# DIY的IoTシステムに向けた管理SaaSに関する一検討

永野 元基[1] 新井 悠介[1] 藤橋 卓也[1] 渡辺 尚[1] 猿渡 俊介[1]

**概要**：さまざまな現場のエンドユーザがIoT（Internet of Things）を自身の手で導入して現場環境や現場稼働状況を「見える化」することで，生産効率やマーケティング戦略の向上に繋げる動きが日本だけでなく世界にも広がっている．IoTシステム開発をまとめた書籍やWebページの充実にともなって，エンドユーザ自身がIoTシステムを開発するハードルは下がっている．一方で，開発したIoTシステムから継続的にデータを取得して現場の改善に繋げるには，開発したIoTシステムをエンドユーザ自身の手で運用・維持・管理することが重要である．しかしながら，エンドユーザ自身がIoTシステムの運用・維持・管理を実現するシステムを構築するハードルは高い．また，エンドユーザが個々の需要にしたがって開発したIoTシステムに対して既存のPaaS (Platform as a Service)，SaaS (Software as a Service) をそのまま導入してIoTシステムを運用・維持・管理することは困難である．本稿では，エンドユーザが開発したIoTシステムの運用・維持・管理を支援するSaaSを提案する．提案SaaSは，1) IoTデバイス向け機能，2) フロントエンド機能，3) バックエンド機能，4) API Gatewayを組み合わせることで，エンドユーザ自身が構築したIoTシステムを容易に運用できるとともにデバイス稼働状況を視覚的に確認することができる．性能評価から，提案SaaSを通してエンドユーザがIoTシステムを容易に運用・維持・管理できることを示唆した．また，運用・維持・管理を支援する従来のシステムと比較して，提案SaaSは低コストでIoTシステムに含まれるデバイス稼働状況を管理できることを明らかにした．

## 1. はじめに

多様な現場におけるIoTの活用 [1–4] が成功するには，現場で働く人々の現状に対する問題意識が自分の手によるIoTシステムの構築と運用，すなわちDevOps [5] に繋がることが非常に重要である．現在のIoTシステムにおいて，システム構築，すなわち「Dev」を担う部分は現場のエンドユーザにとって実現することが容易である．例えば，現在のIoTシステムの典型的な構成として，Raspberry PiやArduino等のデバイスに接続されたセンサから取得したデータをSORACOMなどの携帯電話モジュールを利用してAWS (Amazon Web Services) 等のクラウドサービスにデータを蓄積するIoTシステムを考える．センサからデータを取得する方法やクラウドサービスにセンサデータを蓄積する方法はすでに多くの書籍やWebページでまとめられているため，等を参照することで比較的容易に実現することができる．一方で，IoTシステムの運用・維持・管理，すなわち「Ops」を担う部分を現場のエンドユーザが自ら構築するハードルは高い．このハードルの高さは，自ら構築したIoTシステムの運用・維持・管理をする場合，ネットワークの運用方法やネットワーク機器で発生した障害の検出方法に挙げられる，システム開発とは異なった知見が必要となる点に起因する．

また，IoTシステムの運用・維持・管理を支援する既存のPaaSおよびSaaS [6] はエンドユーザ自身が開発したIoTシステムに対してそのまま導入することはできない．例えば，IoTデバイス向けアプリケーション開発を支援するPaaSであるAWS IoT [7] やAzure IoT [8] はエンドユーザによるIoTシステム開発は支援できるものの，開発したIoTシステムを運用・維持・管理するための機能を有していない．同様に，システムを監視するための機能を提供するSaaSであるZabbix [9] やmackerel [10] は特定の用途に限られたIoTシステムの運用・維持・管理は支援できるものの，エンドユーザがそれぞれの需要に応じて開発した多様なIoTシステムに対してそのまま導入することは困難である．

本稿では，エンドユーザが構築したIoTシステムの運用・維持・管理を支援する新たなSaaSを提案する．本SaaSによって運用・維持・管理に関する知識を持たないエンドユーザが，マウスクリック等の簡潔な操作で自ら構築したIoTシステムの運用管理を容易に実現できる．提案SaaSは 1) IoTデバイス向け機能，2) フロントエンド機能，3) バックエンド機能，4) API Gatewayから構成される．IoT

[1] 大阪大学大学院情報科学研究科

デバイス向け機能は，IoT デバイスへの介入，IoT デバイスからデータを受信するサーバへの介入を通して，監視対象の IoT システムに属する各 IoT デバイスからの定期生存報告機能を実現する．具体的には，生存報告 Webhook，motch SDK，motch daemon の 3 要素を実装している．フロントエンド機能は，サーバに登録済である IoT デバイスの編集やデバイス生存状況の可視化を Web ブラウザを介して実現する．バックエンド機能はサーバレスアーキテクチャを採用することで IoT デバイス情報，利用者情報，IoT デバイス生死情報のやり取りをスケールアウトが容易な方法で実現する．API Gateway は，API やリアルタイム更新のためのインタフェースを提供するとともに，バックエンド機能と IoT デバイス向け機能や，バックエンド機能とフロントエンド機能をつなぐ役割を担う．提案手法による IoT システムの運用・維持・管理の容易化を示す定性評価として，提案 SaaS を利用した複数のユースケースを示した．ユースケースから，エンドユーザ自身が IoT デバイスに対して，1 行のコマンドを入力して motch SDK を導入あるいは motch daemon を導入することで IoT デバイスからサーバに対する定期的な生存報告が可能になること，Web ブラウザを通して IoT デバイスの稼働状況を視覚的に確認できることが分かった．また，IoT システムに属する IoT デバイス数に応じた既存手法と提案 SaaS のコストを定量評価した結果，提案手法はコストを低減できることを明らかにした．

本稿の構成は以下の通りである．第 2 節では IoT システムの運用・維持・管理に求められる要件と，それらを満たせない既存手法に存在する課題について述べる．第 3 節では IoT システムの運用・維持・管理に求められる要件を満たす，我々が実装した提案手法である motch の機能と実装について述べる．第 4 節では提案手法 motch の評価として，ユースケースによる評価，サポートできる IoT デバイスについての評価，システムの運用におけるスケーラビリティと運用コストの評価について述べる．

## 2. IoT システムにおける運用・維持・管理の課題

### 2.1 運用・維持・管理に求められる要件

エンドユーザが開発した IoT システムを運用・維持・管理するためには 1) 導入が容易であること，2) 多種多様な IoT デバイスに対応できること，3) 管理が容易であること，4) スペック制約が緩いことの 4 要件を満たす必要がある．「1) 導入が容易であること」はエンドユーザが運用・維持・管理に関する知識を必要とせず，わずかな操作で IoT システムを管理することができることを表している．「2) 多種多様な IoT デバイスに対応できること」は，IoT の難しいところは応用によって求められる機能要件が異なり，機能要件によって使用するデバイスの種類が異なることに起因して必要となる．「3) 管理が容易であること」は，管理用のコンソールが用意されていること，デバイスの稼働状況がリアルタイムで表示されることを意味している．「4) スペック制約が緩いこと」とは，IoT システムに用いる IoT デバイスのスペックに制約を設けないこと，IoT デバイスに専用のハードウェアを必要としないこと，IoT デバイスに新たな通信手段を用意する必要がないこと，IoT デバイスがインターネット上に公開されている必要がないことを意味する．

### 2.2 既存手法とその課題

IoT システムの運用・維持・管理を目的とした既存技術は，エンドユーザが開発した IoT システムの運用管理に求められる上述の要件を満たすことができない．例えば，クラウドサービスとして IoT デバイス向けアプリケーションの開発を支援する PaaS である AWS IoT や Azure IoT はエンドユーザ自身による IoT サービスの開発を簡易化するものの，開発した IoT サービスの管理は容易に実現できない．上述の PaaS を用いてエンドユーザ自身が IoT システムの管理機能を実現するには，PaaS の機能に対する深い理解，デバイス管理に求められる IoT デバイスからの定期的な生存報告機能の実装，各 IoT デバイスの稼働状況を容易に確認できるインタフェースの実装，それらの機能を支えるバックエンド機能の実装がそれぞれ求められる．ソフトウェアエンジニアリングに関する知識を持たない多くのエンドユーザが自身の手で各機能を実装することは非常に困難である．

監視機能を SaaS として提供する Zabbix や mackerel は IoT デバイスの管理を可能にするものの，IoT システムに求めるスペック制約が厳しいため，IoT システムに対して多様な IoT デバイスを導入できない．Zabbix や mackerel は主に Linux が動作するサーバの監視を目的として作られた SaaS である．したがって，Linux が動作している IoT デバイスには容易に導入できる一方，Linux が動作していない IoT デバイスを監視することは困難である．例えば，mackerel では Linux が動作しない IoT デバイスを監視対象とする方法もあるが，mackerel の API に対する深い理解が求められる．

