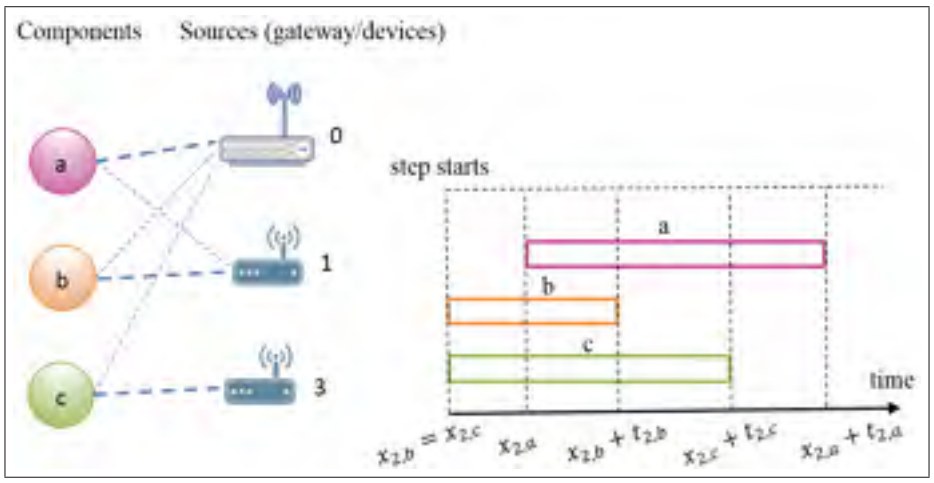


Figure 4.5 A bipartite graph presenting downloadable components of device 2

4.3.2 Procedure P_2

The procedure P_2 does not change the assignments of download sources (i.e. the variables $a_{i,j,m}$). Instead, it analyzes the current schedule and shifts the download time $x_{i,m}$ to the earliest as possible. P_2 sequentially executes on each component m, it checks the routes of disseminating m in the network. For each device i that downloads m, P_2 checks if the start download

Algorithm 4.3 Procedure P_2 - Adjust a schedule

```
1  Input: A schedule 𝒮, component dependency graph 𝒟
2  Output: A schedule 𝒮′ that has the finishing time less than 𝒮
3  Find a topological order of components in graph 𝒟
4  for each component m in the topological order do
5      𝒯_m is the tree presents paths of distributing m in the network
6      Traverse 𝒯_m by Breadth-first search and sequentially add visited nodes in a list 𝒱_m
7      for each node i in the list 𝒱_m do
8          if the time {x_{i,m}} can be moved to an earlier moment then
9              Adjust {x_{i,m}} to the earliest as possible
10         end
11     end
12 end
```

time $x_{i,m}$ can be moved to an earlier one. That is, if the source of i has m sooner than $x_{i,m}$, and if i have updated all the necessary components called by m before $x_{i,m}$, then P_2 shifts $x_{i,m}$ to the earliest as possible. P_2 iterates the components in a topological order, it means that when considering a component m, all the components that m depends on have been already adjusted. The outline of P_2 is presented in Algorithm 4.3.

Algorithm 4.4 ESUS Algorithm

```
1  Input: Number of iterations N, deadline T_max, network matrix 𝒜_G, software
       component matrix 𝒜_D, software component sizes {s_m^new} and {s_m^old}
2  Output: final set of download assignments {a_{i,j,m}}, final set of download time{x_{i,m}}
3  for t from 1 to N do
4      Generate schedule S_t by P_1;
5      if S_t does not satisfy Tmax then
6          Adjust S_t by P_2;
7      end
8      if S_t still does not satisfy Tmax then
9          Start new iteration t + 1;
10     end
11     else
12         if S_t is better than current best solution then
13             Update the best solution is S_t;
14         end
15     end
16 end
```

4.3.3 ESUS algorithm

Our main algorithm - ESUS is presented in Algorithm 4.4. Due to the randomness of P_1, ESUS performs the two procedures in a number N of iterations and selects the best solution. The bipartite graph construction procedure has complexity $O(M^2 \times N)$ with M is the number of components and N is the number of nodes. The matching procedure requires $O(M \times N)$ steps. And the procedure P_2 requires $O(M^2 \times N)$ steps. So, in general, ESUS algorithm has polynomial complexity.

4.4 Conclusion

In this chapter, we have presented the research methodology. We first introduced the descriptions of the component-based software system in IoT devices and the IoT networks under investigation. Then, we proposed an energy model of the update process inside a device, based on the analysis of updating a component-based software system in the flash memory. The energy model is followed by the optimization formulation of our scheduling problem, with the objective of minimizing the total energy consumption of updating the entire network, while satisfying a deadline constraint. Finally, we presented our proposed algorithm to find a near optimal update schedule in polynomial time.

