Energy Efficient Software Update Mechanism for Networked IoT Devices

Ngoc Hai Bui, Kim Khoa Nguyen, Chuan Pham, Mohamed Cheriet
Synchromedia - École de Technologie Supérieure, University of Quebec, Canada
Email: {ngoc-hai.bui.1, chuan.pham.1}@ens.etsmtl.ca, {kim-khoa.nguyen, mohamed.cheriet}@etsmtl.ca

***Abstract*—Due to security issues and incremental user requirements, software in IoT devices needs to be changed frequently. Recently, advanced IoT devices employ the component-based software architecture 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 the entire network. A key operation which consumes a significant amount of energy in the update process is flash re-writing, in which the order of re-writing components into the memory is decisive for energy consumption. In this paper, we propose a mechanism that schedules updates on all devices in an IoT network to minimize the energy consumption, taking into account the deadline constraint for updating the entire network. We introduce a novel energy model of the update process, then propose an algorithm to approximate the optimal schedule for updating all devices in the network. Simulation results show that our algorithm can obtain a near optimal which is, on average, 7.1% different from the global minimum.**

***Index Terms*—energy efficiency, software update, IoT device, component-based IoT software.**

I. Introduction

In the technological revolution represented by the Internet of Things (IoT), a tremendous number of IoT devices can interconnect and provide various intelligent services and applications [1]. In order to adapt requirements of IoT applications, software in IoT devices needs to be changed frequently to improve existing functionalities or to fix revealed bugs. To maintain effective operations, software update must become an integral part of IoT systems.

Research on software update for wireless sensor/IoT networks can be classified into three main categories: Data dissemination, data minimization, and execution environment [2]. Data dissemination protocols [3] focus on the ways to deliver software updates in the network, to minimize communication costs. On the other hand, data minimization [4] focuses on reducing the size of updates, and has a direct impact on the energy used for communication and processing. Therefore, it not only helps extend sensor network lifetime, but also decreases updating time. In addition, the execution environment such as virtual machine [5], image-based and component-based [6], also has a significant impact on how the software in an IoT device can be updated. Recently, the common execution environment in advanced IoT devices is component-based, such as Contiki, SOS [7], in which software is partitioned into small blocks, so called components, which can be added or updated at run-time. In such environment, only parts of the entire software need to be changed during the update process, allowing to reduce the amount of data need to transfer. For example, in Fig. 1, 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; 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 sending 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 [8], they can be downloaded an stored in a buffer such as EEPROM, then the Contiki core will link them to existing components and load into the flash memory in run-time.

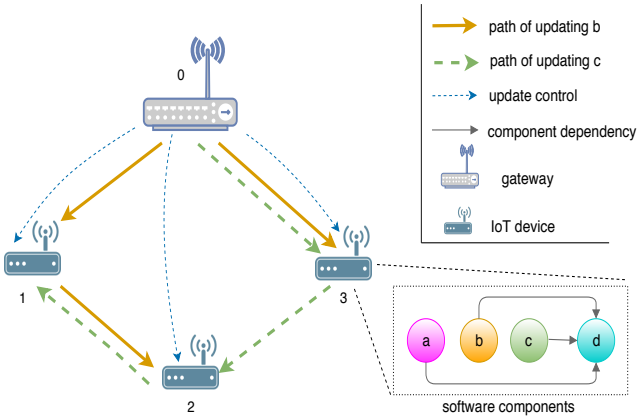
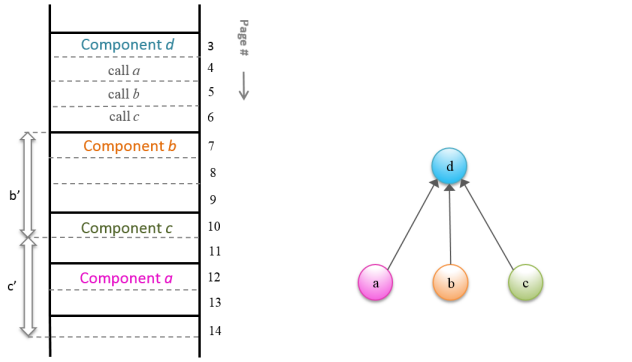


Fig. 1: An IoT network with a gateway is responsible to update IoT devices.

