Vol.2021-SE-207 No.32
2021/3/2

# IoTデバイス内アプリケーションの開発効率向上のためにコードの変更を動的に適用する方式の提案と実装

栗林 健太郎[1,2,a)]　山崎 進[3]　力武 健次[4,1]　丹 康雄[2]

**概要**：IoTデバイスは多様な用途において増え続け，2030年にはその数が1250億に達すると見込む調査報告がある．増え続ける多様な需要を満たすためには，IoTデバイスの開発効率の向上が必要であり，そのための開発プラットフォームが多数現れている．IoTデバイス内アプリケーションの開発において，開発者によるコードの変更を適用することで生じる動作の変更が意図した通りであるかどうかを確認するためには，変更内容をターゲットとなるデバイスへ適用し実際に動作させる必要がある．既存方式では，更新内容の生成および適用に加えて，デバイスの再起動に時間を要するため，迅速な開発サイクルの実現が困難である．本研究では，先行研究に基づきコードの変更をデバイスへ適用する方式について（1）ファームウェアイメージの全体を適用する方式，（2）ファームウェアイメージの差分を適用する方式，（3）アプリケーションコードを動的に適用する方式の3つに分類した．その上で，開発効率の向上を目的として（3）を動的な性質を持つ言語によって実装し得る方式として位置づけ直して提案するとともに実装し，各方式について更新に要する時間を比較検討した．その結果，提案方式は既存方式に比べて更新に要する時間が95%短くなった．

# A Method to Realize a Rapid Development Cycle of IoT Applications by Dynamically Applying Local Code Changes to the Devices

Kentaro Kuribayashi[1,2,a)]　Susumu Yamazaki[3]　Kenji Rikitake[4,1]　Yasuo Tan[2]

***Abstract:*** Software development efficiency should be improved to meet the ever-increasing demand for IoT devices, which is expected to reach 125 billion by 2030. Application of the code changes to the target devices and running the code on the devices are necessary to check whether the behavioral changes caused by the code changes by the software developers in IoT applications are as intended. The existing method takes time to generate and apply the updated contents and restart the device, making it difficult to realize a rapid development cycle. In this study, we have classified the methods of the code changes application to the devices into three categories based on the previous research: (1) rewriting the entire firmware image; (2) updating by applying the differences in the firmware image; and (3) applying the application code dynamically. To improve the development efficiency, we proposed and implemented the third method as it can be implemented by a language with dynamic characteristics, and compared the time required to update each method. Our research results show that the proposed method reduces the required time for updating by 95% compared to the existing method.

## 1. はじめに

IoTデバイスは年々増え続け，2030年にはその数が1250億に達すると見込む調査報告がある [1]．また同調査報告

[1] GMOペパボ株式会社 ペパボ研究所
Pepabo R&D Institute, GMO Pepabo, Inc., Fukuoka City, Fukuoka 810-0001, Japan
[2] 北陸先端科学技術大学院大学
Japan Advanced Institute of Science and Technology, Nomi City, Ishikawa 923-1292, Japan
[3] 北九州市立大学
University of Kitakyushu, Kitakyushu City, Fukuoka 808–0135, Japan
[4] 力武健次技術士事務所
Kenji Rikitake Professional Engineer's Office, Setagaya City, Tokyo 156-0045 Japan
[a)] antipop@pepabo.com

は，スマートホーム機器のような家庭での利用はもとより，医療用途や製造業等における産業用途，エネルギーやモビリティに関わる社会インフラ基盤にいたるまで，多岐にわたる領域において IoT の活用が進むとしている．それらの増え続ける多様な IoT デバイスの需要を満たすためには，開発者が IoT デバイスを開発する効率を向上させることが必要である．

ハードウェアおよびソフトウェアの両面にわたる開発効率の向上のため，IoT デバイスの開発プラットフォームが多数現れている．ESP8688（およびその後継の ESP32）による開発キット [2]，Arduino [3]，Raspberry Pi [4]，Beagle Bone [5]，NVIDIA Jetson [6] 等がその例である．それらのプラットフォームは，ハードウェア開発効率向上のため，マイクロプロセッサや電源とともに，センシングやアクチュエーションのための様々な規格に対応する入出力インタフェイスや，Wi-Fi や Bluetooth モジュールといったネットワークインタフェイスを提供している．ソフトウェア開発効率向上のため，開発元やオープンソースソフトウェア（OSS）等により Arduino IDE [7]，ESP-IDF [8]，Platform.io [9] といった統合的な開発環境が提供されており，また，それらは開発者によるコードの変更を有線あるいは無線ネットワークを経由して行える OTA（over-the-air）による更新機能を備えている．上述した IoT デバイスを構成するハードウェアおよびソフトウェア，それらに対応する開発プラットフォームを**表 1** の通り整理できる [10, p.180][*1]，[11][*2]．