CHAPTER 5

EVALUATION RESULTS

This chapter is dedicated to the simulation results of our research project, in which we examine the proposed scheduling algorithm in different network instances. We first describe our evaluation methodology, after that, the network settings and the optimization parameter settings are presented. Then, we illustrate different experimental scenarios, together with corresponding results and discussions.

5.1 Evaluation methodology

We evaluate the efficiency of ESUS in three typical network topologies of IoT that are tree, partial mesh and full mesh. For each topology, we define different network configurations by varying the number of IoT devices from 10 to 30, and randomly creating various software component sets from 5 to 9 components. These configurations are suitable for common IoT applications such as smart buildings or smart homes.

To evaluate results of ESUS algorithm, we use the CPLEX solver (CPLEX, 2019) to obtain the optimal update schedules on all the network configurations. The optimal results are compared to the solutions given by ESUS. We additionally evaluate the running time of our algorithm and the solver to examine the algorithm's time complexity.

Besides the optimal results, we also employ CPLEX to find a random feasible schedule for each network configuration. A feasible schedule is a schedule that satisfies all the constraints but does not minimize the energy objective function. We consider the random scheduling is a simple method for finding update schedules and can be used as a baseline to evaluate our algorithm. We calculate the energy consumption of those random schedules and compare them to results of ESUS. The gap between the results of our algorithm and random scheduling can show the efficiency of our approach.

Our evaluation method can assess the proposed algorithm in significant aspects. First, it can show that if ESUS algorithm can compute schedules that are close to the optimal ones. Second, the evaluation method can evaluate the running-time efficiency of ESUS compared to the solver when solving a high complexity problem. And finally, the network configurations used in the simulations help us to see if our algorithm can work effectively in practical network topologies that simulate real IoT applications.

The proposed algorithm ESUS is implemented in Java, it takes as input all the matrices representing the network and component dependency graphs, together with the old and new component size arrays. The outputs of the algorithm are the sets $\{a_{i,j,m}\}$ and $\{x_{i,m}\}$ for each IoT device. The optimization model is written in OPL and running in the CPLEX IDE. Both the CPLEX studio and ESUS Java program are run on a desktop computer with 3 GHz 4-core processor with 8 Gb RAM.

5.2 Simulation settings

5.2.1 Network settings

The number of nodes $|V_G|$ is set from 10 to 30. For simplicity, we set the bandwidth of every connection between a device and the gateway by $b_g = 4$ Kilobytes per second (KB/s), and the bandwidth of each connection between two devices is set by $b_d = 8\ KB/s$. That is, $b_{i,j} = b_g$ if $i = 0\ or\ j = 0$, and otherwise $b_{i,j} = b_d$. These bandwidth values are suitable for real cases that devices have low processing and communication capacities, and are located far from each other. In terms of topology, we define three typical topologies of IoT networks, that are tree, partial mesh and full mesh. The purpose is to see the effectiveness of our algorithm in different kinds of network connectivity.

5.2.2 Optimization settings

The deadline T_{max} is set to 100 seconds, the number of iteration N in ESUS algorithm is set to 20 and the page size ρ is set to 4 KB, which is a common page size of flash memory (Feng, Feng, Yu, Tong & Liu, 2017). Our selected parameters are summarized in Table 5.1.

Table 5.1 Parameter settings in simulations

Parameter	ρ	b_g	b_d	s_m^{old}	T_{max}	N
Value	$4KB$	$4KB/s$	$8KB/s$	$8KB$	100 s	20

5.2.3 Software component sets

The number of software components $|V_D|$ is varied from 5 to 9. Given that the popular sizes of Contiki components range from $2KB$ to 40 KB (Ruckebusch *et al.*, 2016), in our simulation, we set every s_m^{old} to the same fixed size of 8 KB for each component set, and the corresponding s_m^{new} is set by a multiplication of s_m^{old} and a random number ranging from 1 to 4. A constraint graph D is also randomly created for each set of components. An example of component sizes is illustrated in Table 5.2, the nine components of this set are coupled in a constraint graph as shown in Fig. 5.1.

