Energy Efficient Scheduling for Networked IoT Device Software Update

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

***Abstract*—Software in IoT devices needs to be improved regularly to adapt security issues and new user requirements. In advanced IoT networks, devices employ the component-based software architecture in which components can be updated at run-time, such devices can download software components from neighbors, enabling fast distribution of updates in the entire network. One of the most energy consuming operations in the update process is flash re-writing in which the order of re-writing components into the flash memory is decisive for energy consumption. In this paper, we propose a mechanism that schedules updates in an entire IoT network to minimize the energy consumption, while satisfying the deadline constraint for updating all the devices. We mathematically formulate the problem of energy efficient update scheduling as an optimization problem with a novel energy model of the update process, then propose an algorithm to approximate the optimal schedule for updating all devices in the network. We examine the proposed algorithm in three different network instances including a tree, a partial mesh and a full mesh topology. Simulation results illustrate that our algorithm can obtain a near optimum which is, in the best case, only 3.2% different from the minimum.**

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

I. Introduction

The scale of the Internet of Things brings many challenges to deployment and management. In order to adapt incremental user requirements of IoT applications, software in IoT devices needs to be changed regularly to improve functionalities. Software update must become an crucial task to maintain effective performance of IoT systems [1].

Research on software update for IoT/wireless sensor networks can be categorized into three main topics: 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 divided into small modules, so-called components, which can be separately updated at run-time. In such environment, only portions of the whole software need to be replaced through the update process, enable to reduce the amount of data needed to deliver.

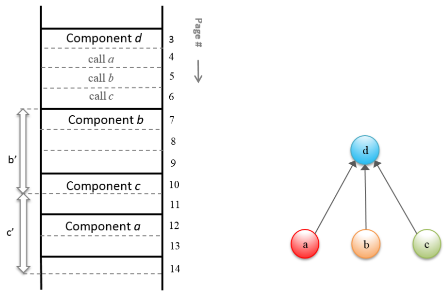
Inside a device, software components are placed in an order in the flash memory as shown in Fig. 1a. Each component occupies several memory pages, when a component is updated (assume its size changes), its memory pages have to be re-written completely and all the components laid next to it in the flash need to be shifted to other addresses [8]. Therefore, all these components also have to be re-written. In this situation, different component update orders can result in different numbers of re-written pages. In addition, in component-based software systems, some components may call the others during their operation [9]. This dependency leads to an update order constraint, in which a component can only be replaced when all components it depends on had been updated. Otherwise, an inconsistency error would be obtained. Because a large amount of energy is consumed for flash re-writing[10], defining an optimal update order is significant for decreasing energy consumption in device software update operation. Some work [11] also mentioned the update order constraint, however, this constraint and its impact on energy have not been considered carefully in previous studies.

Existing work on component-based software for IoT devices often focuses on the ways a component is updated [9], [12], [13], 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 IoT network consisting 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 from the gateway. 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 specifies two decisions: First, where a component can be downloaded for each device; second, when it can be downloaded.

II. System description

A. IoT sofware components

We consider the case in which a gateway downloads software updates from the cloud, and then send to a number of devices of the same type. A device does not have to update all new components at the same time, but one by one. Since components may call others, their dependency causes the order constraints that need to be satisfied by the update schedule.



(a) Components in flash memory of an IoT device, *b* and *c* are updated (b) Software component constraint graph

Fig. 1. Software components in flash memory of an IoT device and the corresponding component constraint graph.

The software component constraint is denoted by a directed graph $D =< V_D, A_D >$ with component set V_D and arc set A_D presenting component constraints. D can be represented by a matrix $\mathcal{M}_D = \{c_{m,n}\}$ 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. 1b.

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 are located in the flash as shown in Fig. 1a. When c is updated, suppose that the new size of c increases compared to the previous size (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. 1a has to be re-written.

B. System model

We focus on a model of an IoT network including a number of connected IoT devices and a gateway. 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 $V_G = \{i|\ i = 0, 1, \ldots |V_G|\}$, in which $i = 0$ represents the gateway, and IoT devices are corresponding to $i > 0$. We denote G by a symmetric matrix $\mathcal{M}_G = \{b_{i,j}\}$ where each entry $b_{i,j}$ presents the bandwidth of the link between two nodes i and j.

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. The amount of time to perform update in the entire network is limited by a deadline T_{max}.