表 1 に示した IoT アプリケーションの開発には，パーソナルコンピュータ向けのアプリケーションや Web アプリケーション等の開発とは異なる困難さがある．パーソナルコンピュータ向けのアプリケーション開発においては，開発者の使用するホストとターゲットとなるホストのプラットフォームおよびアーキテクチャが同一である場合，開発者の使用するホスト上で動作確認を行える．また，Web アプリケーション開発においては，開発者の使用するホストとターゲットとなるホストのプラットフォームやアーキテクチャが異なったとしても，VirtualBox [13] や Docker [14] 等の仮想化技術を用いることで，開発者のホスト内で動作確認を完結する手法が発展している．一方で，IoT アプリケーションの開発においては，開発者が変更したコードの動作を確認するためには，なんらかの方法で変更内容をターゲットとなるデバイスへ適用し，IoT アプリケーションの動作を変更する必要がある．ターゲットとなるマシンのプラットフォームやアーキテクチャに加えて，操作対象となるハードウェア構成が開発者の使用するホストとは異なるためである．さらに，IoT アプリケーションのプロトタイピング時や開発時には，小規模な変更による頻繁な更新を必要とする．そのため，IoT アプリケーションの開発効率の向上を実現するには，開発者によるコードの変更を効率的にデバイスへ適用することが課題となる．

[*1] ハードウェアを構成する要素として電源を [10] の定義に加えた．
[*2] IoT の文脈では，IoT アプリケーションはエンドユーザ向けのアプリケーションを指す [12] が，本研究では [11] に従い IoT デバイス内アプリケーションを IoT アプリケーションと呼称する．

**表 1** IoT デバイスを構成するハードウェアおよびソフトウェア，それらに対応するプラットフォームの整理

**Table 1** Organization of the hardware and software that make up IoT devices and the platforms that support them.

| | Components | Platforms |
|---|---|---|
| Hardware | Sensors | [2–6] |
| | Actuators | |
| | Microprocessors | |
| | Network Interfaces | |
| | Power Supply | |
| Software | IoT Applications | [7–9] |
| | Network Protocol Stack | |
| | Operating System | — |
| | Hardware Drivers | |

開発者によるコードの変更をデバイスへ適用する方式については，これまで多くの先行研究が行われている．[15,16] は，コードの変更をデバイス上へ適用する方式を，書き換え対象に基づき（1）IoT アプリケーションの実装に用いられるスクリプト言語の特性を利用した動的なコード書き換え，(2) デバイス上で動作する仮想機械内で実行されるコードの書き換え，(3) デバイス上のファームウェアイメージの書き換え，(4) デバイス上のネイティブコードへの動的リンクの書き換えの 4 つに分類している．また，コードの変更をデバイスへ適用する状況について [15] は，デバイスの配備前における開発・検証，および，デバイスの配備後における機能追加・更新・拡張の 2 つに分類しており，それぞれについて更新頻度を整理している．それらの先行研究は，利用可能なネットワーク帯域，ハードウェアスペック，電源容量・確保において制約の厳しい IoT デバイスに対して，いかにして安全かつ効率のよい更新が可能であるかを主たる課題としている．そのため，上述の（1）および（2）についてはそれらの制約のもとでの実現は困難であるとして，検討対象外としている [15, 16]．また，更新の目的をデバイスの配備後におけるセキュリティ対策，バグ修正，機能拡張においているため，本研究が課題とする IoT アプリケーション開発効率の向上は目的とされていない．

本研究では，IoT アプリケーションの開発効率の向上という観点から，先行研究におけるコードの変更をデバイスへ適用する方式を，書き換え対象に基づき（1）ファームウェアイメージの全体を適用する方式，(2) ファームウェアイメージの差分を適用する方式，(3) アプリケーションコードを動的に適用する方式の 3 つに分類し直す．ファームウェアイメージの書き換えに基づく更新について，差

分による効率的な更新を実現する研究が進んでいる [17]. そのため，全体を更新する方式を（1），差分を更新する方式を（2）として 2 つに分けて検討することが妥当である．IoT アプリケーションの開発効率を向上を目的として，前述の IoT デバイスの開発プラットフォームを用いたプロトタイピングが広く行われている．それらのプラットフォームは，開発者によるコードの変更をデバイスへ迅速に適用するために，有線あるいは無線ネットワークを経由したアップデートの方法を提供している．また，Raspberry Pi や Beagle Bone，NVIDIA Jetson などのように，Linux ベースのファームウェアが動作するハードウェアスペックを備えたプラットフォームも存在している．IoT アプリケーションのプロトタイピング時や開発時においては，スクリプト言語や仮想機械上で動作する言語のような，ネイティブコードと比較してリソースを相対的に多く要求する言語を用いることは，開発効率の向上への寄与を目的として許容可能である．そのため，更新対象としての IoT アプリケーションを構成するコードは，ネイティブコードはもとより，スクリプト言語や仮想機械上で動作する言語による動的な性質を持つコードであっても，同等の機能を果たすとみなせる．よって，IoT アプリケーションの開発効率の向上という観点からは，アプリケーションのコードの変更をデバイスへ動的に適用する方式を，上述の方式（3）にまとめて検討することが妥当である．