Table 5.2 An example of sizes of a 9-component set used in the simulation

Component	a	b	c	d	e	f	g	h	k
$\boldsymbol{s_m^{new}}$ **(kB)**	16	32	32	24	32	32	32	16	16
$\boldsymbol{s_m^{old}}$ **(kB)**	8	8	8	8	8	8	8	8	8

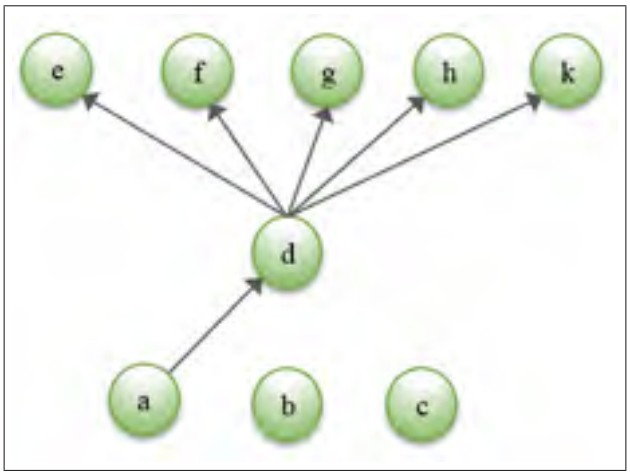


Figure 5.1 The component dependency graph of the 9-component set in Table. 5.2

5.3 Examination of tree topology

5.3.1 Scenario description

In the first scenario, we consider a typical topology of IoT networks in which device connections form a tree rooted at the gateway. The network includes ten nodes and is represented in Fig. 5.2, three nodes $1, 2$ and 3 are directly connected to the gateways and other nodes communicate with the gateway through these nodes. We perform our algorithm and the solver with different component sets as described above.

5.3.2 Results

Fig. 5.3 shows the results corresponding to the software component sets. The energy consumption is calculated by multiplying the total number of re-written pages with 92.57 μJ, which is the energy for re-writing one 4 *KB* flash page, according to (Park *et al.*, 2011). We can observe that results of ESUS are close to the minimal solutions given by CPLEX, with 12.8% difference on average and the closest is 4.1% different. We can also see that ESUS's results are better than the random schedules in most cases, it shows that our algorithm can reduce a

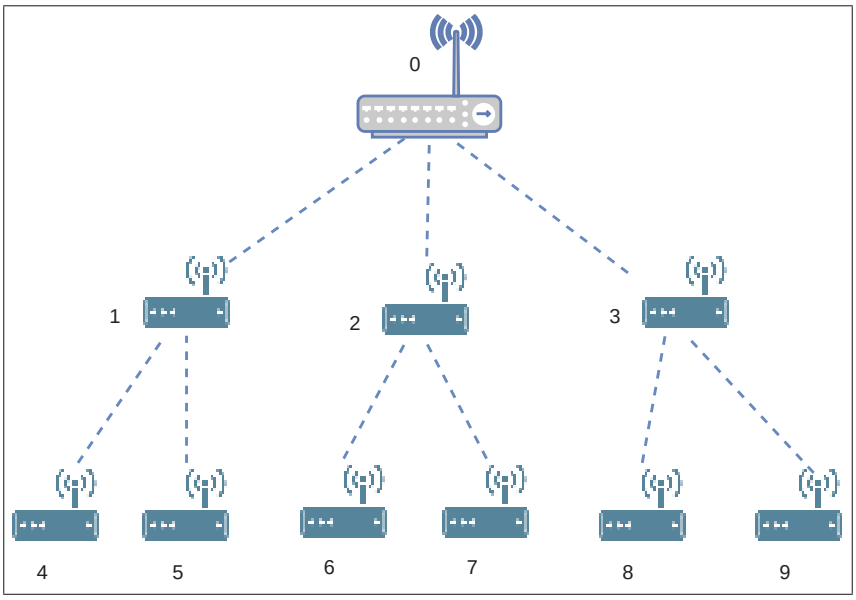


Figure 5.2 The tree topology in the first simulation scenario

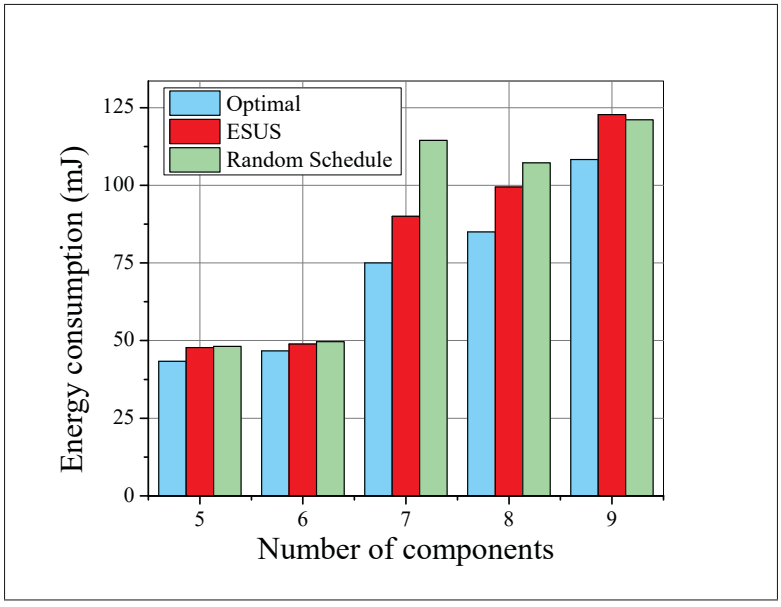


Figure 5.3 Energy consumption with different component sets in the tree topology

significant amount of energy consumed during the update process. Note that, the total energy of the network in the case using the 7-component set can be greater than the 8-component

one, because we have to take into account the significant effects of component sizes and the dependency graphs, which are both randomly generated.