Prior research on component-based software for IoT devices often focuses on the ways a component is replaced [8], [9], [10], and do not consider thoroughly how updates are distributed, especially when multiple updates are required at the same time. In this paper, we consider an application consisting of a number of component-based IoT devices with the same software connected to a gateway, and a set of components needs to be updated to all devices. In this context, a peer-to-peer approach can help reduce the time to deploy the update, since a device can download updated components from multiple neighbor devices at the same time through near-field communications (e.g., Bluetooth or WiFi) without



(a) Components in flash memory of an IoT device, b and c are updated by b' and c'.

(b) Software component constraint graph.

Fig. 2: Software components in flash memory of an IoT device and the corresponding component constraint graph.

having to rely on a more expensive communication with the gateway. Also, caching the update in the gateway [2] is a good solution because devices do not need to get all the new components from the Internet. In the example presented in Fig. 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.

Inside an IoT device, software components are written in a sequence in flash memory as shown in Fig. 2a, from low addresses to high addresses. Each component may occupy several memory pages. When a component is updated (assume its size increases), 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 higher addresses [11]. Hence, all these components also need to be re-written. In that context, different orders could result in different numbers of re-written blocks. For instance, two components b and c in Fig. 2a, account for 3 and 2 pages in the flash, respectively. Suppose that we update both b and c with 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 the component after c that is a, then 4 pages of b', 4 pages of c' and a again, so the total number of re-written pages are *12* plus twice the size of a. On the other hand, 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 [8]. 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. Since a big amount of energy is consumed for flash re-writing [12], determining an optimal update order is substantial for reducing energy consumption in device software update operation. Some work [13] also mentioned the update order constraint, however, this constraint and its impact on energy has not been considered thoroughly in previous studies.

In our work, we 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. A schedule is a plan specifies two decisions: First, where a component can be downloaded for each device; second, when it can be downloaded. Our main contributions are as follows:

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

The rest of this paper is organized as follows. In section II, we present the system description. Section III mathematically formulates the problem. Our proposed algorithm is introduced in section IV. Section V shows our simulation results followed by our conclusion.

II. System description

A. IoT sofware components

We consider the case in which a gateway downloads software updates from a server running on the cloud, and then send to a number of devices of the same type (i.e., having the same hardware and software configuration). Thanks to the component-based architecture, a device does not need to update all new components at a time, but one by one. During 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.

The software component constraint 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 constraints. The graph D can be represented by a matrix $\mathcal{M}_\mathcal{D} = \{c_{m,n}\}$ where each entry 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. 2b. In this example, the device can update component d only after it finishes updating components a, b and c.

Each component occupies a number of memory pages, which is the smallest unit that can be erased and re-written. The modification of any byte in a page will result in the entire page needs to be re-written. Consider a set of components a, b, c and d that lie in the flash as shown in Fig. 2a. We assume the size of c grows after the update (c' is bigger than

c); then all the components lie after c in the memory will have to be shifted to higher addresses. Thus, even if a is not in the update list, it will be moved to 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. 2a has to be re-written. We assume the energy consumption in a flash re-writing operation is proportional to the number of pages modified in this operation.

B. System model

We consider a model of an IoT network including a number of connected IoT devices and a gateway. The gateway manages IoT devices and is responsible for scheduling updates for the devices. Both the gateway and devices are considered as "nodes" in a graph $G =< V_G, E_G >$, with V_G is the set of vertices and E_G is the set of edges representing nodes and links, respectively. Let $V_G = \{i|\ i = 0, 1, \dots |V_G|\}$, in which $i = 0$ represents the gateway, and IoT devices are corresponding to $i > 0$. G can be represented by a symmetric matrix $\mathcal{M}_\mathcal{G} = \{e_{i,j}\}$ where each entry $e_{i,j} = 1$ presents the link between two nodes i and j. For simplicity, we suppose that every connection between a device and the gateway has the same bandwidth b_g, and bandwidth of every connection between two devices also has the same value b_d.