本研究は，IoT アプリケーションの開発効率の向上を目的とした，開発者によるコードの変更を IoT アプリケーションを構成するコードに対して動的に適用する方式について提案する．また，先行研究における，デバイスの配備後という強い制約下での更新方式という観点においては十分には検討されてこなかった，スクリプト言語や仮想機械上で動作する言語による動的な性質を持つコードを用いて提案方式を実装する．具体的には，開発者により変更された IoT アプリケーションを部分的に構成するコードを，ファームウェアイメージの生成を行うことなくネットワークを経由してデバイスに適用した上で IoT アプリケーションの更新を行う．提案方式の実装には，Elixir [18] を用いる．IoT デバイスの開発プラットフォームである Nerves [19] は，仮想機械上で動作する Elixir を開発言語として採用しているため，提案方式を仮想機械上で動作する言語を用いて実現できることに加えて，デバイス上のファームウェアイメージの全体および差分による更新に対応しているため，本研究の整理した方式を網羅できるからである．検証に際しては，デバイス上のファームウェアイメージの書き換えを全体として行う方式と部分的に行う方式とをベースラインとして，更新に要する時間を提案方式の実装と比較検討した．その結果，既存方式がそれぞれ 66.88 秒，70.63 秒であったのに対して，提案方式では 3.30 秒で開発者によるコードの変更をデバイスへ適用することができ，95%の速度向上を実現できた．

本研究の貢献は以下の通りである．

（1） IoT アプリケーションの開発効率の向上を目的とした場合，動的言語によるアプリケーションを構成するコードの動的な変更が有効であることを示した．
（2） 提案方式の実装を行い，ファームウェアイメージを更新する方式と比較した上で，更新に要する時間の短縮という定量的な指標において優位性を示した．
（3） 本研究の示した結果が，IoT デバイスの開発プラットフォームにとって IoT アプリケーションを構成するコードの動的な書き換えをサポートすることが優位性となることの示唆となった．

本論文の構成を述べる．2 章で，開発者によるコードの変更をデバイスへ適用する方式について先行研究に基づき検討し，課題を述べる．3 章で，既存方式の課題を解決する提案方式について述べる．4 章で提案手法について実験に基づく評価を行い，5 章でまとめる．

## 2. 本研究の背景

本章では，本研究の背景について述べる．はじめに，2.1 節で先行研究について検討する．次に，2.2 節で本研究の観点について述べる．最後に，2.3 節で本研究の観点に基づいて開発者によるコードの変更をデバイスへ適用する方式を整理する．

### 2.1 先行研究

IoT デバイスは，一度配備された後も継続的にソフトウェアの更新が必要である．また，更新に際しては IoT デバイスのリソース制約の厳しさが課題となる．[20] は，IoT デバイスの製品ライフサイクルを初期・中期・後期の 3 段階に分類した上で，各段階において有効なセキュリティ対策についてまとめている．IoT デバイスのソフトウェア更新は製品ライフサイクルの全段階を通して必要であるとし，リソース制約の厳しい IoT デバイスに対していかにして効率的かつ安全に更新を実行できるかという観点で先行研究を検討している．[21] は，Wireless Sensor Networks を構成するデバイスへのソフトウェア更新について，広範なサーベイを行った．デバイスのデプロイ後のフェーズに着目し，フィールドに配置された，直接はアクセスできない多数のデバイスに対するソフトウェア更新を可能とする様々なプロトコルを検討している．[11] は，IoT デバイスのソフトウェア更新のプロセスを更新内容の検証，および，更新内容の配布の 2 つのフェーズに分類した上で，リソース制約の厳しい IoT デバイスに対してはエネルギー消費量のより少ない方式が有効であることを示している．

上記を背景に，開発者によるコードの変更をデバイスへ適用する方式について様々な提案がなされている．Rucke-

busch らは，[15,16] においてコードの変更をデバイス上へ適用する方式を，書き換え対象に基づき（1）IoT アプリケーションの実装に用いられるスクリプト言語の特性を利用した動的なコード書き換え，（2）デバイス上で動作する仮想機械内で実行されるコードの書き換え，（3）デバイス上のファームウェアイメージの書き換え，（4）デバイス上のネイティブコードへの動的リンクの書き換えの 4 つに分類している．このうち（3）の方式は，更新対象となるファームウェアのサイズが他の方式と比較して大きいことから，ネットワーク経由での更新内容の転送に時間を要し，デバイスへの適用において電力をより多く消費する．そのため [17,22] は，（3）の方式による更新を効率化するために，ファームウェアイメージの生成プロセスをソースコードレベルでの類似性の向上，および，ファームウェアイメージの差分生成のアルゴリズムの改善の 2 つの観点で整理し，[22] はファームウェアイメージの差分のサイズを従来方式よりも縮小する方式を提案している．