III. Problem formulation

A. Energy consumption model

We define two set 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 device i downloads component m. The update schedule of each device i is characterized by the sets $\{a_{i,j,m}\}$ and $\{x_{i,m}\}$. We denote the sizes of a component $m \in V_D$ before and after updating by s_m^{old} and s_m^{new}. The duration of device i to completely download component m can be calculated as follows:

$$t_{i,m} = \begin{cases} 0, & i = 0, \\ \dfrac{s_m^{new}}{\sum\limits_{j \in V_G} a_{i,j,m} b_{i,j}}, & i > 0. \end{cases} \tag{1}$$

The amount of energy consumed when a device i updates a component m can be calculated by multiplying the energy for writing one flash page with the number of re-written pages:

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

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 $\beta(m) = V_D \backslash (\alpha(m) \cup m)$ is the set of components lie before m in the flash memory. The binary indicator $c_{h,k} \in \mathcal{M}_D$ 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 updating m, i.e., $size(h)$ is s_h^{new} if h is updated before m, otherwise $size(h)$ is s_h^{old}:

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

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

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

We can see that the quantity $\sum\limits_{h \in \alpha(m)} \sum\limits_{k \in \beta(m)} c_{h,k}$ in equation (2) is constant and does not depend on the update order, so this quantity can be skipped 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:

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

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:

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

In Eq. 6, E_i is a function of $\{a_{i,j,m}\}$ and $\{x_{i,m}\}$.

B. Optimization model

The optimization model for our scheduling problem is formulated as:

$$\min \sum_{i=1}^{|V_G|} E_i. \tag{7}$$

Subject to:

$$x_{i,m} \geq 0, \quad \forall i \in V_G, i > 0, m \in V_D. \tag{8}$$

$$x_{0,m} = 0, \quad \forall m \in V_D. \tag{9}$$

$$\sum_{j \in V_G} a_{i,j,m} = 1, \quad \forall i \in V_G, i > 0, m \in V_D. \tag{10}$$

$$a_{i,j,m} \leq \phi(b_{i,j}), \quad \forall i, j \in V_G, i > 0, m \in V_D. \tag{11}$$

$$a_{i,j,m}(x_{i,m} - x_{j,m} - t_{j,m}) \geq 0, \quad \forall i, j \in V_G, i > 0, m \in V_D. \tag{12}$$

$$a_{i,j,m}a_{i,j,n}(x_{i,m} - x_{i,n} - t_{i,n})(x_{i,n} - x_{i,m} - t_{i,m}) \leq 0, \quad \forall i, j \in V_G, m \neq n \in V_D. \tag{13}$$

$$c_{m,n}(x_{i,n} - x_{i,m} - t_{i,m}) \geq 0, \quad \forall i \in V_G, m, n \in V_D. \tag{14}$$

$$x_{i,m} + t_{i,m} \leq T_{max}, \quad \forall i \in V_G, m \in V_D. \tag{15}$$

In our model, constraint (9) states that the gateway does not download from any source. Condition (10) indicates that a device only downloads a component m from one other node. Constraint (11) is the network topology constraint, a device i can download from device/gateway j only if there is a link (i, j); where $\phi(b_{i,j}) = 1$ if $b_{i,j} > 0$, means that link (i, j) exists, otherwise $\phi(b_{i,j}) = 0$ if $b_{i,j} = 0$. Constraint (12) means that 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 (13) states that a device can only download one component from each other node at a time. Constraint (14) indicates the download order of each device needs to satisfy the component constraint graph. And finally, condition (15) is the deadline constraint T_{max}.

Algorithm 1: P_1 - Generate a schedule

```
1 repeat
2     Each step, do
3     for each device $i$ do
4         Construct the bipartite graph $B_i$;
5         Do $Matching$ the bipartie graph $B_i$;
6         With $\{m\}$ is the set of downloadable
           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;
7     end
8     Calculate the finishing time of this step;
9 until all nodes complete downloading all components;
```

Algorithm 2: ESUS Algorithm

```
1 for $t$ from $1$ to $N$ do
2     Generate schedule $S_t$ by $P_1$;
3     if $S_t$ does not satisfy $Tmax$ then
4         Adjust $S_t$ by $P_2$;
5     end
6     if $S_t$ still does not satisfy $Tmax$ then
7         Start new iteration $t + 1$;
8     end
9     else
10        if $S_t$ is better than current best solution then
11            Update the best solution is $S_t$;
12        end
13    end
14 end
```

IV. Proposed algorithm

We propose an algorithm called ESUS, it 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 divides the schedule into steps. At each step, each device i maintains a list of downloadable components and a list of possible sources, that can be represented as a bipartite graph B_i. P_1 finds a matching of B_i with the purpose to maximize the number of downloadable components. After that, P_1 calculates $x_{i,m}$ for each downloadable component m so that the order of 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.