IoT システムに関して統合プラットフォームを提供する SORACOM [11] や sakura.io [12] はセルラー回線を介して IoT デバイスの接続，データ収集，管理を支援するものの，各 IoT デバイスに対する厳しいスペック制約が求められる．各統合プラットフォームを利用するためには，IoT システム内の各 IoT デバイスに対してセルラー回線に接続可能なボードを追加するとともに，セルラー回線の契約が必要となる．IoT デバイスを対象としたセルラー回線契約は安価であることが多いが，セルラー回線に接続するためのボードは高価である．また，IoT デバイスはそれぞれ異な

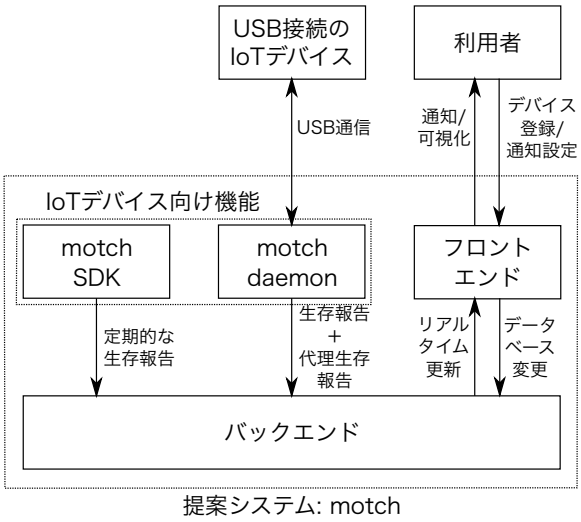


図 1: DIY 的 IoT システムに運用・維持・管理機能を提供する SaaS の全体像

る消費電力制約を持つことに起因して，セルラー回線接続にともなう消費電力増加に対応できない場合もある．

本研究では，IoT システムの運用・維持・管理を容易にすることに主眼を置き，エンドユーザが開発した IoT システムの運用管理に求められる上述の要件を満たす SaaS を開発した．この SaaS を用いることで，エンドユーザは自身による管理システムの構築を行うことなく，IoT システムに管理機能を容易に追加することができる．またエンドユーザによって開発された IoT システムには，開発プラットフォームが異なる多様なスペックの IoT デバイスが使われている可能性がある．この SaaS を用いることで，エンドユーザはデバイスのスペックによらず，全てのデバイスを監視下に置くことができる．さらにこの SaaS は，IoT デバイスのハードウェアの変更を伴わず，ソフトウェアへの小規模な実装により導入することが可能である．これにより，追加コストを発生させることなく IoT システムの管理を導入することができる．

## 3. 提案システム: motch

### 3.1 motch の全体像

本稿では，エンドユーザが自身の手で開発した IoT システムの運用・維持・管理における課題を解決する新たな SaaS を提案する．図 1 に，提案 SaaS の全体像を示す．本 SaaS は IoT デバイス向け機能，フロントエンド機能，バックエンド機能から構成される．

IoT デバイス向け機能は，エンドユーザが開発した IoT システムに属する IoT デバイスに定期的な生存報告機能を実装するために用いる．IoT デバイス向け機能は 1) 生存報告 Webhook，2) motch SDK，3) motch daemon の 3 機能を提供する．「1) 生存報告 Webhook」は，HTTP POST リクエストを送信することにより生存報告を実現する Webhook エンドポイントである．「2) motch SDK」は，IoT デバイス上のプログラムに 2 行追加することで IoT デバイスによる定期的な生存報告機能を実装できるライブラリである．「3) motch daemon」は，Linux が動作する IoT デバイス等に，生存報告機能や直接インターネットに接続できないデバイスの代理生存報告機能を提供するためのソフトウェアである．利用者は IoT デバイス向け機能を，利用者が構築した IoT システムの形態にあわせて選択する必要がある．具体的には，IoT デバイスがネットワークを通じて利用者等が運営するサーバにデータを定期的に送信している場合，生存報告 Webhook を用いる．サーバは IoT デバイスからのデータ受信にフックして Webhook エンドポイントにアクセスすることで IoT デバイスの生存報告を実現する．一方で，IoT デバイスがサーバに対して定期的にデータを送信しない場合は，IoT デバイスに対して motch SDK あるいは motch daemon を導入する．例えば汎用 PC や，Raspberry Pi に代表される Linux が動作する ARM アーキテクチャのシングルボードコンピュータに対しては motch daemon を導入する．mbed や Arduino に代表されるマイクロコンピュータのうち，ネットワークインタフェースを備えたものに対しては motch SDK を導入する．また，ネットワークインタフェースを備えておらず，USB 経由で UART を用いた通信が可能な IoT デバイスに対しては motch daemon を導入したデバイスを接続する．motch daemon が導入されたデバイスを経由してネットワークインタフェースを有しない IoT デバイスの生存報告が可能となる．

フロントエンド機能は Web ブラウザ上からデータベースに登録済の IoT デバイスの編集やデバイス生存状況の可視化を実現する．フロントエンド機能は 1) デバイス編集，2) motch SDK 等のインストール，3) デバイス状況可視化，4) デバイス状況通知の 4 機能を提供する．「1) デバイス編集機能」は，IoT デバイスの登録や編集を行うためのインタフェースと，利用者によって変更された情報をバックエンドに送信する機能を提供する．「2) motch SDK 等のインストール機能」は，motch SDK や motch daemon のダウンロードリンクを表示するためのインタフェースを提供する．「3) デバイス状況可視化機能」は，利用者によって登録された IoT デバイスの稼働状況をバックエンドから取得して表示するインタフェースを提供する．「4) デバイス状況通知機能」は，利用者によって登録された IoT デバイスの稼働状況の変化をバックエンドから取得して通知する機能を提供する．IoT デバイス稼働状況はバックエンドとの WebSocket 通信によりリアルタイムに更新される．

バックエンド機能は利用者や登録されている IoT デバイスの情報，SaaS の動作に必要な情報の保持や，IoT デバイスの死活監視を実現する．バックエンド機能は 1) データベース，2)　デバイスの死活監視，3) デバイス状況変化

通知の 3 機能を提供する．「1) データベース」は，利用者や登録されている IoT デバイスの情報，その他 SaaS の動作に必要な情報を保持する．「2) デバイスの死活監視」は，デバイスから生存報告が来ていることを定期的に確認する機能を提供する．「3) デバイス状況変化通知」は，デバイスの死亡が検知されたら所有するユーザに通知を送信する機能を提供する．

利用者はフロントエンドによって提供される Web 管理画面を操作することにより，デバイスの登録や通知設定などを行うことができる．フロントエンドは利用者の操作を受けて，利用者の意図に沿うようにデータベースの内容を変更するリクエストを，API Gateway を通じてバックエンドに送信する．バックエンドは API Gateway から受け取ったフロントエンドのリクエストを解釈し，利用者の意図する操作をデータベースに対して行う．また，IoT デバイス向け機能より送信された生存報告は API Gateway を通じてバックエンドに渡され，バックエンド内部で生存報告のデータベースへの保存や死活監視プロセスの生成が行われる．死活監視プロセスによって IoT デバイスの死亡が確認されると，フロントエンドの Web 管理画面上にデバイスの死亡通知が表示されたり，利用者によって設定された通知先にデバイスの死亡通知が送信される．デバイスの追加や死亡といった稼働状況の変動は，API Gateway を通じてバックエンドからフロントエンドに即時送信され，Web 管理画面の表示がリアルタイムに更新される．

IoT デバイス向け機能およびフロントエンド機能によるバックエンド機能へのアクセスはデータベースの改変を招く恐れがある．提案 SaaS ではバックエンド機能に認証機能を備えて IoT デバイス情報や利用者設定に対するアクセスや変更がエンドユーザ自身の意図に基づくものであるか判別する．具体的には，AWS の 1 サービスである Cognito の機能を使用して，認証サーバの構築，Web 管理画面へのユーザ認証機能の提供，認証トークンの発行・検証を実現している．Cognito を用いることにより，安全な認証基盤を低コストで容易に実現することができる．また，フロントエンド機能では Web 管理画面上での操作によって利用者設定を変更するとき，操作前にメールアドレスとパスワードを要求して操作者がエンドユーザであることを認証する．具体的には，AWS が提供するフレームワーク AWS Amplify を認証に利用する．Amplify には Cognito による認証基盤を利用するためのインタフェースが機能の一つとして組み込まれており，このインタフェースを利用することで Web アプリケーションに認証機能を容易に実装することができる．同様に，IoT デバイス向け機能では，IoT デバイスによる生存報告の送信元が偽装される可能性がある．生存報告元の IoT デバイスが IoT システム内のデバイスであることを確認するために，あらかじめ motch SDK，motch daemon，生存報告 Webhook が出力する URL に認証情報を含める．認証情報とは，IoT デバイスが生存報告やデバイス登録・変更をするために必要となる，IoT デバイスを所有する利用者のユーザ ID とデータベースに登録された IoT デバイス自身のデバイス ID の組である．各機能のほとんどの通信には HTTPS 通信を用いることで，認証情報の漏洩が起こりにくいデバイス認証を実現する．