### 2.2 本研究の観点

本研究は，IoT アプリケーションの開発効率の向上という観点から，開発者によるコードの変更をデバイスへ適用する方式を検討する．上記で検討した先行研究は，利用可能なネットワーク帯域，ハードウェアスペック，電源容量・確保等において制約の厳しい IoT デバイスに対して，いかにして安全かつ効率のよいソフトウェア更新が可能であるかを主たる課題としている．そのため，上述の開発者によるコードの変更をデバイスへ適用する方式を提案した [15,16] は，方式（1）および（2）について，それらの研究が対象とする強い制約の課されたデバイスにおいては実現困難であるとして，検討対象外としている．前述の通り [20] は，IoT デバイスのライフサイクル全体を通したソフトウェア更新が必要であることを述べている．また，コードの変更をデバイスへ適用する状況について [15] は，デバイスの配備前における開発・検証，および，デバイスの配備後における機能追加・更新・拡張の 2 つに分類しており，それぞれについて更新頻度を整理している．しかし，それらの先行研究はソフトウェア更新の主たる目的をデバイスの配備後におけるセキュリティ対策，バグ修正，機能拡張等に置いているため，IoT アプリケーション開発効率の向上を目的とするソフトウェア更新の方式については検討されていない．そのため，本研究の観点においては先行研究に加えてさらなる検討を要する．

### 2.3 方式の整理

本研究では，上述した IoT アプリケーションの開発効率の向上という観点から，先行研究の提案した方式を整理し直す．先行研究における（1）から（4）の方式について，（1）（2）および（4）をひとつにまとめる．また，（3）に関する先行研究の積み重ねから，ファームウェアイメージによる更新については全体と差分による方式に分けることが妥当であると考える．その結果，方式を以下の 3 つに分類する．

（1） ファームウェアイメージの全体を適用する方式
（2） ファームウェアイメージの差分を適用する方式
（3） アプリケーションコードを動的に適用する方式

[23] は，IoT デバイスの開発に用いられているボードを網羅的に総覧した上で，それらをスペックに応じて Low-end，Middle-end，High-end の 3 つに分類している．製品化された IoT デバイスの実装に際しては，デバイスの用いられる環境の制約やコストによって Low-end から Middle-end のハードウェアが選ばれたとしても，開発環境としては，たとえば Raspberry Pi のような High-end に分類されるハードウェアを用いることは可能である．また，IoT アプリケーションのプロトタイピング時や開発時においては，スクリプト言語や仮想機械上で動作する言語のような，ネイティブコードと比較してリソースを相対的に多く要求する言語を用いることは，開発効率の向上への寄与を目的として許容可能である．そのため，更新対象としての IoT アプリケーションを構成するコードは，ネイティブコードはもとより，スクリプト言語や仮想機械上で動作する言語による動的な性質を持つコードであっても，開発時においては同等の機能を果たすとみなせる．よって，本研究における IoT アプリケーションの開発効率の向上という観点からは，アプリケーションのコードの変更をデバイスへ動的に適用する方式を，上述の方式（3）にまとめて検討することが妥当である．

## 3. 提案方式と実装

本章では，提案方式とその実装について述べる．はじめに，3.1 節で提案方式について述べる．次に，3.2 節で提案方式の実装について述べる．

### 3.1 提案方式

本研究は，IoT アプリケーションの開発時における開発効率の向上を目的とした，開発者によるコードの変更を IoT アプリケーションを構成するコードに対して動的に適用する方式について提案する．先行研究においても，2.3 節で見た通り，アプリケーションコードを動的に適用する方式自体は提案されてきた．しかし，その目的はデバイスの配備後という制約下での更新方式という観点が主であり，開発効率の向上という観点は検討されてこなかった．また，利用可能なネットワーク帯域，ハードウェアスペック，電源容量・確保等において制約の厳しい IoT デバイスにおいては，必要とするリソースが相対的に大きくなるスクリプト言語や仮想機械上で動作する言語のような動的な性質

を持つ言語は検討外とされてきた．本研究では，IoT アプリケーションのプロトタイピング時や開発時という状況を前提とすることで，開発効率の向上のためにハードウェアを比較的自由に選択した上でリソース要求の大きな言語を用いることができる方式として，アプリケーションコードを動的に適用する方式を提案し，その有用性を主張する．

**図 1** は，2.3 節で整理した（1）ファームウェアイメージの全体を適用する方式，および，（2）ファームウェアイメージの差分を適用する方式の処理の流れを示している．まず，開発者によるコードの変更に基づき，新たなファームウェアイメージを生成するフェーズが実行される．ファームウェアイメージの全体であるか差分であるかの違いはあるが，ビルドフェーズがあること自体は変わらない．次に，ファームウェアイメージをデバイスへ送信し適用するフェーズが実行される．ここでは，ネットワークを経由して送信することを前提としている．最後に，新しいファームウェアイメージをロードするためにデバイスが再起動するフェーズが実行される．

**図 2** は，提案方式の処理の流れを示している．開発者によるコードの変更に基づき，IoT アプリケーションを実行するランタイムに対してコードを送信した上で，デバイス上で動作する IoT アプリケーションを動作させたまま変更されたコードを適用することで，アプリケーションの更新を行う．図 1 とは異なり，コードを逐次的に送信・適用するため，ファームウェアイメージのようなひとまとまりの成果物を生成・送信するフェーズは存在しない．また，IoT アプリケーションを実行するランタイムにおけるコードの適用であるため，IoT デバイス自体の再起動を要しない．