A device receives components from both the gateway and other devices. 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. Since the total update time is also important, it is necessary to limit the amount of time to perform update in the entire network by a deadline T_{max}. We can also assume that the installation time of devices is much smaller than the download time and can be skipped.

Let $a_{i,j,m}$ denotes the assignment of equals to 1 if device i downloads component m from gateway/device j, and $x_{i,m}$ denotes the start time device i downloads component m. The update schedule of each device i is characterized by the sets $\{a_{i,j,m}\}$ and $\{x_{i,m}\}$. In our problem, we need to find an optimal schedule for the entire network, which minimizes the total energy consumed during update process while satisfies the four constraints: (i) the dependency order constraint of components, (ii) the constraint that a device can only send a component after having it, (iii) the constraint that a device can download at most one component from one source at a time, and (iv) the deadline constraint of updating the entire network.

We remark that the computation of our scheduling is centralized, i.e., the schedule is computed by a centralized controller running in the gateway [14], as presented in Fig. 1. 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 this device can download which component from which source. The process finishes when every node has all the new components.

III. Problem formulation

In this section, we present the energy model for the update process in an IoT device and the mathematical formulation of our energy efficient software update scheduling problem.

A. Energy consumption model

As mentioned in *Introduction*, 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 amount of time device i completely downloads component m is represented by $t_{i,m}$ and can be calculated as:

$$t_{i,m} = \begin{cases} 0, & i = 0, \\ a_{i,0,m} \times \frac{s_m^{new}}{b_g} + (1 - a_{i,0,m}) \times \frac{s_m^{new}}{b_d}, & i > 0. \end{cases} \quad (1)$$

In (1), $t_{i,m}$ is 0 if device i is the gateway, otherwise $t_{i,m}$ is calculated by dividing the size of m to the corresponding bandwidth. The amount of energy consumed when a device i updates a component m can be calculated as:

$$E_{i,m} = e\times \left(\frac{s_m^{new}}{\rho} + \lambda_m \left(\sum_{h \in \alpha(m)} \frac{size(h)}{\rho} + \sum_{h \in \alpha(m)} \sum_{k \in \beta(m)} c_{h,k} \right) \right), \quad (2)$$

where e is the energy consumption for writing one page, ρ is the size of one page, λ_m is a variable 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 variable $c_{h,k}$ is corresponding to an entry (h, k) in matrix $\mathcal{M}_\mathcal{D}$ that equals to 1 if 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 updating m, i.e., $size(h)$ is s_h^{new} if h is updated before m, otherwise $size(h)$ is s_h^{old}. $size(h)$ can be computed as:

$$size(h) = s_h^{new}\delta_{h,m} + s_h^{old}(1 - \delta_{h,m}), \quad (3)$$

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

$$\delta_{h,m} = \begin{cases} 1 & x_{i,h} + t_{i,h} < x_{i,m} + t_{i,m}, \\ 0 & x_{i,h} + t_{i,h} \geq x_{i,m} + t_{i,m}. \end{cases} \quad (4)$$

Given a device with a fixed number of components, we can easily see that the quantity $\sum_{h \in \alpha(m)} \sum_{k \in \beta(m)} c_{h,k}$ in equation (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 increase, means that $\lambda_m = 1,\ \forall m \in V_D$, then we can have the simplified form of $E_{i,m}$ as:

$$\bar{E}_{i,m} = \frac{e}{\rho}\left(s_m^{new} + \sum_{h\in\alpha(m)}\left(s_h^{new}\delta_{h,m} + s_h^{old}(1-\delta_{h,m})\right)\right). \quad (5)$$

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:

$$E_i = \sum_{m\in V_D} \bar{E}_{i,m}. \quad (6)$$

E_i can be considered as a function of $\{a_{i,j,m}\}$ and $\{x_{i,m}\}$.

B. Optimization model

For the convenience of discussion, the notations are summarized in table I.