5.4 Examination of partial mesh topology

5.4.1 Scenario description

In the second scenario, we define a partial mesh topology as shown in Fig. 5.4. In this network, IoT devices have more connections compared to the tree topology, device 2 can connect directly to 1 and 3, also 4 and 5, 6 and 7, 8 and 9 can connect to each other.

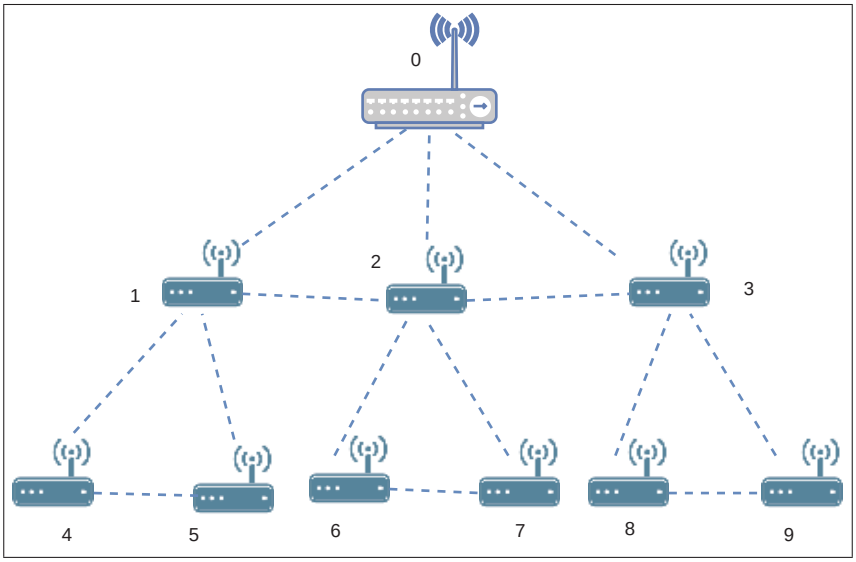


Figure 5.4 The partial mesh topology in the second simulation scenario

5.4.2 Results

The corresponding results of this network instance are represented in Fig. 5.5, with the same component sets as in the first scenario. The optimal results of CPLEX are the same as in the tree network. Besides, we can see that the results of ESUS are closer to optimal ones, compared to its results in the tree topology, with 7.0% difference on average. The reason for this is, when

nodes have more connections, ESUS can be easier to find efficient scheduling solutions. We can also observe that our algorithm are better than the random schedules in all cases.

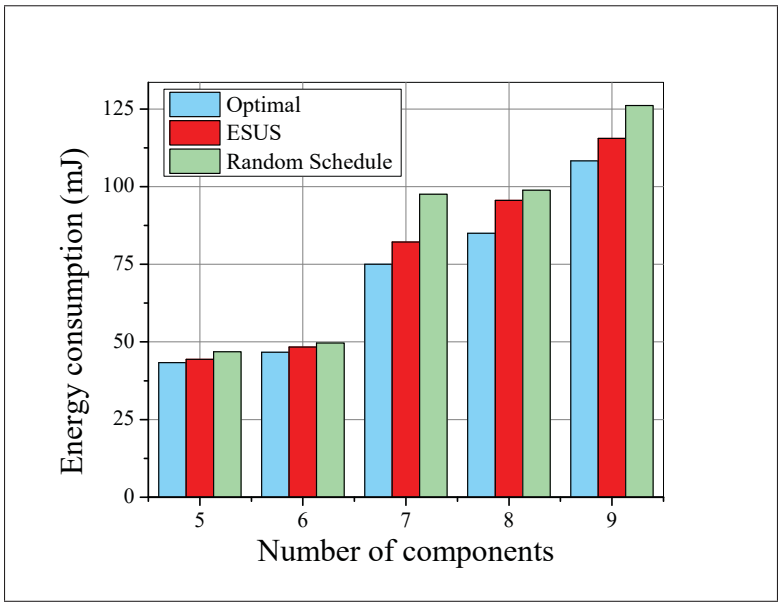


Figure 5.5 Energy consumption with different component sets in the partial mesh topology

5.5 Examination of full mesh topology

5.5.1 Scenario description

In the last scenario, we evaluate a mesh topology in which all devices can connect to each other as well as connect to the gateway, so the graph G presenting the network is a complete graph. Such topology can be common in smart homes and smart buildings applications.