In case T_{max} is not satisfied by the initial schedule given by P_1, ESUS uses 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. For each device i downloads m, it checks if $x_{i,m}$ 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 it changes $x_{i,m}$ to the earliest as possible. P_2 iterates the components in a

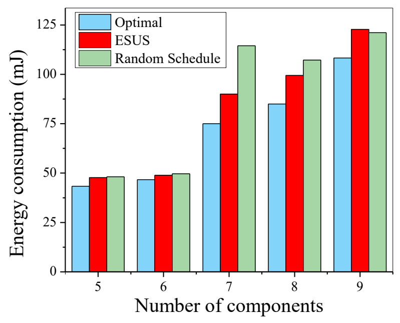


Fig. 2. Energy consumption with different component sets in the tree topology with 10 nodes.

topological order, it means that when examining a component m, all the components that m depends on are already adjusted. The outline of ESUS is presented in Algorithm 2.

V. Simulation results

In our simulation, we examine the proposed scheduling algorithm in different network instances. To evaluate results of ESUS, we use the CPLEX solver to find optimal schedules of the 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.

A. *Settings*

Network settings. We define two typical topologies of IoT networks, that are tree and full mesh. For simplicity, we set the bandwidth of every connection between a device and the gateway by $b_g = 4\ KB/s$, and the bandwidth of each connection between two devices is set by $b_d = 8\ KB/s$.

Optimization settings. For each component set, every s_m^{old} is set to the same fixed size, the corresponding s_m^{new} is set by a multiplication of s_m^{old} and a random number, and a constraint graph D is also randomly created. 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 $4\ KB$.

B. *Tree topology*

In the first scenario, we examine the network of ten nodes in which device connections form a tree rooted at the gateway. Fig. 2 shows the results corresponding to different software component sets. 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.

C. *Full mesh topology*

In the second 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.

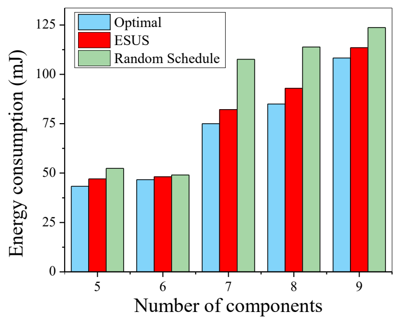


Fig. 3. Energy consumption with different component sets in the full mesh topology with 10 nodes.

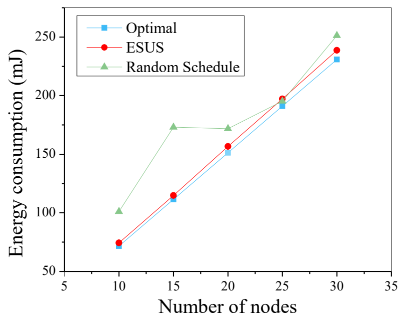


Fig. 4. Energy consumption with different number of nodes in the full mesh topology.

1) *Evaluation of different software component sets:*

Fig. 3 shows the results on a network instance of ten nodes with the same software component sets as in the first scenario. 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. 3 also shows that ESUS outperforms the random schedules, with up to 30.8 % re-written pages saved.

2) *Evaluation of different number of nodes:*

In another sub-scenario, we fix the component set and examine the results with full mesh networks of different numbers of nodes. As shown in Fig. 4, ESUS can approximate the optimal solutions in all cases, and its results are better than random schedules in most cases.

VI. Conclusion

In this paper, we have introduced the problem of energy efficient software update scheduling in IoT networks. We built a novel energy model for the update process and formulated the problem as an optimization problem. We then proposed ESUS algorithm to find a near-optimal solution. Simulation results showed that our algorithm can effectively approximate the optimal solution given by CPLEX solver. In the future, we will extend our work by considering 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] W. Dong *et al.*, “Optimizing relocatable code for efficient software update in networked embedded systems,” *ACM TOSN*, vol. 11, no. 2, p. 22, 2015.

[9] 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.

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

[11] 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.

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

[13] 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.