### 3.2 IoT デバイス向け機能

**生存報告 Webhook**

生存報告 Webhook では POST リクエストの受信を生存報告とみなす JSON API エンドポイントを提供する．本エンドポイントはフロントエンド機能に対応するバックエンドでデータベースに登録された各 IoT デバイスに対して 1 つ発行する．サーバは各 IoT デバイスから定期的に受信するデータ，あるいは相当するアクションにフックして IoT デバイスに対応するエンドポイントの URL に HTTP POST リクエストを送信することで生存報告を実現する．

**motch SDK**

motch SDK は IoT デバイス上のプログラムに 2 行追加して IoT デバイスによる定期的な生存報告機能を実装できるライブラリである．新たに追加する 2 行のコードは，ライブラリのインクルードに用いる 1 行と IoT デバイスの生死監視を有効化する 1 行から構成される．motch SDK はこれまで Arduino IDE 上で開発する場合を想定して ESP-WROOM-02 [13]，ESP-WROOM-32 [14]，WioLTE [15] に対応している．

motch SDK を用いた定期的な生存報告の実装を行う場合，利用者は Web 管理画面を操作して提案システムにデバイスを登録し，登録完了後表示される画面からライブラリのファイルをダウンロードする．ダウンロードしたライブラリのファイルをユーザが記述したプログラムと同じディレクトリに配置し，ユーザ記述プログラムにライブラリファイルをインクルードする．そして監視の有効化のために，プログラム開始後 1 度だけ実行される部分に motch_enable 関数を加える．このライブラリのインクルードと監視の有効化の 2 行の追加のみで，IoT デバイスに定期的な生存報告を実装することができる．

提案システムによって生成されたライブラリを用いて，Arduino プログラムに生存報告を実装した様子を Listing1 に示す．

Listing 1: Arduino プログラムに生存報告を実装した様子

```
#include ...
/* Array of #include and #define */
#include "motch.h"

/* Declarations of global variables or
  functions */
```

```
void setup() {
    ⋮
    motch_enable();
}

void loop() {
    ⋮
}
```

motch SDK を用いて ESP-WROOM-02 等のマイコンに生存報告を実装する場合，利用者は Web 管理画面を操作してダウンロードしたライブラリのファイルを Arduino プログラムと同じディレクトリに配置し，Arduino プログラムにダウンロードしたライブラリファイル (例えば，motch.h) を，#include 命令を用いてインクルードする (例えば，#include ”motch.h”) ．そして，setup 関数 [16] の中に motch_enable 関数呼び出しを書き加える．このライブラリのインクルードと監視の有効化の 2 行の追加のみで，ESP-WROOM-02 のプログラムに定期的な生存報告を実装することができる．

**マクロフックによる生存報告機能の簡易追加**

マクロフックは，Arduino 言語を用いたプログラミングにおける loop 関数の実行開始タイミングに任意の処理を行うために，motch SDK の実装に用いた新たな方法である．ESP-WROOM-02 や WioLTE 向けの motch SDK を実装する際，これらのマイコンのタイマの仕様上 loop 関数の開始時に生存報告を行うべきタイミングかどうかを確認する必要があった．プログラムへの 2 行の追加のみで生存報告の実装を可能とするためには，利用者の手で loop 関数を改変することなく実装することができる必要がある．そのため，ライブラリ側から loop 関数の実行開始を検知して生存報告処理を実行できる必要があった．しかし，Arduino 言語を用いたプログラミングにおいては，setup 関数や loop 関数の実行開始タイミングをイベント等により検知することができない．そのため，setup 関数や loop 関数の実行開始タイミングに任意の処理を挟むことが不可能であった．この問題に対処するための方法がマクロフックである．具体的には，利用者がプログラム中で定義した loop 関数の名前を別のものに置き換え (例えば，motch_loop) ，私がライブラリ内で再定義した loop 関数の中で，フラグをチェックして生存報告を行う処理と利用者定義 loop 関数 (例えば，motch_loop 関数) の処理を続けて行う．このようにすることで，利用者に 2 行で生存報告を実装することができるインタフェースを提供している．Listing2 に示す Arduino プログラムが motch SDK の適用によって変換されたものを Listing3 に示す．

Listing 2: サンプルプログラム

```
#include "motch.h"
void setup() {
  Serial.begin(115200);
  motch_enable();
}
void loop() {
  Serial.println("Hello");
  delay(1000);
}
```

Listing 3: マクロフックにより変換されたプログラム

```
// HTTPS のためのライブラリのインクルードPOST
boolean post_flag = false;
void motch_loop();
void motch_enable() {
  // 報告周期・タイマを設定
}
void set_post_flag() {
  // タイマによって定期的にコールされる
  post_flag = true;
}
void post() {
  // 生存報告
}

void loop() {
  if (post_flag == true) {
    post();
  }
  motch_loop();
}
#define loop motch_loop

void setup() {
  Serial.begin(115200);
  motch_enable();
}

void motch_loop() {
  Serial.println("Hello");
  delay(1000);
}
```

**motch daemon**

motch daemon は Linux が動作する IoT デバイスや PC などにインストールして定期的に自身の生存を報告する機能と他のデバイスの生存を自身が代理で報告する機能を導入できるプログラムである．motch daemon の構成図を図 2 に示す．motch daemon は，USB ポートを持つ Debian 系 Linux が動作した Raspberry Pi などのシングルボードコンピュータや，PC 向けに実装している．motch daemon は Web 管理画面に表示される 1 行のコマンドを実行するだけで IoT デバイスや PC にインストールできる．また，イ

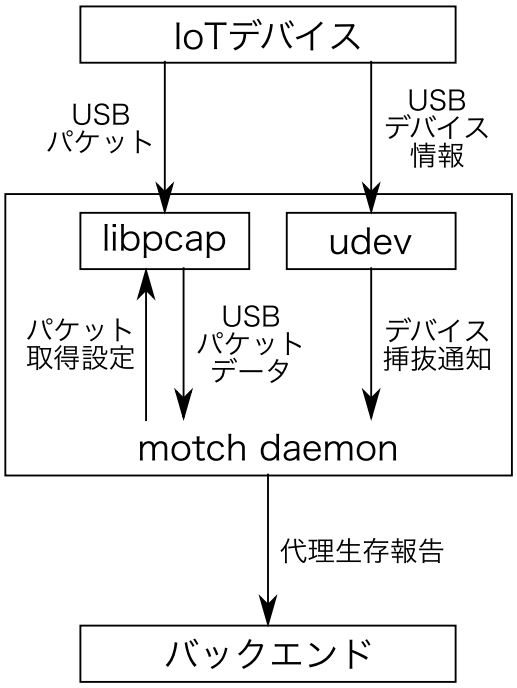

図 2: motch daemon の構成図

ンストールが終了すると自動的に systemd service [17] として IoT デバイスや PC に登録される．より具体的には，利用者は Web 管理画面を操作して提案システムに監視対象となるデバイスを登録する．監視対象となるデバイスの登録を完了した後，利用者は Web 管理画面上に表示される 1 行のコマンドを取得して監視対象となるデバイスのコンソール上でコマンドを実行する．この motch daemon のインストールコマンドを，Debian 系 Linux がインストールされたシングルボードコンピュータや PC 上の Bourne Shell 互換シェルで実行すると，以下のような動作をする．まず，motch daemon やその実行に必要な認証情報と依存関係のあるライブラリのダウンロードとインストールを行う．そして，motch daemon をシングルボードコンピュータや PC 起動時に，自動で実行開始させるための systemd service へ常時起動設定の登録を行う．利用者は motch daemon を IoT デバイスや PC にインストールすると，motch daemon の存在を意識することなく，監視対象のデバイス自身による生存報告や，監視対象デバイスと USB で接続された IoT デバイスの代理生存報告が可能となる．

motch daemon が提供する生存報告機能および代理生存報告機能は Python 3 によって実装されている．生存報告機能と代理生存報告機能は全く別のプロセスとして実行される．代理生存報告機能のうち USB デバイス検出機能と USB パケットを用いた生存報告機能は同一プロセス内の別スレッドで実行され，パケット検出機能はこのプロセスの子プロセスとして実行される．