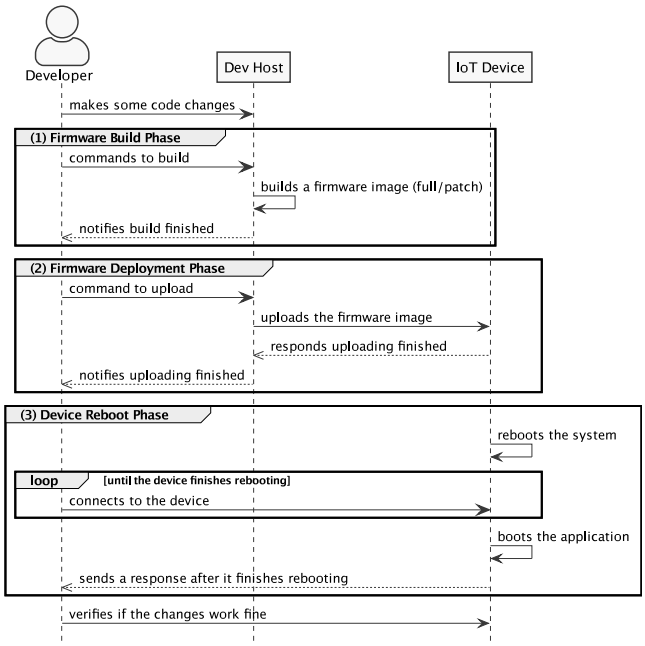


**図 1** 既存方式のシーケンス図
**Fig. 1** Sequence diagram of the existing methods

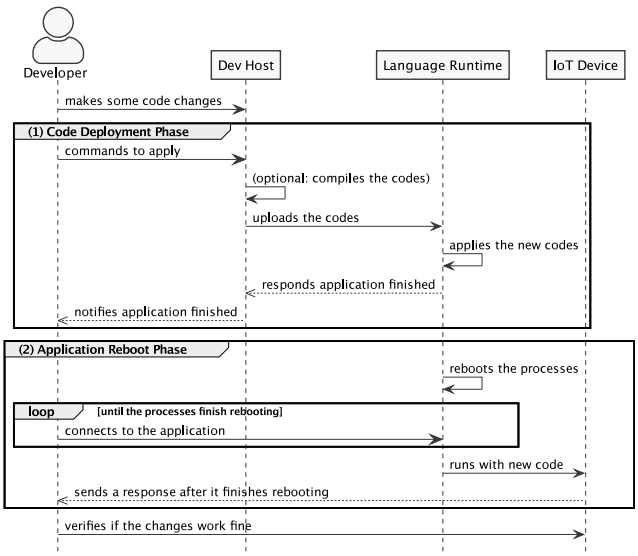


**図 2** 提案方式のシーケンス図
**Fig. 2** Sequnce diagram of the proposed method

### 3.2 実装

提案方式の実装には，プログラミング言語として Elixir [18] を，IoT デバイスの開発プラットフォームとして Nerves [19] を用いる．Elixir は動的型付けの関数型言語であり，大規模な並行プロセス動作を可能にするため，同様の目的で開発された言語システムである Erlang/OTP [24] の動作する仮想機械 Erlang VM 上で動作するよう設計されている．Nerves は，Elixir を Erlang VM 上で動作するのに必要十分なサイズの Linux によるファームウェアを提供するプラットフォームであり，Raspberry Pi や Beagle Bone 等を用いて IoT デバイスのプロトタイピングや開発を迅速に行える仕組みを提供している．また，仮想機械上で動作する Elixir を開発言語として採用しているため，アプリケーションコードを動的に適用する方式を実現できる．さらに，ファームウェアイメージの全体を適用する方式に加えてファームウェアイメージの差分を適用する方式にも対応しているため，本研究の整理した方式を網羅できる．以上の理由から，本研究では Elixir と Nerves を用いて開発した IoT アプリケーションを対象に，開発者によるコードの変更をデバイスに対して動的に適用する方式を実装した．

プログラミング言語 Elixir の実行基盤となる仮想機械である Erlang VM は，実行時にオブジェクトコードを動的に置き換えることができる機能を提供している [25]．コードの置き換えは，Erlang VM 上で動作するノードと呼ばれるプロセスを通じて行われる．ノードは遠隔手続き呼び出し（RPC: Remote Procedure Call）に対応しており，遠隔のホスト上で動作するノード上に対して RPC を実行することで，そのホスト上の Erlang VM への操作を実行できる．その機能を用いて，本実装では IoT デバイス上で動作するアプリケーションの動作を変更するためのコード

の適用を実現する．具体的には，RPC を通じて Elixir から Erlang の `code:load_binary/3` 関数をモジュール名，ファイル名，オブジェクトコードのバイナリを引数として呼び出すことで，Erlang VM 上にロードされたコードを置き換える．

