Live Code Update for IoT Devices in Energy Harvesting Environments

Raksha S[1], Sushmitha K[1], Deepika B[1], Ganavi N[1], Chandini S B[2]

[1]*Student, Dept. of Information Science and Engineering, VVCE, Mysuru, Karnataka, India*
[2]*Assistant Professor, Dept. of Information Science and Engineering, VVCE, Mysuru, Karnataka, India*

---***---

Abstract - *There is a rapid rise in growth exhibited by number of Internet of Things (IoT) devices. IoT devices is usually connected with the physical world. These devices may harvest energy as a power source, which inflict particularly rigid operating restrictions. In addition IoT devices may need software updates to fix bugs, add functionality, or input computational capability. This paper proposes in-place code updating to update deployed code for IoT energy harvesting devices. This strategy applies patches in-place while the code is still running and active; which competently eliminate system down time and reduce resource demands for updates.*

Key words: Live Code, Live code Update for IoT devices, IoT devices, Energy Harvesting Environments, Live Code for IoT devices.

1. INTRODUCTION

Internet of Things (IoT) is an ecosystem of connected physical objects that are accessible through the internet. The 'thing' in IoT could be a person with a heart monitor or an automobile with built-in-sensors, i.e. objects that have been assigned an IP address and have the ability to collect and transfer data over a network without manual assistance or intervention. The embedded technology in the objects helps them to interact with internal states or the external environment, which in turn affects the decisions taken.

The Internet of Things (IoT) is a novel computing paradigm that couples sensing devices, computing nodes, communication devices with various types of objects in physical world for data collection, exchange, and remote control. IoT devices often have very tight constraints on cost, form factor, and power/energy consumption. These devices often rely on ambient power sources such as wireless energy, RF energy, solar energy, and piezoelectric energy.

The ambient power is not only scanty but also often unreliable. This makes it necessary to equip these devices with non-volatile memory to store program state in order to ensure forward progress. Often state has to be stored and only a couple of instruction can be executed per power cycle [1].

It is obvious that, IoT devices may need software updates to fix bugs, add functionality, or improve computational capability. So how to deliver code updates to the energy harvesting devices post-deployment is another critical concern. This project proposes novel strategies to update deployed code for IoT energy-harvesting devices based on in-place code updating.

Our objective is to propose a novel in-place patching strategy for the three code update scenarios of insertion, deletion, and modification that minimizes the down time of a device. We aim to develop strategies on device down time, code update delivery cost, code memory update cost, and runtime performance overhead.

2. EXISTING SYSTEM

Currently post-deployment code update schemes focus mainly on reducing the amount of data transferred over the wireless network. This includes proposals involving naïve, incremental updates, modular designs, and network encoding. These incremental update schemes attempt to minimize the code transferred to a device by sending only the "delta" difference between the old and new images, instead of sending the entire image. The new image then is constructed from the old image and the delta [2], [3], [4], [5], [6].

These approaches have the following steps:

- transmit the code update,
- construct a new image if the code update was a delta, and
- Reboot the device using the new code image.

This approach is called image rewrite. This method has several limitations.

- ➢ QOS level of the device is degraded.
- ➢ Rebooting and re-launching is required during which it is unresponsive to external events. This delay is down time of the device [7], [8].
- ➢ It takes longer for code updates to be delivered to devices because existing image rewrite approaches do not work well with incremental updating.

- Image rewriting takes longer for code updates to be applied once the patches are delivered because the whole new code image must be rewritten to code memory, even for small updates.
- Image rewriting increases the cost of device by requiring larger code memory. Image rewrite approaches require the old image to be running while constructing or downloading the new image.

3. PROPOSED SYSTEM

In energy harvesting systems for IoT devices we propose the idea of live code updates. By avoiding the above problem in-place through patches we update the code image; while the code is still executing and live. It is a form of cumulative code update in which only delta script is sent to the target device.

Different from other additive update proposals, instead of formulating a new image, the delta is applied in-place on live code. Since the delta is applied in-place, there is no shifting involved.

In-place code needs to manage the situation in which code being updated could be actively executing and live, unlike image rewriting. A code update usually involves multiple patches in which execution of code being patched must be interrupted to avoid unstable code. Therefore, the goal of in-place code updating is to decrease the set of code memory writes during which the code drops into an unstable state. Hence we call this set of writes the atomic update set. To minimize the automatic update set we propose code patch strategy.