生存報告機能では Python の標準ライブラリである threading ライブラリを用いて生存報告スレッドを生成する．このとき，生存を報告する周期を一定に保つために，現在の時刻と実行開始時刻を元にして直近の生存報告時刻を決定する．

代理生存報告機能は，主に USB デバイス検出機能，パケット検出機能，USB パケットを用いた生存報告機能の 3 機能から構成される．USB デバイス検出機能は，Linux が動作する IoT デバイスや PC に対して USB を通じて異なる IoT デバイスが接続されたとき，提案システム内のデータベースに USB 接続された IoT デバイスを登録する．具体的には，Linux の機能の 1 つである udev を用いて USB デバイスの挿抜を検出し，監視対象デバイスに挿入した USB デバイスのベンダ ID，プロダクト ID，シリアルナンバ，USB 上でのバス ID とデバイス ID を取得する．提案システムでは USB デバイスの情報を取得した後，バックエンドから登録済のデバイス一覧を取得して，先ほど取得したベンダ ID，プロダクト ID，シリアルナンバから USB デバイスが登録済であるかどうかを照合する．もし，登録済のデバイス一覧と一致するデバイスが存在しない場合は，USB 接続されたデバイスは新たなデバイスであると認識して，一意のデバイス ID を生成し，バックエンドのデータベース上にベンダ ID，プロダクト ID，シリアルナンバとともに登録する．そしてデータベース上のデバイス ID と USB 上のバス ID およびデバイス ID との対応表を作成する．本対応表はパケット検出機能に利用する．

パケット検出機能は，C 言語で記述されたライブラリである libpcap [18] を用いて，所望の USB バス ID，デバイス ID のパケットのみを選り分けて取得する機能である．パケット検出機能は USB デバイス検出機能と USB パケットを用いた生存報告機能の子プロセスとして実行される．これは同一プロセスで 3 つの機能をそれぞれ別のスレッドにて実行した場合に，I/O のブロッキングと思われる現象によりパケット検出機能以外のスレッドの動作が停止するという問題が発生するためである．

USB パケットを用いた生存報告機能は，パケット検出機能により検出された USB デバイスごとのパケットを定期的に確認する．ある USB デバイスからパケットが 1 つ以上到着していれば，USB デバイス検出機能が保持する対応表を用いて対応するデバイス ID を取得するとともに，そのデバイスからの生存報告として代理で生存報告を行う．生存報告を終えると，到着したパケットをクリアして次の生存報告タイミングまで待機する．

### 3.3 フロントエンド機能

#### デバイス編集機能

デバイス編集機能は，デバイスの新規追加，デバイス情報の編集，デバイスから送信される数値データを元にした通知の設定，登録デバイスの削除の 4 つの要素で構成される．デバイスの追加は，利用者が追加しようとしているデ


図 3: デバイスの追加

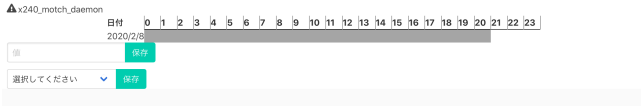


図 4: デバイス詳細画面

バイスの名前と種類をデータベース上に登録する機能を提供する．デバイスの追加は，デバイス追加画面より行うことができる．デバイスの追加を行っている様子を図 3 に示す．利用者はデバイスの名前とデバイスの種類を設定する．「デバイスを追加する」ボタンをクリックすると，デバイスがデータベースに新規登録される．「キャンセル」ボタンをクリックすると，デバイスは登録されないまま終了する．デバイス情報の編集は，データベースに登録済であるデバイスの名前と種類を編集して，データベース上の情報を修正する機能を提供する．デバイスから送信される数値データを元にした通知の設定は，デバイスによる生存報告時に受信した数値データに対して条件を設定してデータベース上に登録する．具体的には，設定可能な条件として，受信した数値データががある閾値を超えた時に通知する，ある閾値を下回った時に通知する，ある値と等しくなった時に通知する，ある値と等しくない場合に通知する，前の生存報告時から変化すれば通知する，という通知条件を設定できる．デバイスの削除は，登録済であるデバイスの情報をデータベース上から削除する機能を提供する．デバイス情報の編集，デバイスから送信される数値データを元にした通知の設定はデバイス詳細画面より行うことができる．デバイスの削除はデバイス詳細画面とデバイス一覧ページより行うことができる．デバイス詳細画面を図 4 に示す．

**motch SDK 等のインストール機能**

motch SDK 等のインストール機能は，motch SDK のダウンロードリンクや motch daemon のインストールコマンドを表示する．motch SDK のダウンロードリンクは，クリックするとダウンロードページに移動し，motch SDK のダウンロードが開始されるリンクである．motch SDK 等のインストール機能によって構成される SDK 等のダウンロード画面は，デバイス一覧画面で SDK インストールボタンをクリックすると開くことができる他，デバイスを追加した時に自動で開き，デバイスへのインストールと生存報告の開始を促す．motch SDK のダウンロードリンクや motch daemon のインストールコマンドは，motch SDK


図 5: motch SDK のインストール画面


図 6: motch daemon のインストール画面

等インストール画面に表示される．motch SDK 等インストール画面が表示されている様子を図 5, 6 に示す．図 5 は motch SDK のインストールが可能なインストール画面であり，図 6 は motch daemon のインストールが可能なインストール画面である．

**デバイス稼働状況可視化**

デバイス状況可視化機能は，デバイスの一覧表示，現在のデバイス稼働状況の表示，画面を表示した日のデバイス稼働状況の履歴の表示，デバイスの接続関係の表示の 4 つの要素で構成される．デバイスの一覧表示は，利用者が登録しているデバイスをデバイス一覧画面に表示する．また，各デバイスについて，デバイス詳細画面へのリンクや現在のデバイス稼働状況，表示した日のデバイスの稼働状況の履歴，motch SDK 等のインストール画面を開くボタンやデバイスを削除するボタンを表示する．現在のデバイス稼働状況の表示は，デバイスの状況変化の最新の履歴を元にして現在のデバイス稼働状況を表示する．画面を表示した日のデバイス稼働状況の履歴の表示は，過去 24 時間の生存報告を元にして，画面を表示した日のデバイス稼働状況の履歴を表示する．デバイスの接続関係の表示は，利用者が登録しているデバイスのリストから木構造の接続関係を抽出し，デバイス一覧画面上に木構造で表現されたデバイスの接続関係を表示する．デバイスの状況の可視化はデバイス一覧画面とデバイス詳細画面より行うことができる．デバイス一覧画面では利用者が所有する全てのデバイスの稼働状況を見ることができる．デバイス一覧画面に全てのデバイスの稼働状況が表示されている様子を図 7 に示す．デバイス詳細画面では 1 つのデバイスの稼働状況のみを見ることができる．

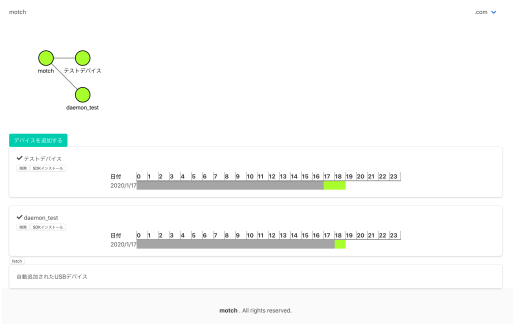

図 7: デバイス一覧画面で全てのデバイスの稼働状況を閲覧している様子

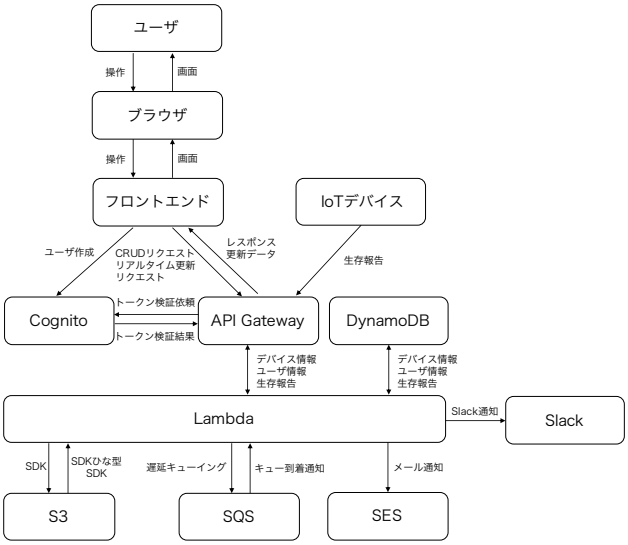


図 8: バックエンドのアーキテクチャ

**デバイス状況変化通知**

デバイスの状況変化の通知機能は，デバイスの状況変化をリアルタイム更新機能を用いて取得し，Web 管理画面上に JavaScript の Notification API を用いて通知ポップアップを表示する．Notification API による通知は利用者による明示的な許可がなければ表示されないため，利用者は通知機能を使用するかどうかを選択することができる．デバイスの状況変化の通知機能が通知するイベントは，デバイスからの生存報告が途絶えた，つまりデバイスが死亡したというイベント，デバイスからの生存報告が再開された，つまりデバイスが生き返ったというイベント，motch daemon により USB デバイスがデバイス追加候補として自動追加されたというイベントである．デバイスの状況変化の通知は，現状ではデバイス一覧画面を表示しているときのみ表示される．