本実装では，IoT デバイス上で動作する Erlang VM 内で動作するノードへ開発者の用いるホストから接続し，上述の RPC を通じて Erlang VM 上で動作するコードの置き換えを行う．Erlang VM 上で動作するオブジェクトコードの置き換えが起こった際，古いコードには `old`，新しいコードには `current` というラベルが付され区別される．その状況で，再度コードの置き換えが起こると，`old` のコードは破棄され `current` のコードが `old` となるとともに，破棄されたコードを参照しているプロセスは強制的に実行終了される [25]．本実装では，開発者によるコードの変更が確実に適用されるよう，連続して 2 回の適用を行っている．また，本報告の執筆時点においては，ファイルの更新時刻や内容を考慮しない実装となっているため，開発者が開発の対象とするファイルすべて（具体的には，開発者が直接の開発対象とする `lib/` ディレクトリ以下に置かれる Elixir のコードを含むファイル）を更新対象とする実装になっている．そのため，本実装によるコードの適用後には，開発対象となるコードによって生成された Erlang VM 上のプロセスの再起動が発生する．

なお，実装については GitHub 上のリポジトリ[*3]，および，Elixir のライブラリリポジトリである Hex[*4]でオープンソースソフトウェア（OSS）として公開している．

## 4. 実験と評価

本章では，提案方式について実験に基づき既存方式と比較検討することで評価する．はじめに，4.1 節で実験の対象と方法について述べる．次に，4.2 節で既存方式と提案方式について実験した結果を示す．最後に，4.3 で評価結果についての発展的な議論を述べる．

### 4.1 対象と方法

本章における実験は，2 章で検討した既存方式と 3 章で検討した提案方式とを対象に行う．それぞれ以下の通りであり，(3) が提案方式である．

(1) ファームウェアイメージの全体を適用する方式
(2) ファームウェアイメージの差分を適用する方式
(3) アプリケーションコードを動的に適用する方式

(1) および (2) については，IoT アプリケーションの実装に用いた Nerves が提供する機能を用いてコードの変更を適用する．(1) については `mix firmware` コマンドを，(2) については `mix firmware.patch` コマンドを用いてファームウェアイメージを生成し，`mix update` コマンドを用いてデバイスへの配備を行う．(3) については，3.2 節で述べた実装を用いる．

[*3] https://github.com/kentaro/mix_tasks_upload_hotswap
[*4] https://hex.pm/packages/mix_tasks_upload_hotswap

**表 2** 方式 (1) および (2) における更新のフェーズと計測方法
**Table 2** Update phases of the method (1) and (2) and the measurement methods.

| Phase | Measurement Methods |
|---|---|
| 1. Firmware Build (full/patch) | `time` command |
| 2. Firmware Deployment | `time` command |
| 3. Device Reboot | `ping` command |
| 4. Application Reboot | `ncat` command |

それぞれの方式を比較するための指標として，開発者によるコードの変更をデバイスに適用した上で，IoT アプリケーションが変更を取り込んだ状態で動作するまでに要する時間を用いる．更新対象の IoT アプリケーションとして，TCP ソケット経由でクライアントから受け取った文字列をそのまま返却するサーバ（いわゆる Echo サーバ）を実装し，IoT デバイス上で動作させる．コードを更新した際，(1) および (2) についてはデバイスの再起動に加えて IoT アプリケーションの起動が発生し，その間は IoT アプリケーションはもとよりデバイスへの通信も遮断される．，一方で，(3) については，前述の通り IoT アプリケーションが動作する Erlang VM 上のプロセスの再起動が発生するが，デバイスの再起動は発生しないためデバイスへの通信に関しては影響がない．しかしその場合でも，IoT アプリケーションが TCP 接続を要する実装である場合，更新の間は対象アプリケーションとの通信が遮断される．そのため，コードの変更を適用した後に再び IoT アプリケーションと TCP 通信が確立することをもって，いずれの方式においても動作が再開したとみなすことが，評価の条件をそろえるという観点で妥当である．そのため，IoT アプリケーションのレベルで通信するオーバーヘッドの少ない機能として Echo サーバを実装し，実験に用いる．

評価指標とした，開発者によるコードの変更をデバイスに適用した上で，IoT アプリケーションが変更を取り込んだ状態で動作するまでに要する時間は，方式 (1) および (2) については**表 2** の通り，方式 (3) については**表 3** の通り内訳を分解できる．また，それぞれのフェーズにかかる時間を計測する方法についても記載した．また，実験に用いた環境は，開発用のホストおよび IoT デバイスについて，それぞれ**表 4** の通りである．

### 4.2 実験結果

4.1 節で述べた対象と方法に基づき，既存方式と提案方式について，開発者によるコードの変更がデバイスに適用され，IoT アプリケーションが変更を取り込んだ状態で動作するまでに要する時間を計測した．その結果は，**表 5** の

表 3　方式（3）における更新のフェーズと計測方法
**Table 3**　Update phases of the method (2) and the measurement methods.

| Phase | Measurement Methods |
|---|---|
| 1. Code Deployment | `time` command |
| 2. Application Reboot | `ncat` command |