TABLE I: Notations

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
$e_{i,j}$	A variable indicates the link between two devices i and j
$c_{m,n}$	A variable indicates that component n calls component m
Decision variables	
$a_{i,j,m}$	A variable equals to 1 if device i downloads component m from device/gateway j
$x_{i,m}$	Start time device i downloads component m

The optimization model for our scheduling problem is formulated as:

$$\min \sum_{i=1}^{|V_G|} E_i. \quad (7)$$

Subject to:

$$a_{i,j,m} \in \{0,1\}, \quad \forall i,j \in V_G, m \in V_D. \quad (8)$$

$$x_{i,m} \geq 0, \quad \forall i \in V_G, i > 0, m \in V_D. \quad (9)$$

$$x_{0,m} = 0, \quad \forall m \in V_D. \quad (10)$$

$$\sum_{j\in V_G} a_{i,j,m} = 1, \quad \forall i \in V_G, i > 0, m \in V_D. \quad (11)$$

$$a_{i,j,m} \leq e_{i,j}, \quad \forall i,j \in V_G, i > 0, m \in V_D. \quad (12)$$

$$a_{i,j,m}(x_{i,m} - x_{j,m} - t_{j,m}) \geq 0, \quad \forall i,j \in V_G, i > 0, m \in V_D. \quad (13)$$

$$a_{i,j,m}a_{i,j,n}(x_{i,m} - x_{i,n} - t_{i,n})(x_{i,n} - x_{i,m} - t_{i,m}) \leq 0, \quad \forall i,j \in V_G, m \neq n \in V_D. \quad (14)$$

$$c_{m,n}(x_{i,n} - x_{i,m} - t_{i,m}) \geq 0, \quad \forall i \in V_G, m,n \in V_D. \quad (15)$$

$$x_{i,m} + t_{i,m} \leq T_{max}, \quad \forall i \in V_G, m \in V_D. \quad (16)$$

In our model, constraint (8) is the value constraint, each variable $a_{i,j,m}$ can only be 1 or 0 to indicate if device j download component m from device i or not. Constraint (9) indicates that each start time needs to be greater or equal to 0. Condition (10) means that the gateway does not download from any source. The constraint that a device only downloads a component m from one other node is indicated in condition (11). Constraint (12) is the network topology constraint, means that a device i can download from device/gateway j only if there is a link (i,j). Constraint (13) is a device j can only send a component to a device i after it finishes downloading this component, with $t_{j,m}$ is calculated by formula (1). Constraint (14) shows that a device can only download one component from each other node at a time, in other words, in one link there are at most one component is transferred in one moment. Constraint (15) indicates the download order of each gateway needs to satisfy the component constraint graph. And finally, condition (16) is the deadline constraint T_{max}.

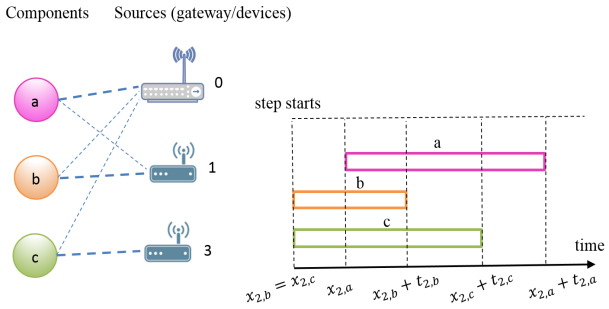


Fig. 3: A bipartite graph presents downloadable components of device 2 and corresponding sources at a step, with a matching and corresponding download time. The thicker edges in the graph present the matching.

IV. ALGORITHMIC SOLUTION

We propose an algorithm called ESUS to solve the problem of energy efficient update scheduling for IoT networks. Our proposed algorithm employs procedure P_1 to generate an energy efficient schedule without considering the deadline constraint T_{max}. The outline of P_1 is described in Algorithm 1, this procedure is based on the idea of dividing the schedule into steps. At each step, each device i maintains a list of downloadable components, and a list of possible sources (other devices or the gateway), that can be represented as a bipartite graph B_i. P_1 finds 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 have smallest number of sources, then randomly assigns a source to this component and updates source lists of other components. 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*. An example of a bipartite graph is shown in Fig. 3, at this step, device 2 has three downloadable components a, b, and c that lie in its flash as in Fig. 2a. With this graph, it can get all the components by downloading a from 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.