### 3.4 バックエンド機能

バックエンドは，IoT デバイスや Web 管理画面からのアクセスを受けて，各デバイスや各利用者の全ての状態を管理するサーバ群である．バックエンドのサーバ群は，サーバレスアーキテクチャによって構築されている．サーバレスアーキテクチャとは，イベント駆動で計算資源を確保するようなリソースを組み合わせたアーキテクチャである．サーバが常に接続を待ち受けて起動している従来のアーキテクチャと異なり，HTTP リクエストが来た，キューにメッセージが到着した等のイベントによってあらゆるコンポーネントが起動され，計算資源を必要な量だけ使って処理し，またあるコンポーネントにイベントと共に処理結果を渡す，という思想のもとに構築される．実際にバックエンドの各要素は，AWS のサービスを用いてコンポーネント化された上で構築されている．これらは別のコンポーネントが発生するイベントをトリガに動作することができ，そのリクエストの規模に応じて使用する計算資源を自動で柔軟に割り当てられている．

バックエンドの概念図を図 8 に示す．ユーザはブラウザ上に表示された画面を操作して，ユーザ登録やデバイス情報の作成，更新，閲覧，削除，リアルタイム更新の開始を行う．ブラウザはユーザの操作をフロントエンドに伝達する．また，フロントエンドによって構成された画面を描画する．フロントエンドはユーザの操作をバックエンドに対する操作に変換してバックエンドにリクエストを送信する．例えば，ユーザが新規登録したいときは Cognito にアクセスしてユーザ作成リクエストを送信する．デバイス情報を作成，更新，閲覧，削除したいときやリアルタイム更新機能を有効にしたいときは API Gateway にリクエストを送信する．また，バックエンドからのレスポンスや更新データの受信を元に画面を構成する．IoT デバイスは生存報告のためのリクエストを送信する．API Gateway はフロントエンドや IoT デバイスからのリクエストを受け付ける．リクエストを受け付けた際は，リクエストの送信元を認証するために，リクエストに付与されているトークンを Cognito に送信して検証を依頼する．Cognito によって送信元が確認されたら，リクエストを解釈して Lambda を呼び出す．Lambda からレスポンスが返ってきたら，レスポンスの内容をアクセス元に送信する．またリアルタイム更新が有効な場合，Lambda からデータ更新イベントが発生したときに更新データをアクセス元に送信する．Cognito はユーザ情報を保存している．また，ユーザ情報を元にトークンの検証を行う．DynamoDB はデバイスや生存報告の情報を保存するデータベースである．S3 は SDK や SDK の雛形を保存するストレージである．SQS は死亡検知プロセスに必要なキューを提供する．SES はユーザへのデバイス死亡通知をメールで行うために用いるメール送信サービスである．Lambda はデータベース変更ロジック，デバイス死亡検出ロジック，デバイス死亡通知ロジック，SDK 生成ロジックの 4 つのロジックを提供する．データベース変更ロジックは，API Gateway からユーザの操作リクエストを受け取ったときに実行される．ユーザによるデバイス情報へのアクセスや IoT デバイスによる生存報告

の作成に対して，DynamoDB に書き込みや読み出しを行う．デバイス死亡検出ロジックは，DynamoDB の生存報告テーブルが更新されたときに実行される．生存報告が作成されたら SQS に遅延付きのメッセージを送信する．設定した遅延時間が経過したのち SQS にメッセージが到着すると，DynamoDB から取り出したデバイスの最新の生存報告が行われた時刻を現在時刻と比較する．その際，デバイスごとに設定されたタイムアウトよりも長い時間が経過していれば，デバイスが死亡したと判断してデバイス死亡通知ロジックを実行する．デバイス死亡通知ロジックは，デバイスの死亡を検知したときに実行される．死亡したデバイスの所有者に，メールや Slack を用いてデバイスの死亡を通知する．メールの送信には SES を用いる．Slack へは Lambda が直接 HTTPS リクエストを送信して通知する．SDK 生成ロジックは，DynamoDB のデバイス情報テーブルにエントリが新規作成されたときに実行される．S3 に保存された SDK の雛形を読み出して，各デバイスごとの認証情報を付与したものを，デバイス固有の SDK として S3 に保存する．

**データベース**

データベースは，デバイス情報やデバイスからの生存報告，各利用者の設定情報を保存している．より具体的には，AWS の中のサービスの 1 つである DynamoDB による NoSQL データベースに Device テーブル，Report テーブル，User テーブル，Connection テーブルを作成している．Device テーブルではデバイスの ID や名前，種類，所有している利用者のユーザ ID，デバイスが作成された Unix 時間，接続関係上親となるデバイスの ID，motch daemon によって自動的に作成され利用者の承認を待っている状態か否か，USB デバイスのベンダ ID，プロダクト ID，シリアルナンバ，デバイスから送信される数値データに対してどのような条件で通知を行うか，デバイスから送信される数値データに関する通知の条件に用いる閾値を保存する．Report テーブルにはデバイスからの生存報告と，デバイスの状況変化の検出機能により検出されたデバイスの死亡が記録される．このため Report テーブルでは生存報告を行ったデバイスの ID，生存報告された Unix 時間，生存報告を行ったデバイスを所有する利用者のユーザ ID，デバイスの稼働状況，前回の生存報告から稼働状況が変化したかどうか，当該生存報告が有効である時間を保存する．User テーブルではユーザ ID，利用者の名前，通知設定を保存する．Connection テーブルでは WebSocket のコネクション ID とユーザ ID の対応を保存している．

**デバイス状況変化検出**

デバイスの状況変化の検出は，IoT デバイスからの生存報告が来ているかどうかを定期的に確認し，生存報告がなければ死亡した旨をデータベースに書き込み，デバイスの状況変化の通知機能に通知メッセージの送信を依頼する．より具体的には，これらの機能を AWS のサービスのうち DynamoDB，SQS，Lambda を用いて実現している．DynamoDB はデバイスが死亡したことを Report テーブルに書き込むために，SQS はイベント駆動で定期的に生存確認プロセスを起動するために，Lambda は生存報告が来たときに生存確認プロセス起動リクエストを SQS に送信し，生存確認プロセスを実行するために用いている．デバイスからの生存報告が来ると Lambda の計算資源が起動され，SQS に遅延付きメッセージを送信する．この遅延は生存報告に含められた timeout パラメータと同じだけの時間とする．指定された遅延時間ののち SQS にメッセージが到着すると，先述のものとは違う Lambda の計算資源が起動される．この Lambda は DynamoDB 上の Report テーブルにアクセスして，生存報告を送ってきたデバイスの最新の生存報告を参照する．最新の生存報告が更新されていればデバイスが生存しているとみなし，何も行わない．最新の生存報告が更新されておらず，timeout パラメータに設定された時間よりも前のものであればそのデバイスが死亡したとみなし，DynamoDB の Report テーブルに稼働状況が死亡に変化した旨を書き込み，デバイスの状況変化の通知機能にデバイス死亡通知メッセージの送信を依頼する．

**デバイス状況変化通知**

デバイスの状況変化の通知は，利用者が設定した E メールアドレスや Slack チャンネルに対して，デバイスの稼働状況の変化やデバイスから送信された値に関する通知を行う．より具体的には，AWS のサービスのうち Lambda と SES を用いてこれらの機能を実現している．Lambda はデバイスの状況変化の通知リクエストをトリガに起動し，Slack チャンネルへ投稿するための Webhook URL に HTTPS POST リクエストを送信する Lambda を起動したり，指定された E メールアドレス宛に E メールを送信する SES を起動する．利用者は自身が設定した Slack チャンネルや E メールアドレスに受信したメッセージにより，デバイスの状況変化を知ることができる．

### 3.5 API Gateway

AWS のサービスの 1 つである API Gateway は，提案システムにおいてバックエンドとその他の要素を繋ぐ役割を持つ．API Gateway は従来の常時起動サーバ型のアーキテクチャで用いられることは少ないが，サーバレスアーキテクチャでは，その親和性の高さから API サーバを構築する際に頻繁に用いられる．それには 2 つの理由がある．1 つはサーバレスアーキテクチャが苦手とする，接続を待ち受けるという動作を代替することができるからである．もう 1 つはアクセスが発生した際にイベントを発生し，Lambda を起動することができるからである．提案システムでは，IoT デバイスからの生存報告，フロントエンドからのデバイス情報へのアクセスとリアルタイム更新を実現するため