It proposes in-place patching for insertion, deletion and modification code update;
Code insertion is performed by placing a jump from the location of the insertion point in the original image to the inserted code. Following the insertion point a jump back to the instruction immediately is added to continue execution in the original image, at the end of the inserted code.

Code deletion is performed by placing a jump from the location of the first deleted instruction in the original image to another jump instruction. The next jump jumps back to the original image directly following the last deleted instruction.

Code modification is performed in a regular fashion except, after execution of the modified code, control flow jumps to the point after the old code in the original image. A code update with multiple patches has two phases:

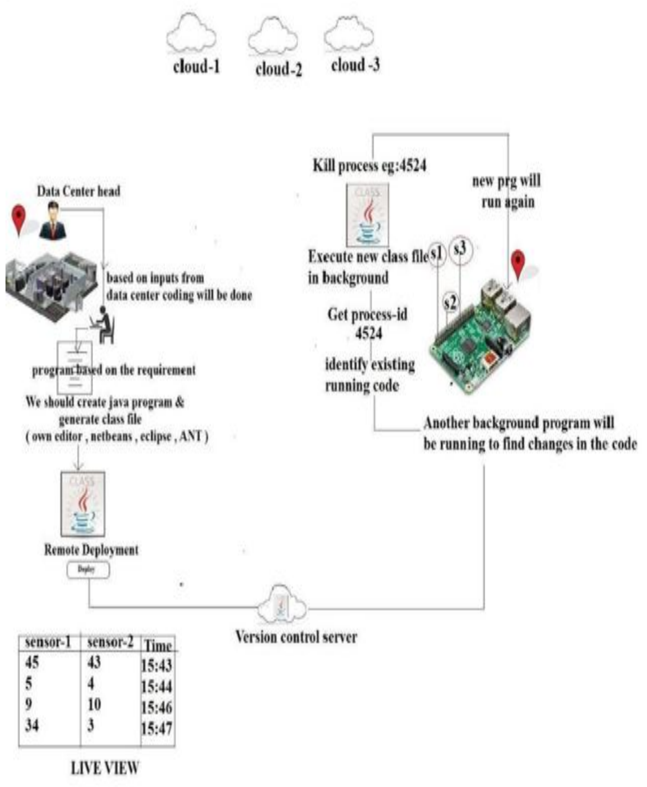
Phase 1: The modified or all new code is written to free space in code memory. Since no writes are performed on the original image, the code is never inconsistent.

Phase 2: All jumps to the code written in Phase 1 are written atomically. Each of these jumps has the potential to put the code into an inconsistent state and modifies the functionality of the old image. Until all writes completes the execution of this code must be prevented. Rather the size of patches themselves, the atomic update set of jumps to patched code is proportional to the number of locations which needs to be patched in the original image.

3.1 Code Update Approach Comparison

	Device Down Time	Reprogramming Energy	Memory Requirement
Naïve	✗ (Need reboot)	✗ (Transfer and write whole image)	✗ (Need twice code image size)
Incremental	✗ (Need reboot)	Δ (Transfer only delta but write whole image)	✗ (Need twice code image size)
In-place	✓ (No reboot needed due to live update)	✓ (Transfer only patches and apply in-place)	✓ (need only code image size)

3.2 System Architecture


cloud-1
cloud-2
cloud -3
Data Center head
based on inputs from data center coding will be done
program based on the requirement
We should create java program & generate class file (own editor , netbeans , eclipse , ANT)
Remote Deployment
Deploy
Kill process eg:4524
new prg will run again
Execute new class file in background
s1
s3
s2
Get process-id 4524
identify existing running code
Another background program will be running to find changes in the code
Version control server
LIVE VIEW

sensor-1	sensor-2	Time
45	43	15:43
5	4	15:44
9	10	15:46
34	3	15:47

Figure -1 System Architecture

4. MODULES

The proposed system consists of five modules;

4.1 Developer side option to create script and maintain version control

This module involves creating an editor for the developers to write programs. The editor involves options to create new file, write code, view existing code, make changes to existing code, save and upload the code. The developer takes inputs from the users or the data centre head. He/she then creates java programs based on the requirements.