5.5.2 Evaluation of different software component sets

Fig. 5.6 shows the results on a network instance of 10 nodes with the same software component sets as in the two first scenarios. We remark that the optimal results are still unchanged, and the results of ESUS are almost the same as in the second scenario, with 7.1% difference on

average and the closest is only 3.2% different. Fig. 5.6 also shows that ESUS outperforms the random schedules, with up to 30.8 % energy saved.

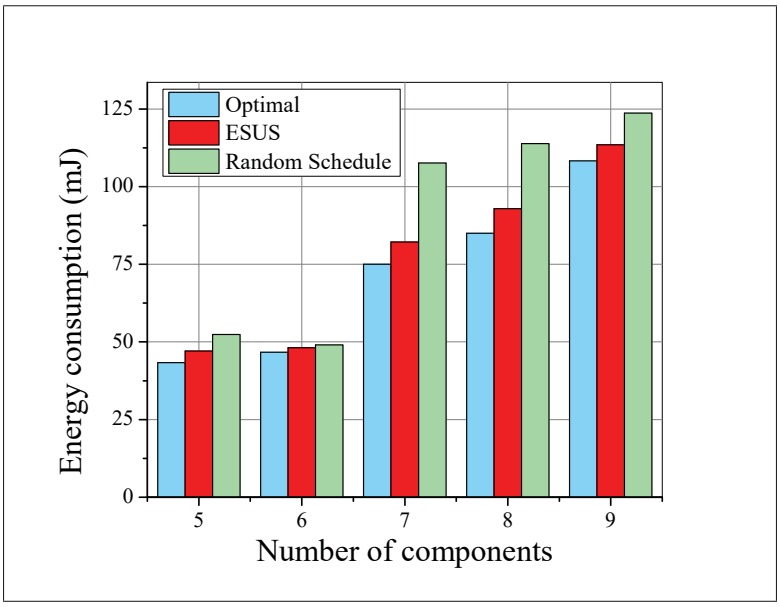


Figure 5.6 Energy consumption with different component sets in the full mesh topology with 10 nodes

5.5.3 Evaluation of different number of nodes

In another sub-scenario, we fix the component set is the one represented in Table 5.2 and Fig. 5.1, and examine the results with full mesh networks of different numbers of nodes. As shown in Fig. 5.7, ESUS can approximate the optimal solutions in all cases, and its results are better than random schedules in most cases. We also see that both ESUS solutions and the optimal ones are almost linearly related to the number of nodes, that is because of the full mesh topology in our simulation.

5.5.4 Effect of the deadline

In this sub-scenario, we fix the number of nodes in the full mesh topology to 10 and examine the results with different values of T_{max}. The component set used in this simulation is a set

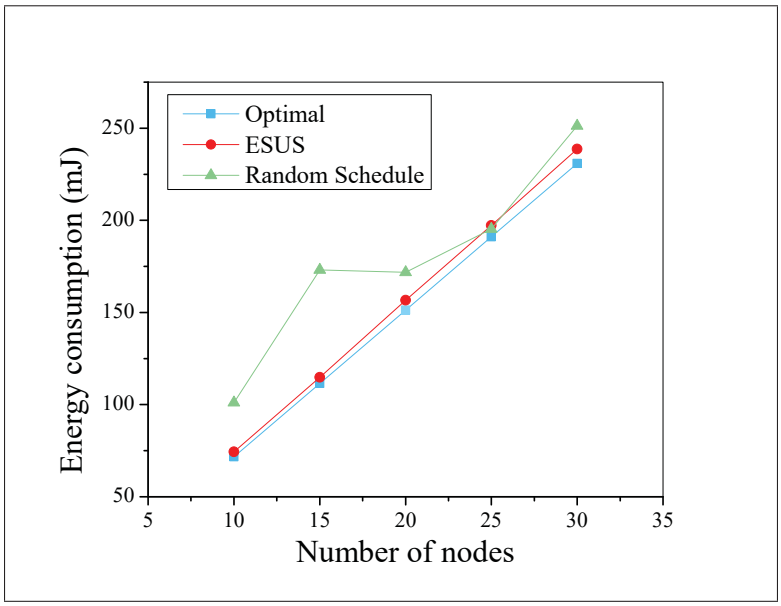


Figure 5.7 Energy consumption with different number of nodes in the full mesh topology