に，それらの REST API へのアクセスを待ち受ける部分に API Gateway が用いられている．

**IoT デバイスとバックエンドを繋ぐ API Gateway**

IoT デバイスとバックエンドを繋ぐ API Gateway は，生存報告の送信先とデバイスの作成，登録デバイス一覧の取得のための REST API として振る舞う．API Gateway を生存報告の送信先として用いるのは，motch SDK と motch daemon の生存報告機能である．IoT デバイスは定期的に生存報告を行うために，生存報告のための情報を載せた HTTPS POST リクエストもしくは HTTP POST リクエストを API Gateway に送信する．API Gateway はリクエストを解釈し，それに対応する Lambda を起動する．起動されたデバイスによってデータベース上に生存報告が登録される．また，API Gateway をデバイスの作成と登録デバイス一覧の取得のために用いるのは，motch daemon の USB デバイス自動追加機能である．motch daemon は USB デバイスを挿入した際，それがすでにデータベースに登録されていないかを確認するために API Gateway にアクセスして，デバイス一覧を取得する．デバイス一覧と挿入されたデバイスの情報を照合し，存在していなければ新規デバイスの情報を載せた HTTPS リクエストを API Gateway に送信する．API Gateway はリクエストを解釈し，それに対応する Lambda を起動する．起動された Lambda によってデータベース上に新規デバイスが作成される．

**フロントエンドとバックエンドを繋ぐ API Gateway**

フロントエンドとバックエンドを繋ぐ API Gateway は，REST API と WebSocket エンドポイントの 2 つの振る舞いをする．

まずは、REST API としての API Gateway として動作する場合である．フロントエンドは，データベースに格納されているデバイスや利用者の設定の情報を取得するときに REST API を呼び出す．それに対してバックエンドは，データベースへのクエリを発行して所望のデータを取り出す．より具体的には，REST API がフロントエンドによって呼び出されると，API Gateway がフロントエンドからのリクエストを解釈し，対応する Lambda を起動する．この Lambda はバックエンドで実行され，データベースへのクエリを発行して実行結果を取得し，HTTP レスポンスを生成して返却する．API Gateway は返却された HTTP レスポンスをフロントエンドに送信する．

またフロントエンドは，デバイス情報や利用者設定などのデータベースの内容を変更したい時にも REST API を呼び出す．それに対してバックエンドは，データベースへのクエリを発行してデータベースに変更を書き込む．データベースへのアクセスの流れはデータ取得時と同じであるが，返却されるデータが存在しないので，API Gateway によって返却される HTTP レスポンスが空となる点が異なる．

次に、WebSocket エンドポイントとしての API Gateway として振る舞う場合である．フロントエンドは，定期的に REST API にアクセスすればデータベース内の最新の情報を取得することができる．しかし API Gateway は 1 回の API 呼び出しごとに利用料金が課せられるため，フロントエンドが頻繁に API を呼び出すことは現実的でない．そのために必要なのが，頻繁に API にアクセスしなくてもデータベースの最新情報を取得できる仕組みである．API Gateway の機能の 1 つである WebSocket エンドポイントを利用すれば，リアルタイム更新機能を実現することができる．より具体的には，リアルタイム更新機能は API Gateway と，バックエンド内の DynamoDB，Lambda を用いて実現されている．フロントエンドが API Gateway の WebSocket エンドポイントに接続すると，WebSocket コネクションを開始し，Lambda を起動する．この Lambda はセッション管理を行うためのもので，DynamoDB の Connection テーブルにコネクション ID を書き込む．フロントエンドが WebSocket で接続されている間，バックエンドは DynamoDB へのデータ追加をトリガにして起動する Lambda により，API Gateway に WebSocket 通信に送信したいデータを送信する．フロントエンドは API Gateway から送られてきた，データベースの更新通知を受信して，内部に保持しているデータベースの一部のコピーを更新する．これにより，頻繁にバックエンドへの問い合わせを行うことなく生存状況等を更新することができる．

## 4. 評価

### 4.1 ユースケースによる評価: motch SDK を用いるケース

構築した SaaS の有用性を定性的に確認するため，ユースケースによる評価を行った．ユースケースによる評価を行った IoT デバイス用 SDK は ESP-WROOM-02 [13]，ESP32-WROOM-322 [14]，WioLTE Cat.1 [15] である．いずれの SDK においても手順は同じであるため，本稿では特に ESP-WROOM-02 について取り上げて評価を行う．

ESP-WROOM-02 上で動作するプログラムに生存報告を実装する際の手順を以下に示す．

(1) Web 管理画面を開き，ログインする
(2) デバイス一覧画面を表示する
(3) デバイス追加画面を開き，デバイスを作成する
(4) SDK インストール画面で IoT デバイス用 SDK のダウンロードリンクをクリックする
(5) ダウンロードされた C ヘッダファイルをプログラムと同じディレクトリに配置する
(6) プログラム冒頭で C ヘッダファイルをインクルードする
(7) setup 関数内に motch_enable 関数呼び出しを追記する


図 9: 個人メニュー

(8) プログラムをコンパイルする
(9) ESP-WROOM-02 へ転送する
以上の手順で IoT デバイス用 SDK を導入することにより，定期的な生存報告を開始させることができる．利用者はライブラリのインクルードと監視の有効化に必要な 2 行分のプログラム追加のみで，ESP-WROOM-02 のプログラムに定期的な生存報告を実装することができる．他の IoT デバイス用 SDK である ESP32-WROOM-32，WioLTE Cat.1 についても同様の手順で生存報告の機能を付与することができる．

**ユースケースによる評価: motch daemon を導入するケース**

本ユースケースでは，Arduino UNO に接続した温度センサや湿度センサから得られるデータを元にした農場の環境モニタリングをエンドユーザが実現したいものとする．ここで，エンドユーザはすでに Arduino を通じて USB 経由で PC にセンサから得られるデータを転送して所望のストレージに保存するという IoT システムを構築しているものとする．この IoT システムを運用し，センサデータを継続的に受信するためには，Arduino UNO の死活監視が重要となる．利用者がこの IoT システムに Arduino UNO の死活監視を導入したいと考えたとき，利用者はまず PC で Web 管理画面にアクセスし，ログインする．ログインが完了すると，画面右上に設定画面やデバイス一覧画面にアクセスすることができる個人メニューが出現する (図 9)．このメニューの中の「管理画面」をクリックすると利用者はデバイス一覧画面にアクセスすることができる．デバイス一覧画面上に表示されたデバイス追加ボタンをクリックするとデバイス追加画面が表示される．デバイス追加画面で，利用者はデバイスの名前を入力して，デバイスの種類の選択肢の中から「motch daemon」を選ぶ (図 10)．名前入力と種類選択が終了して「デバイス追加」ボタンをクリックすると，デバイス追加リクエストが送信される．バックエンドのデータベースにデバイス情報が登録されると，デバイス一覧画面上に新規追加された motch daemon が表示されると同時に，IoT デバイスが追加された旨の通知と motch SDK 等インストール画面が表示される (図 11)．PC に Arduino が接続されている場合はそれを一旦取り外した後，利用者は motch SDK 等インストール画面


図 10: デバイスの追加


図 11: motch SDK 等インストール画面

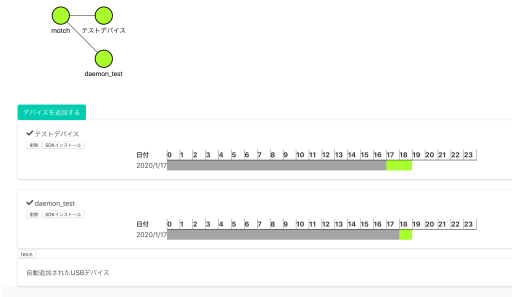


図 12: motch daemon からの生存報告が確認できるデバイス一覧画面

の「motch daemon インストールコマンド」をコピーして，motch daemon をインストールしたいデバイスのコンソール上に貼り付けて実行する．インストールが終了すると，デバイス一覧画面上に表示されている motch daemon が緑色になり，motch daemon をインストールした PC から生存報告が行われていることが確認できる (図 12)．ここで利用者が Arduino を接続すると，デバイス一覧画面の下部に自動追加されたデバイスとして Arduino が表示される．利用者が「承認」ボタンを押して自動追加を承認すると，デバイス一覧画面に Arduino が追加される (図 13)．また，利用者がこの IoT システムを Raspberry Pi [19] に移植し農場に配置したいと考えた時，利用者は同様にして新しい motch daemon を追加し，Raspberry Pi にインストールする．インストールが終了すると，Raspberry Pi の生存がデバイス一覧画面上で確認できるようになる．この Raspberry Pi に Arduino UNO を繋ぎかえると，デバイス一覧画面上の接続関係が自動的に更新され，PC ではなく Raspberry Pi の配下に Arduino UNO が接続されている様子が確認できるようになる (図 14)．