Algorithm 1: P_1 - Generate a schedule

```
repeat
    Each step, do
    for each device i do
        Construct the bipartite graph B_i;
        Do Matching the bipartie graph 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}} have
          the order as in the flash;
    end
    Calculate the finishing time of this step;
until all nodes complete downloading all components;
```

Algorithm 2: ESUS Algorithm

```
for t from 1 to N do
    Generate schedule S_t by P_1;
    if S_t does not satisfy Tmax then
        Adjust S_t by P_2;
    end
    if S_t still does not satisfy Tmax then
        Start new iteration t + 1;
    end
    else
        if S_t is better than current best solution then
            Update the best solution is S_t;
        end
    end
end
```

In case T_{max} is not satisfied by the initial schedule given by P_1, ESUS employs the procedure P_2 to properly adjust the schedule to reduce the update time. P_2 analyzes and shifts the download time to the earliest as possible. It sequentially performs on each component m, P_2 checks the paths of distributing m in the network. For each device i downloads m, it checks if the start downloading time $x_{i,m}$ can be shifted to an earlier one. That is, if the source of i has m sooner than $x_{i,m}$, and if i has all the necessary components called by m before $x_{i,m}$, then P_2 changes $x_{i,m}$ to the earliest as possible. P_2 iterates the components in a topological order, it means that when examining a component m, all the components that m depends on are already adjusted. Due to the randomness of P_1, ESUS runs the two procedures in a number of iterations N and chooses the best solution. The outline of our main algorithm - ESUS is presented in Algorithm 2.

V. Simulation results

A. Settings

In our simulation, we vary the number of nodes $|V_G|$ from 10 to 30, number of component $|V_D|$ from 5 to 9. For each component set, each s_m^{old} is set to the same fixed size, and the corresponding s_m^{new} is randomly created so that s_m^{new} is a multiple of s_m^{old}. An example of $\{s_m^{new}\}$ and $\{s_m^{old}\}$ of a set including 9 components is shown in Table II. A corresponding constraint graph is also randomly created for each set of components. Without loss of generality, we define a mesh topology in which all devices can connect to each other as well as connect to the gateway, so G is a complete graph, such topology can be common in smart home and smart building applications. The deadline T_{max} is chosen to ensure that feasible schedules exist, to do that, we use CPLEX solver to find the minimal time T_{min} for updating the network, then choose $T_{max} > T_{min}$. Our selected parameters are summarized in Table III.

TABLE II: Sizes of a 9 component set used in the simulation.

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

TABLE III: Parameter settings for simulation.

Parameter	ρ	b_g	b_d	s_m^{old}	T_{max}	N
Value	$4KB$	$4KB/s$	$8KB/s$	$8KB$	50 s	20

We use the CPLEX solver to find optimal schedules for our optimization problem. Besides that, we also employ CPLEX to find a random feasible schedule for each network instance, that is a schedule satisfied all the constraints but does not minimize the energy objective function. We calculate the energy consumption of those random schedules and compare to results of ESUS algorithm and optimal solutions given by CPLEX solver, in order to evaluate our algorithm.

B. Results

Fig. 4 shows the results corresponding to a network instance of 10 nodes with different software component sets. We can see that results of ESUS are close to the minimal solutions given by CPLEX, with 7.1% difference on average and the closest is only 3.2% different. Fig. 4 also shows that ESUS outperforms the random schedules, with up to 30.8 % re-written pages saved. In another scenario, we fix the component set is the one in Table II, and examine the results with different number of nodes. As shown in Fig. 5, 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

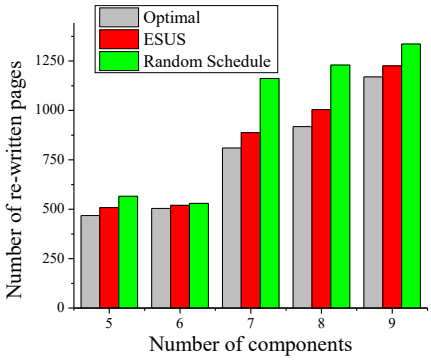


Fig. 4: Comparison of number of re-written pages with different component sets.