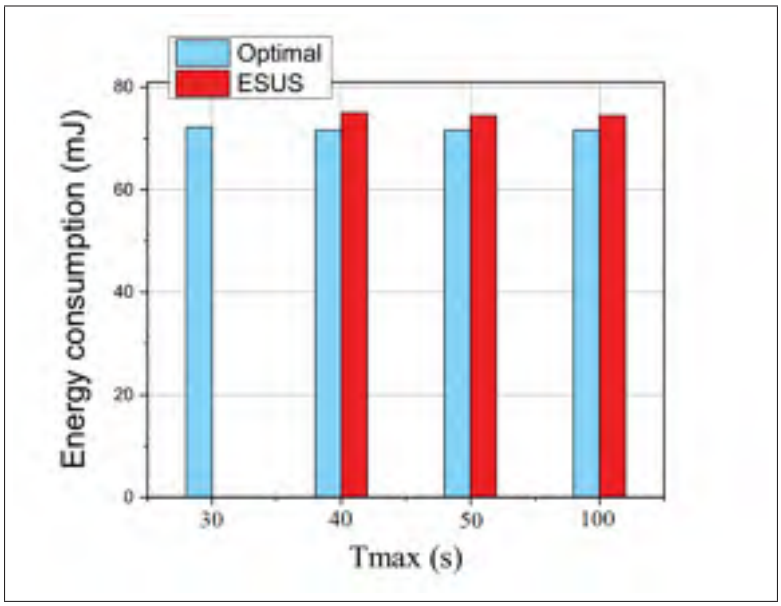


Figure 5.8 Energy consumption with different T_{max} in the 10-node full mesh topology

of 7 components. The results are illustrated in Fig. 5.8, the missing value indicates that the solution is infeasible. The optimal solution remains unchanged (71.65 *mJ*) with the values 40,

50 and 100 of T_{max}, and it increase slightly to 72.20 mJ when T_{max} downs to 30. As for ESUS, its result is the same for $T_{max} = 50$ and 100, it also increase a little when $T_{max} = 40$. However, with $T_{max} = 30$, ESUS could not find a feasible solution.

5.5.5 Running time evaluation

We also evaluate our algorithm in terms of performance, the running time of ESUS and CPLEX are compared. We perform on three different sets from 7 to 9 components with full mesh networks of different numbers of nodes and $T_{max} = 100s$. The results are shown in Table 5.3, let T_{ESUS} be the average time taken by ESUS to find optimal solutions, and T_{CPLEX} be the average elapsed time by CPLEX solver. We can observe that ESUS runs much faster than CPLEX, especially when the number of nodes increases. Note that, in case of finding random schedules, the running time of CPLEX is significantly reduced because it only has to search for a feasible solution.

Table 5.3 Comparison between average running time of ESUS algorithm and CPLEX

Number of Nodes	10	15	20	25	30
T_{CPLEX} (s)	9.43	94.15	642.72	787.84	8734.43
T_{ESUS} (s)	0.044	0.053	0.084	0.099	0.142

5.6 Discussion

In this chapter, we have presented the simulation results of our update scheduling method in different network instances. Various component sets and different numbers of nodes were taken under investigation. The results of ESUS were compared to the optimal solutions and the random schedules given by CPLEX solver. We also evaluated our algorithm in terms of running time, to show the efficiency of it compared to the solver, which often takes a lot of time to generate the optimal solution.

In summary, our proposed algorithm could reduce a notable number of pages needed to be re-written during the update process, compared to random schedules, that helps to save a significant amount of energy consumption. The results of ESUS were close to the minimal energy consumed in optimal schedules given by CPLEX solver. Note that in all cases, the deadline T_{max} was always satisfied by the schedules given by ESUS. Moreover, our algorithm performs much faster than the solver to give the solution. This advantage can be exploited to quickly adapt new requirements of IoT applications.

The simulation results show the efficiency of our update method to minimize the energy consumption of updating component-based IoT device networks. They also show the advantage of P2P update mechanism with centralized control by a gateway.

CONCLUSION AND RECOMMENDATIONS

In the technological revolution represented by the Internet of Things (IoT), a massive number of IoT devices can interconnect and provide various intelligent services and applications. With the IoT boom, software update is one of the most important tasks of IoT systems in order to adapt incremental user requirements and to maintain effective operations. The scale of IoT brings two main challenges to software update management in IoT device networks. The first is, how to delivery updates to IoT devices in an energy efficient way. And second, how to avoid long system downtime during the update process.

In this thesis, we addressed the problem of energy efficient software update scheduling in IoT device networks, focusing on the kind of devices that employ the component-based software system. In order to solve the problem, we introduced a novel energy model of the update process inside a single device, taking into account the component update order. After that, we formulated the scheduling problem as an optimization problem in the form of integer non-linear programming (INLP), with the objective function to minimize the total energy consumption of the update process in the entire network.