表 1: motch でサポートできる IoT デバイス

| 分類 | 対応方針 | デバイス例 |
|---|---|---|
| Linux が動作するシングルボードコンピュータ | motch daemon をインストール | Raspberry Pi, Raspberry Pi Zero, BeagleBone, Tinker Board, NanoPi, ZeroPi, pcDuino, PINE A64+, ROCK64, LattePanda, LicheePi Nano |
| Arduino + 内蔵ネットワークインタフェース | motch SDK を導入 | ESP-WROOM-02, ESP-WROOM-32, Arduino Yun, GR-LYCHEE |
| Arduino + 外部ネットワークインタフェース | motch SDK を導入 | Arduino UNO, Arduino Nano, Arduino M0, Spresense, chipKIT uC32, GR-CITRUS, GR-KAEDE |
| mbed + 内蔵ネットワークインタフェース | motch SDK を導入 | mbed LPC1768, STM32 Nucleo Board, SeeedStudio Arch, Arch Pro, GR-PEACH, GR-LYCHEE |
| 近距離無線モジュール | motch daemon に接続することで対応 | TWE-LITE, MONOSTICK, Xbee ZB S2C, Xbee3 Zigbee3.0, Xbee Wi-Fi |
| USB デバイス | motch daemon に接続することで対応 | ESP-WROOM-02, ESP-WROOM-32, GR-LYCHEE, micro:bit, Arduino UNO, Arduino Nano, Arduino M0, Spresense, chipKIT uC32, GR-CITRUS, GR-KAEDE, mbed LPC1768, STM32 Nucleo Board, SeeedStudio Arch, Arch Pro, GR-PEACH |

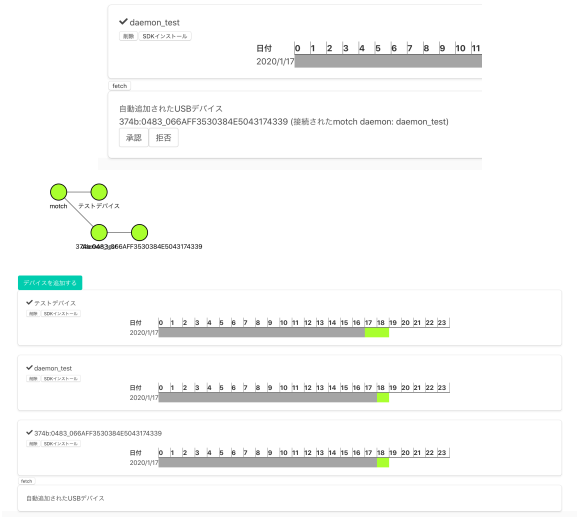


図 13: motch daemon による自動追加と利用者による追加承認

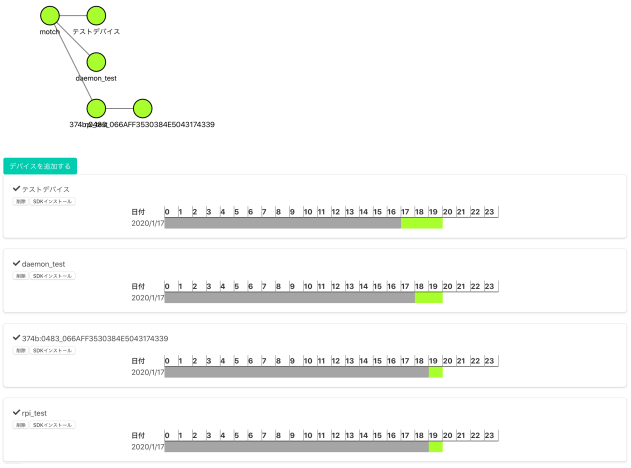


図 14: 接続関係が自動的に更新されている様子

### 4.2 motch でサポートできる IoT デバイス

IoT デバイスとしては，Raspberry Pi，Raspberry Pi Zero，Tinker Board，NanoPi，ZeroPi，Arduino Uno，Arduino Nano，BeagleBone，Arduino M0，pcDuino，PINE A64+，ROCK64，chipKIT uC32，Arduino Yun，Seeeduino Arch，LattePanda，LicheePi Nano，Spresense，GR-PEACH，GR-CITRUS，GR-KAEDE，GR-LYCHEE，STM32 Nucleo Board，micro:bit，mbed LPC1768，Arch Pro，ESP-WROOM-02，ESP-WROOM-32，Xbee ZB S2C，Xbee3 Zigbee3.0，Xbee Wi-Fi，MONOSTICK，TWE-LITE，USB カメラなど IoT サービスで使用されているデバイスは多種多様である．しかしながら，これらの多種多様なデバイスは表 1 のように 5 つに分類することができる．

Linux が動作するシングルボードコンピュータとしては，Raspberry Pi や Tinker Board などを想定している．Linux が動作することで既存のソフトウェア資産の有効活用がしやすいため，多くの IoT サービスで利用されている．これらの Linux が動作するシングルボードコンピュータでは，3.2 節に示した motch daemon で死活管理を行うことを想定している．

Arduino は，シングルボードコンピュータよりも低消費電力性や低コスト性が求められる領域で用いられることが多い．Arduino を用いたものは内蔵ネットワークインタフェースを介してネットワークに接続するもの，外部ネットワークインタフェースを介してネットワークに接続するもの，USB でホスト PC を介してネットワーク接続するものに大別することができる．「Arduino + 内蔵ネットワークインタフェース」と「Arduino + 外部ネットワークインタフェース」に関しては，3.2 節に示した motch SDK におけるマクロフックを利用した手法で対応ができる．Arduino

で用いられている Arduino 言語は C 言語に変換されてからコンパイルされるため，マクロフックを利用することができる．USB でホスト PC を介するものに関しては USB デバイスと同様の対応を取るので後述する．

mbed は，Arduino と同様にシングルボードコンピュータよりも低消費電力性や低コスト性が求めらる領域で用いられることが多い．Arduino と mbed 両方対応しているデバイスも多く，Arduino と mbed を抑えておけばマイコンを用いた IoT デバイスの大部分をサポートすることができる．mbed では，C++を用いて実装されるため，3.2 節に示した motch SDK におけるマクロフックを利用した手法で対応することができる．

近距離無線モジュールは，Arduino や mbed を用いた場合よりもさらに低消費電力かつ低コスト性が求められる IoT サービスで利用されることが多い．近距離無線モジュールのファームウェアを直接いじることができないことが多く，外部に接続したセンサの値を読み込んでシンクノードに送るだけという単機能のものが主流である．このような近距離無線モジュールに対しては，覆うの近距離無線モジュールのシンクノードがホスト PC に USB 接続されることに着目する．具体的には，3.2 節の motch daemon において，USB 接続されたシンクノードで受信する USB フレームをスニッフィングして近距離無線モジュールの生存を自動的に検出する仕組みを実現している．

USB デバイスは，無線インタフェースを持たずに直接ホスト PC に接続される Arduino 対応デバイス，mbed 対応デバイス，USB カメラなどを想定している．特に Raspberry Pi と USB 接続されたセンサを組み合わせた IoT サービスは多い．これらの USB デバイスに対しては，近距離無線モジュールと同様に，motch daemon による USB フレームスニッフィングの仕組みによって生存を自動検出する仕組みを実現している．

### 4.3 スケーラビリティの評価

サービスを運用する上ではユーザの増加に伴ってサーバコンピュータの性能を向上させていく，つまりスケールさせる必要がある．motch と同等のサービスを従来のサーバアーキテクチャを採用して AWS 上に構築した場合，EC2 におけるインスタンスの追加やインスタンスタイプのアップグレードによって性能向上を実現することができる．一方サーバレスアーキテクチャを採用して AWS 上に構築した場合，Lambda が自動的にスケールアウトすることで運用者による追加作業なしに性能向上を実現することができる．しかし Lambda を用いてスケールするシステムを作るためには，各関数がステートレスで実行されることや並列化などを考慮したシステム設計が必要となる．ここではスケーラビリティを考慮せずに製作した Lambda ストレートフォワード版，インスタンスタイプの異なる 2 つの EC2 版の 3 つを比較手法として，マイクロベンチマークで提案手法のスケーラビリティを評価した．比較する際に用いた評価指標は，デバイス数を変化させたときの各手法における生存確認プロセス完了までの時間とした．図 15 に評価結果を示す．デバイスの数の増加に伴って生存確認プロセス完了までの時間が伸びていく他の手法に比べて，提案手法は生存確認プロセスを一定時間で完了していることがわかる．