To maintain version control:

- For each save and updates of code maintain sequence number
- If (updates found && sequence_no == prevseq)
  - maintain new copy with same sequence
  - keep the code ready for update
- Otherwise
  - sequence++
  - Old patch saved and new file copy kept ready for update

4.2 Live code update via cloud

In this module the code update is uploaded to the cloud. After developer creates java programs and generates class files he/she then uploads it to the cloud.

It involves following steps:

- Start the cloud service
- USR_AUTH _KEY ← call auth_key(User Name, PWD) –Authentication key registered in UIDaaS
- If (USR_AUTH_KEY==AUTH_KEY)
  — Users are granted permission to access cloud services
  Update();
- Otherwise
  — Users are not granted permission to access cloud service

4.3 Patch manager to handle new copy , compile and execute code

Patch manager is embedded in the Raspberry pi. This unit continuously checks for new updates by running a background process. Whenever new updates are found it runs the new program.

The objective of the Patch manager function is as follows:

```
patchManager( )
{
    • Check the version id and process id
    • Check the code updates
    • If updates found, kill existing process
      and get new code
    • Threaded program to run new code
}
```

4.4 Application to view live data from IoT based on new changes

Once the code has been updated it is necessary for the users to see the changes made. For this purpose a module to view live data is included. It involves creating UI application for the users to read the updated data values in their required format.

It consists of the following step:

```
• User needs to connect to the cloud with
  authentication
• If (authenticated)
  {
      readSensorData(int sensorId)
      {
        — Read operation called by reader
        tasks
          — Find data size
          — Create a buffer
          — Read data from cloud
                acquire(sensorId)
          — Display data at user interface
      }
  }
```

4.5 Integration module

The functionality of this module is to integrate the above mentioned modules and making sure that the entire system as a whole works as required. That is to update the codes in-place through patches; while the code is still live and still executing.

5. ADVANTAGES OF THE EXISTING SYSTEM

The proposed system has the following capabilities:

- In-place updating reduces the problems existing with traditional methods
- Does not suffer from QOS degradation
- Code update delivery cost is reduced
- Reduces the cost to apply the code update
- Does not require to hold two images i.e., requires less memory
- Rebooting and rewriting is not involved thus reducing system down time

6. CONCLUSION AND FUTURE ENHANCEMENT

In this paper we proposed live code update solution that effectively applies patches to code as it executes for IoT devices that can; Improve QOS by eliminating device down time, Improve debug/tune/upgrade turn-around time by shortening device reprogramming energy. It significantly reduces memory requirements of the device as it does not keep the old image. Finally it offers insignificant performance overhead.

The project can be further enhanced by generating less number of patches. Frequent patching may lead to creation of "holes" in the code. This includes future works to deal with fragmentation. May also require further research on how to perform updates to data; modify semantics of variables and may need to do "data migration".

REFERENCES

[1] "NAND Flash 101; An Introduction to NAND Flash and How to Design It In to Your Next Product", https://www.micron.com/~/media/documents/products/technical-note/nand-flash/tn2919_nand_101.pdf.

[2] R. K. Panta, S. Bagchi, and S. P. Midkiff, "Zephr: Efficient Incremental Reprogramming of Sensor Nodes Using Function call Indirections and Difference Computation", in USENIX Annual Technical Conference, June 2009.

[3] R. K. Panta and S. Bagchi, "Hermes; Fast and Energy Efficient Incremental Code Updates for Wireless Sensor Networks", in International Conference on Computer Communications (INFOCOM), April 2009.

[4] W. Dong, Y. Liu, C. Chen, J. Bu, C. Huang, and Z. Zhao, "R2: Incremental Reprogramming Using Relocatable Code In Networked Embedded Systems," IEEE transactions on Computers, Sept 2013.

[5] W. Dong, C. Chen, J. Bu, and Y. Liu, "Optimizing Relocatable Code for Efficient Software Update In Networked Embedded Systems," ACM Transactions on Sensor Networks, July 2014.

[6] W. Li, Y. Zhang, J. Yang, and J. Zheng, "UCC: Update-conscious Compilation for Energy Efficiency in Wireless Sensor Networks," in conference on Programming Language Design and Implementation (PLDI), June 2007.

[7] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, "Fidelity and Yield in A Volcano Monitoring Sensor Network," in Symposium on Operating System Design and Implementation (OSDI), November 2006.

[8] Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J Regehr, "Surviving Sensor Network Software Faults," in Symposium on Operating Systems Principles (SOSP), October 2009.