Due to the high complexity of the problem, we then proposed the ESUS algorithm to find a near-optimal schedule for updating all devices in the network. Our algorithm divides the schedule into steps and tries to assign the downloads of each device in the best way. To evaluate our method, we employed the CPLEX solver to obtain optimal schedules and random ones, then compared these results to the outputs of ESUS. Different network instances and various software component sets were examined. Through simulation results, we showed that our algorithm can effectively approximate the optimal solution given by CPLEX solver with much lower running time.

Our main contributions in this research project are as follows:

- A novel energy model of the update process of component-based IoT devices, considering the component update order.
- A mathematical model of the optimization problem for scheduling updates over an IoT network, that minimizes the total energy consumption of devices during the update.
- An algorithm to approximate the optimal schedule for updating all devices in the network.

In the future, we will extend our work by considering different application demands, so that devices have different software component sets which make more complexity for the update management. Also, other kinds of software execution environments such as virtual machines or image-based will be taken into account. In addition, we are interested in extending the research problem by considering multiple gateways in the network. In this context, the network can have different sources of updates and an efficient collaboration scheme between gateways needs to be proposed.

APPENDIX I

ARTICLES PUBLISHED IN CONFERENCES

This thesis is related to two papers published in conferences:

- "Energy Efficient Scheduling for Networked IoT Device Software Update". CNSM 2019 (short paper), Halifax, Nova Scotia, Canada. Published in October 2019.
- "Energy Efficient Software Update Mechanism for Networked IoT Devices". IEEE Globecom 2019, Waikoloa, Hawaii, USA. Published in December 2019.

REFERENCES

Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M. & Ayyash, M. (2015). Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE communications surveys & tutorials*, 17(4), 2347–2376.

Amjad, M., Sharif, M., Afzal, M. K. & Kim, S. W. (2016). TinyOS-new trends, comparative views, and supported sensing applications: A review. *IEEE Sensors Journal*, 16(9), 2865–2889.

Arduino Mega. (2019). Arduino Mega 2560. Consulted at https://www.arduino.cc/en/main/products.

Arduino.cc. (2019). Arduino. Consulted at https://www.arduino.cc.

AVR ATmega2560. (2019). ATmega2560. Consulted at https://www.microchip.com/wwwproducts/en/ATmega2560.

Barcelo, M., Correa, A., Llorca, J., Tulino, A. M., Vicario, J. L. & Morell, A. (2016). IoT-cloud service optimization in next generation smart environments. *IEEE Journal on Selected Areas in Communications*, 34(12), 4077–4090.

Boulis, A., Han, C.-C., Shea, R. & Srivastava, M. B. (2007). SensorWare: Programming sensor networks beyond code update and querying. *Pervasive and mobile computing*, 3(4), 386–412.

Brown, S. & Sreenan, C. (2013). Software updating in wireless sensor networks: A survey and lacunae. *Journal of Sensor and Actuator Networks*, 2(4), 717–760.

Cao, Q., Abdelzaher, T., Stankovic, J. & He, T. (2008). The liteos operating system: Towards unix-like abstractions for wireless sensor networks. *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*, pp. 233–244.

CPLEX. (2019). CPLEX Optimizer. Consulted at https://www.ibm.com/analytics/cplex-optimizer.

Dong, C. & Yu, F. (2015). An efficient network reprogramming protocol for wireless sensor networks. *Computer Communications*, 55, 41–50.

Dong, W., Chen, C., Bu, J. & Huang, C. (2013a). Enabling efficient reprogramming through reduction of executable modules in networked embedded systems. *Ad Hoc Networks*, 11(1), 473–489.

Dong, W. et al. (2013b). R3: Optimizing relocatable code for efficient reprogramming in networked embedded systems. *2013 Proceedings IEEE INFOCOM*, pp. 315–319.

Dong, W. et al. (2015). Optimizing relocatable code for efficient software update in networked embedded systems. *ACM Transactions on Sensor Networks (TOSN)*, 11(2), 22.

Feng, Y., Feng, D., Yu, C., Tong, W. & Liu, J. (2017). Mapping granularity adaptive ftl based on flash page re-programming. *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 374–379.

Fok, C.-L., Roman, G.-C. & Lu, C. (2009). Agilla: A mobile agent middleware for self-adaptive wireless sensor networks. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(3), 16.

Hahm, O., Baccelli, E., Petersen, H. & Tsiftes, N. (2016). Operating systems for low-end devices in the internet of things: a survey. *IEEE Internet of Things Journal*, 3(5), 720–734.

Healy, M., Newe, T. & Lewis, E. (2008). Wireless sensor node hardware: A review. *SENSORS, 2008 IEEE*, pp. 621–624.

Heo, J., Gu, B., Eo, S. I., Kim, P. & Jeon, G. (2010). Energy efficient program updating for sensor nodes with flash memory. *Proceedings of the 2010 ACM symposium on applied computing*, pp. 194–200.