表 4　実験環境
**Table 4**　Experiment Environment

| | Item | Specification |
|---|---|---|
| Dev Host | Model | MacBook Pro 13-inch, 2018 |
| | CPU | 2.7 GHz Quad-Core Intel Core i7 |
| | Memory | 16 GB 2133 MHz LPDDR3 |
| IoT Device | Model | Raspberry Pi 3 Model B |
| | Network | Wired LAN |
| | Platform | Nerves 1.7.2 |

通りである．また，各方式における更新対象となったファイルサイズを**表 6** に示した．

開発者によるコードの変更を IoT デバイスに適用し，IoT アプリケーションが変更を取り込んだ状態で動作するまでに要した時間は，方式（1）では合計で 66.88 秒，方式（2）では 70.63 秒であったのに対し，提案方式である（3）では 3.30 秒であり，提案方式が既存方式より短かった．方式（1）および（2）はファームウェアイメージの生成，デバイスへの適用，IoT デバイスの再起動のそれぞれにおいて時間を要しており，提案方式はそれらの時間を要する処理を必要としないため優位であることが確かめられた．方式（2）は更新対象のファイルサイズが（1）の 7 分の 1 であるにも関わらず，ファームウェアイメージの差分の生成および適用に方式（1）よりも時間を要している．（1）と違って差分を生成すること，および，差分に基づく適用を実行することにオーバーヘッドが存在するからであると考えられる．また，方式（3）に関する Application Reboot については，計測できなかった．更新対象の Echo サーバは更新の実行時に新しいコードを読み込んで再起動するが，本実験によっては起動中の通信が遮断された状態を捉えられない速度で起動が完了したためである．

なお，実験のプロトコルおよび結果に関する詳細は GitHub 上のリポジトリ[*5]で公開している．

### 4.3　議論

4.2 節で検討した実験結果により，提案方式が開発者によるコードの変更を適用する速度を向上させることで，IoT アプリケーションの開発効率の向上が見込めることを確認できた．一方で，提案方式の実装については議論の余地が残されている．実験に用いた提案方式の実装は，変更対象のコードの依存関係を考慮することなくコードの動的な置き換えを行うため，更新対象のコード間に複雑な依存関係がある場合は，不整合を起こす可能性があると考えられる．また，本実装では IoT アプリケーションが依存する外部モジュールの追加・更新・削除には対応していないため，依存モジュールに関するコードの変更は適用されない．ただし，不整合が起きたり依存モジュールに関する変更があったりした場合でも，ファームウェアイメージを更新することで容易に対応できる．また，プロトタイピング時や開発時のように小規模な変更を頻繁に適用する状況において，提案方式によって開発効率を向上できることは実験結果から確かであるため，提案方式の有効性を損なう問題ではないと考える．

提案方式の一般性について確認する．提案方式は，本研究で用いたものとは異なる言語によっても実装可能であると考えられる．本研究は IoT アプリケーションの開発効率の向上を目的としているため，プロトタイピング時や開発時における利便のために，相対的にスペックの高い開発プラットフォームを利用可能であることが前提である．そのため，ネイティブコードを直接実行する場合にくらべ相対的にリソースを多く要求する Erlang VM のような仮想機械による実行環境を利用可能である．よって，同様に他の仮想機械を前提とした実行基盤を持つ言語を用いても，提案方式について実装が可能であると考えられる．また，Erlang VM を用いる言語自体の IoT デバイスにおける利用可能性についても，プロトタイピング時や開発時にとどまらない適用が可能である見込みがある．Nerves の開発元では実運用を前提とした開発を行っており，実際に利用事例も報告されている [26]．また，[27] は AtomVM という IoT デバイス上で動作することに特化した Erlang VM を開発しており，[23] が Middle-end と分類する ESP8266 の後継である ESP32 上で動作する．よって，今後，提案方式が配備後のデバイスに対しても適用できる可能性もあると期待できる．

## 5.　おわりに

本研究は，IoT アプリケーションの開発効率の向上という観点から，開発者によるコードの変更をデバイスへ適用する方式を提案し実装した．先行研究に基づき方式を整理した上で，提案方式と既存方式とを実験により比較検討した．そのの結果として，開発者によるコードの変更を IoT デバイスに適用し，IoT アプリケーションが変更を取り込んだ状態で動作するまでに要した時間について，提案方式は 95%の改善を実現できた．IoT アプリケーションのプロトタイピング時や開発時においては小規模な変更を頻繁に適用することから，提案方式が実現した改善結果は IoT アプリケーションの開発効率の向上に大きく寄与するものと考える．このことから，IoT デバイスの開発プラット

[*5] `https://github.com/kentaro/ipsj-sigse207-paper-experiments`

表 5　各方式によるコードの更新に要した時間の比較（単位：秒）
**Table 5** Time comparison of code application between the existing methods and the proposed methods (in sec.).

| Method | Firmware Build | Firmware Deployment | Code Deployment | Device Reboot | Application Reboot | Total | Ratio* |
|---|---|---|---|---|---|---|---|
| (1) Firmware Update (full) | 29.45 | 14.49 | — | 22.51 | 0.63 | 66.88 | 100 |
| (2) Firmware Update (patch) | 30.09 | 17.46 | — | 22.54 | 0.54 | 70.63 | 105 |
| (3) Proposed Method | — | — | 3.30 | — | N/A | **3.30** | **5** |

* Ratio column shows the ratios when the total time of the method (1) is set to 100.