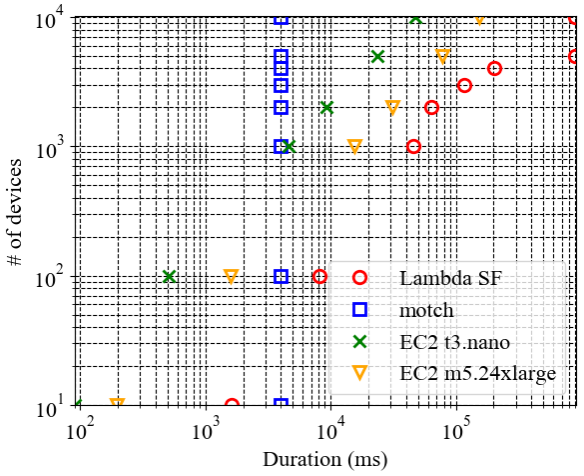


図 15: スケーラビリティの評価

### 4.4 コスト評価

AWS [20] が公開している各サービスの料金情報を元にして，提案システムの稼働に必要な経費を算出した．利用者が 1000 人，利用者あたりのデバイス数が 10 台，生存報告の間隔が 15 分，リージョンが東京リージョンという条件において，1 ヶ月の利用料金は約 124 ドルとなった．このうちデバイスの生存報告によって生じるコストが約 65 ドルであり，全体の半分程度を占めた．次に支配的なのがデータベースへの書き込みコストで，約 43 ドルであった．生存報告が行われるたびにデータベースへの書き込みが発生するため，データベースへの書き込みコストの大半はデバイスの生存報告によって生じるものである．よってデバイスの数が増加したり，生存報告の間隔が短くなったりすると，稼働に必要な経費が高騰すると考えられる．また，上述の条件の一部を変更し，利用者が 1 人，利用者あたりのデバイス数が 1 台とすると，1 ヶ月の利用料金は約 0.026 ドルとなった (表 2)．SaaS に登録されているデバイス数の変動によって利用料金が柔軟に変化するのは，サーバレスアーキテクチャを採用して従量課金で各サービスを利用するように設計したためである．

また，ユーザあたりのデバイス数を 10 台に固定し，ユーザ数を変化させたときの 1 ヶ月あたりの運用コストを図 16 に示す．EC2 を利用した場合と比較するために，時間あたりの利用料金が最も安い t3a.nano，t2，t3 系インスタンス

表 2: 1 台のみ登録されている場合の AWS 利用料金

| サービス | 金額 | 単価 ($/100 万回) | 回数 |
|---|---|---|---|
| Lambda | $3.1 \times 10^{-3}$ | 0.408 | $7.6 \times 10^3$ |
| API Gateway REST API | $1.5 \times 10^{-3}$ | 4.25 | $3.6 \times 10^2$ |
| API Gateway WebSocket | $1.3 \times 10^{-2}$ | メッセージ：1.26<br>接続料：0.315 | 5<br>$4.3 \times 10^4$ 分 |
| DynamoDB | $4.5 \times 10^{-3}$ | 書き込み：1.4269<br>読み込み：0.285 | $3.2 \times 10^3$<br>150 |
| AWS IoT MQTT | $3.6 \times 10^{-3}$ | メッセージ：1.20<br>接続料：0.096 | $2.9 \times 10^2$<br>$1.4 \times 10^2$ 分 |

の中で最も高価な t3.large，m5n 系インスタンスの中から m5n.large，最も高価な m5n.24xlarge を選び，1 ヶ月連続稼働させたときの運用コストと比較した．コスト評価の結果から，提案手法には以下の 2 つのメリットがあるといえる．一つは，ユーザ数が少ない時には EC2 で最も利用料金が安いインスタンスを 1 つ用いるよりも運用コストが安いということ，もう一つは，ユーザ数が増えてきても自動的にスケールすることにより，ユーザ数に適した計算リソースを常に利用できるということである．

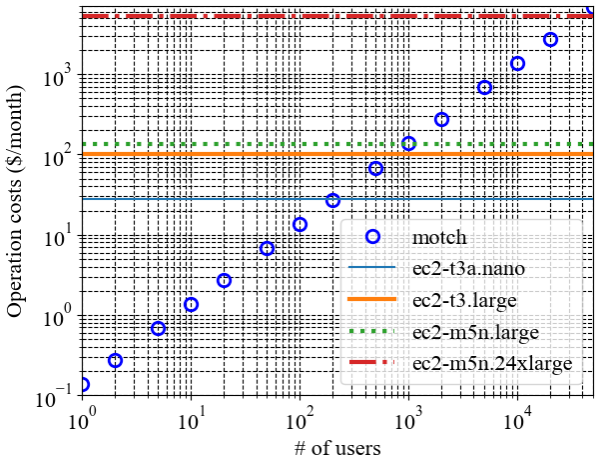


図 16: EC2 と motch の 1 ヶ月あたりの運用コスト

## 5. 結論

本稿では，DIY 的な IoT システムに向けた運用・維持・管理 SaaS である motch を提案した．第 2 節では IoT システムの運用・維持・管理に求められる要件と，それらを満たせない既存手法に存在する課題について述べた．第 3 節では IoT システムの運用・維持・管理に求められる要件を満たす，我々が実装した提案手法である motch の機能と実装について述べた．第 4 節では提案手法 motch の評価として，ユースケースによる評価，サポートできる IoT デバイスについての評価，システムの運用におけるスケーラビリティと運用コストの評価について述べた．評価の結果，実装やシステムへの登録が容易であること，あらゆるデバイスを管理に含められること，及びシステム運用におけるコストパフォーマンスが高いことがわかった．

## 謝辞

本研究は JSPS 科研費 (JP17KT0042)，NTT アクセスサービスシステム研究所および情報処理推進機構主催の未踏 IT 人材発掘・育成事業の支援の下で行った．

### 参考文献

[1] C.J. R. Oram, M. Garcia, and B.B. Park, “Data analysis of transit systems using low-cost IoT technology.,” IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), pp.497–502, 2017.

[2] B. Potter, G. Valentino, L. Yates, T. Benzing, and A. Salman, “Environmental monitoring using a drone-enabled wireless sensor network.,” Systems and Information Engineering Design Symposium (SIEDS), pp.1–6, October 2019.

[3] I. Lee, and K. Lee, “The Internet of Things (IoT): Applications, investments, and challenges for enterprises,” Business Horizons, vol.58, April 2015.

[4] L. Dan, C. Xin, H. Chongwei, and J. Liangliang, “Intelligent agriculture greenhouse environment monitoring system based on iot technology,” 2015 International Conference on Intelligent Transportation, Big Data and Smart City, pp.487-490, Dec 2015.

[5] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, “Devops,” IEEE Softw., vol.33, no.3, p.94–100, May 2016.

[6] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, “Internet of things: A survey on enabling technologies, protocols, and applications,” IEEE Communications Surveys Tutorials, vol.17, no.4, pp.2347-2376, 2015.

[7] “AWS IoT Applications & Solutions,” https://aws.amazon.com/iot/.

[8] “Azure IoT — Microsoft Azure,” https://azure.microsoft.com/ja-jp/overview/iot/.

[9] “Zabbix :: The Enterprise-Class Open Source Network Monitoring Solution,” https://www.zabbix.com.

[10] “Mackerel: A Revolutionary Server Management and Monitoring Service,” https://mackerel.io.

[11] “IoT プラットフォーム 株式会社ソラコム,” https://soracom.jp/.

[12] “さくらインターネットが提供する IoT プラットフォームサービス、sakura.io,” https://sakura.io.

[13] “ESP-WROOM-02 Overview — Espressif Systems,” https://www.espressif.com/en/products/hardware/esp-wroom-02/overview.

[14] “ESP32 Overview — Espressif Systems,” https://www.espressif.com/en/products/hardware/esp-wroom-32/overview.

[15] “Wio LTE Cat.1 - Seeed Wiki,” http://wiki.seeedstudio.com/Wio_LTE_Cat.1/.

[16] “Arduino Reference - setup(),” https://www.arduino.cc/reference/en/language/structure/sketch/setup/.

[17] “systemd/systemd: The systemd System and Service Manager,” https://github.com/systemd/systemd.

[18] “TCPDUMP/LIBPCAP public repository,” https://

www.tcpdump.org.
[19] “Teach, Learn, and Make with Raspberry Pi – Raspberry Pi,” https://www.raspberrypi.org/.
[20] “Amazon Web Services (AWS) - Cloud Computing Services,” https://aws.amazon.com/.