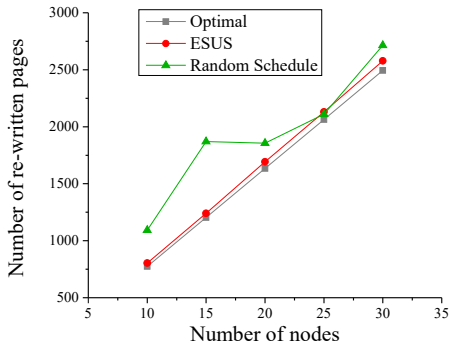


Fig. 5: Comparison of number of re-written pages with different number of nodes.

TABLE IV: 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

solutions and the optimal ones are almost linearly related to the number of nodes, that is because of the mesh network topology in our simulation.

In terms of performance, we compare the running time of ESUS with CPLEX. ESUS is implemented in Java, both CPLEX and ESUS are run on a desktop computer with 3 GHz 4-core processor with 8 Gb RAM. We perform on three different sets from 7 to 9 components with different number of IoT nodes and $T_{max} = 50s$. The running time are shown in Table IV, let T_{ESUS} be the average time taken by ESUS, 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.

VI. Conclusion

In this paper, we have presented the problem of energy efficient software update scheduling in component-based IoT device networks. We formulated the problem as an optimization problem with a novel energy model for the update process. We then proposed ESUS algorithm to find a near-optimal schedule for updating all devices in the network. Through simulation results, we showed that our algorithm can effectively approximate the optimal solution given by CPLEX solver with much lower running time.

In the future, we will extend our work by considering specific network topologies and different application demands, other kinds software execution environment such as virtual machine or image based will also be taken into account.

References

[1] A. Al-Fuqaha *et al.*, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Commun. surveys & Tuts*, vol. 17, no. 4, pp. 2347–2376, 2015.

[2] S. Brown and C. Sreenan, "Software updating in wireless sensor networks: A survey and lacunae," *Journal of Sensor and Actuator Networks*, vol. 2, no. 4, pp. 717–760, 2013.

[3] C. Dong and F. Yu, "An efficient network reprogramming protocol for wireless sensor networks," *Computer Communications*, vol. 55, pp. 41–50, 2015.

[4] W. Dong *et al.*, "R3: Optimizing relocatable code for efficient reprogramming in networked embedded systems," in *2013 Proceedings IEEE INFOCOM*, pp. 315–319.

[5] M. Kovatsch *et al.*, "Actinium: A restful runtime container for scriptable internet of things applications," in *2012 3rd IEEE iThings*, pp. 135–142.

[6] A. Taherkordi *et al.*, "Optimizing sensor network reprogramming via in situ reconfigurable components," *ACM TOSN*, vol. 9, no. 2, p. 14, 2013.

[7] O. Hahm *et al.*, "Operating systems for low-end devices in the internet of things: a survey," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 720–734, 2016.

[8] P. Ruckebusch *et al.*, "Gitar: Generic extension for internet-of-things architectures enabling dynamic updates of network and application modules," *Ad Hoc Networks*, vol. 36, pp. 127–151, 2016.

[9] W. Munawar *et al.*, "Dynamic tinyos: Modular and transparent incremental code-updates for sensor networks," in *2010 IEEE ICC*, pp. 1–6.

[10] M. Amjad *et al.*, "Tinyos-new trends, comparative views, and supported sensing applications: A review," *IEEE Sensors Journal*, vol. 16, no. 9, pp. 2865–2889, 2016.

[11] W. Dong *et al.*, "Optimizing relocatable code for efficient software update in networked embedded systems," *ACM TOSN*, vol. 11, no. 2, p. 22, 2015.

[12] R. K. Panta *et al.*, "Efficient incremental code update for sensor networks," *ACM TOSN*, vol. 7, no. 4, p. 30, 2011.

[13] W. Dong *et al.*, "Enabling efficient reprogramming through reduction of executable modules in networked embedded systems," *Ad Hoc Networks*, vol. 11, no. 1, pp. 473–489, 2013.

[14] M. Barcelo *et al.*, "Iot-cloud service optimization in next generation smart environments," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 12, pp. 4077–4090, 2016.