Hu, J., Xue, C. J., He, Y. & Sha, E. H.-M. (2009). Reprogramming with minimal transferred data on wireless sensor network. *2009 IEEE 6th International Conference on Mobile Adhoc and Sensor Systems*, pp. 160–167.

Hui, J. W. & Culler, D. (2004). The dynamic behavior of a data dissemination protocol for network programming at scale. *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pp. 81–94.

Karray, F., Jmal, M. W., Garcia-Ortiz, A., Abid, M. & Obeid, A. M. (2018). A comprehensive survey on wireless sensor node hardware platforms. *Computer Networks*, 144, 89–110.

Khan, I., Belqasmi, F., Glitho, R., Crespi, N., Morrow, M. & Polakos, P. (2015). Wireless sensor network virtualization: A survey. *IEEE Communications Surveys & Tutorials*, 18(1), 553–576.

Kolomvatsos, K. (2018). An intelligent, uncertainty driven management scheme for software updates in pervasive IoT applications. *Future generation computer systems*, 83, 116–131.

Koshy, J. & Pandey, R. (2005a). VM Star: Synthesizing Scalable Runtime Environments for Sensor Networks. *Proceedings of the International Conference on Embedded Networked Sensor Systems (SenSys)*, pp. 243–254.

Koshy, J. & Pandey, R. (2005b). Remote incremental linking for energy-efficient reprogramming of sensor networks. *Proceeedings of the Second European Workshop on Wireless Sensor Networks, 2005.*, pp. 354–365.

Kovatsch, M., Lanter, M. & Duquennoy, S. (2012). Actinium: A restful runtime container for scriptable internet of things applications. *2012 3rd IEEE International Conference on the Internet of Things*, pp. 135–142.

Leukert, B., Kubach, T., Eckert, C., Tsutsumi, K., Crawford, M. & Vayssiere, N. (2016). IoT 2020: Smart and secure IoT platform. *IEC White papers*, 1–181.

Levis, P. & Culler, D. (2002). Maté: A tiny virtual machine for sensor networks. *ACM Sigplan Notices*, 37(10), 85–95.

Levis, P., Patel, N., Culler, D. & Shenker, S. (2004). Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. *Proc. of the 1st USENIX/ACM Symp. on Networked Systems Design and Implementation*, 25.

McGrath, M. J. & Scanaill, C. N. (2013). Key Sensor Technology Components: Hardware and Software Overview. In *Sensor Technologies* (pp. 51–77). Springer.

Munawar, W., Alizai, M. H., Landsiedel, O. & Wehrle, K. (2010). Dynamic tinyos: Modular and transparent incremental code-updates for sensor networks. *2010 IEEE International Conference on Communications*, pp. 1–6.

Norris, D. (2016). *Python for Microcontrollers: Getting Started with MicroPython*. Mcgraw-hill Education-Europe.

Panta, R. K. & Bagchi, S. (2012). Mitigating the Effects of Software Component Shifts for Incremental Reprogramming of Wireless Sensor Networks. *IEEE Transactions on Parallel and Distributed Systems*, 23(10), 1882–1894.

Panta, R. K., Vintila, M. & Bagchi, S. (2010). Fixed cost maintenance for information dissemination in wireless sensor networks. *2010 29th IEEE Symposium on Reliable Distributed Systems*, pp. 54–63.

Panta, R. K., Bagchi, S. & Midkiff, S. P. (2011). Efficient incremental code update for sensor networks. *ACM Transactions on Sensor Networks (TOSN)*, 7(4), 30.

Park, S., Kim, Y., Urgaonkar, B., Lee, J. & Seo, E. (2011). A comprehensive study of energy efficiency and performance of flash-based SSD. *Journal of Systems Architecture*, 57(4), 354–365.

Razzaque, M. A., Milojevic-Jevric, M., Palade, A. & Clarke, S. (2015). Middleware for internet of things: a survey. *IEEE Internet of things journal*, 3(1), 70–95.

Reijers, N. & Langendoen, K. (2003). Efficient code distribution in wireless sensor networks. *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pp. 60–67.

Ruckebusch, P., De Poorter, E., Fortuna, C. & Moerman, I. (2016). Gitar: Generic extension for internet-of-things architectures enabling dynamic updates of network and application modules. *Ad Hoc Networks*, 36, 127–151.

Saginbekov, S. & Jhumka, A. (2014). Efficient code dissemination in wireless sensor networks. *Future Generation Computer Systems*, 39, 111–119.

Taherkordi, A., Loiret, F., Rouvoy, R. & Eliassen, F. (2013). Optimizing sensor network reprogramming via in situ reconfigurable components. *ACM Transactions on Sensor Networks (TOSN)*, 9(2), 14.