表 6　各方式における更新対象ファイルのサイズ
**Table 6** Size of the target files to deploy.

| Method | Size |
|---|---|
| (1) Firmware Update (full) | 42MB |
| (2) Firmware Update (patch) | 6MB |
| (3) Proposed Method | 12KB |

フォームにとって，提案方式が示した IoT アプリケーションを構成するコードの動的な書き換えをサポートすることは，競争優位性となり得ると考えられる．その意味においても，本研究の意義を評価できると考える．

## 参考文献

[1] IHS Markit. The internet of things: a movement, not a market. https://cdn.ihs.com/www/pdf/IoT_ebook.pdf, 2017. Accessed: 2021-2-2.
[2] Development Boards — Espressif Systems. https://www.espressif.com/en/products/devkits. Accessed: 2021-1-24.
[3] Arduino. https://www.arduino.cc/. Accessed: 2021-1-24.
[4] The Raspberry Pi Foundation. Teach, learn, and make with raspberry pi. https://www.raspberrypi.org/. Accessed: 2021-1-24.
[5] Beagleboard.org - community supported open hardware computers for making. https://beagleboard.org/. Accessed: 2021-1-24.
[6] Nvidia embedded systems for next-gen autonomous machines. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/. Accessed: 2021-2-2.
[7] Software — arduino. https://www.arduino.cc/en/software. Accessed: 2021-2-2.
[8] Esp-idf programming guide - esp32 - esp-idf programming guide latest documentation. https://docs.espressif.com/projects/esp-idf/. Accessed: 2021-2-2.
[9] Real-time bidding platform — ssp, dsp solutions - platform.io. https://platform.io/. Accessed: 2021-2-2.
[10] Lee, Edward Ashford and Seshia, Sanjit A. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach, Second Edition*. MIT Press, 2017.
[11] J Bauwens, P Ruckebusch, S Giannoulis, I Moerman, and E D Poorter. Over-the-air software updates in the internet of things: An overview of key principles. *IEEE Commun. Mag.*, Vol. 58, No. 2, pp. 35–41, February 2020.
[12] Sharu Bansal and Dilip Kumar. Iot ecosystem: A survey on devices, gateways, operating systems, middleware and communication. *Int. J. Wireless Inf. Networks*, Vol. 27, No. 3, pp. 340–364, September 2020.
[13] Oracle vm virtualbox. https://www.virtualbox.org/. Accessed: 2021-1-24.
[14] Empowering App Development for Developers — Docker. https://www.docker.com/. Accessed: 2021-1-24.
[15] Peter Ruckebusch, Eli De Poorter, Carolina Fortuna, and Ingrid Moerman. Gitar: Generic extension for internet-of-things architectures enabling dynamic updates of network and application modules. *Ad Hoc Networks*, Vol. 36, pp. 127–151, January 2016.
[16] Peter Ruckebusch, Spilios Giannoulis, Ingrid Moerman, Jeroen Hoebeke, and Eli De Poorter. Modelling the energy consumption for over-the-air software updates in lpwan networks: Sigfox, lora and ieee 802.15.4g. *Internet of Things*, Vol. 3-4, pp. 104–119.
[17] Konstantinos Arakadakis, Pavlos Charalampidis, Antonis Makrogiannakis, and Alexandros Fragkiadakis. Firmware over-the-air programming techniques for iot networks – a survey. September 2020.
[18] The elixir programming language. https://elixir-lang.org/. Accessed: 2021-1-17.
[19] Nerves platform. https://www.nerves-project.org/. Accessed: 2021-1-11.
[20] Narges Yousefnezhad, Avleen Malhi, and Kary Främling. Security in product lifecycle of iot devices: A survey. *Journal of Network and Computer Applications*, Vol. 171, p. 102779, December 2020.
[21] Stephen Brown and Cormac J Sreenan. Software updating in wireless sensor networks: A survey and lacunae. *Journal of Sensor and Actuator Networks*, Vol. 2, No. 4, pp. 717–760, November 2013.
[22] Ondrej Kachman, Marcel Balaz, and Peter Malik. Universal framework for remote firmware updates of low-power devices. *Comput. Commun.*, Vol. 139, pp. 91–102, May 2019.
[23] M O Ojo, S Giordano, G Procissi, and I N Seitanidis. A review of low-end, middle-end, and high-end iot devices. *IEEE Access*, Vol. 6, pp. 70528–70554, 2018.
[24] Erlang programming language. https://www.erlang.org/. Accessed: 2021-1-24.
[25] Chapter 14: Compilation and code loading. https://erlang.org/doc/reference_manual/code_loading.html. Accessed: 2021-1-18.
[26] Case studies. https://www.nerves-project.org/case-studies. Accessed: 2021-1-24.
[27] Davide Bettio. Atomvm. https://github.com/bettio/AtomVM. Accessed: 2021-1-19.